rainer OECHSLE

JAVA-KOMPONENTEN

GRUNDLAGEN, PROTOTYPISCHE REALISIERUNG UND BEISPIELE FÜR KOMPONENTENSYSTEME

EXTRA: Mit kostenlosem E-Book

HANSER

Unser günstiges Mini-Abo ...

3 Ausgaben für nur 15 Euro

... JavaSPEKTRUM jetzt abonnieren!

Sie möchten ergänzende regelmäßige Informationen zu den Entwicklungen in und mit der Programmiersprache Java erhalten?

Dann nutzen Sie zur Weiterbildung JavaSPEKTRUM – Magazin für professionelle Entwicklung und Integration von Embedded Systemen!

Die älteste Fachzeitschrift zu diesem Thema im deutschsprachigen Markt!

JavaSPEKTRUM ist das Praxismagazin für professionelle Softwarearchitekten und -Entwickler, die mit der Java-Plattform arbeiten. Das Heft stellt neue Entwicklungen und Konzepte vor, prüft die Relevanz für den täglichen Einsatz.
Es werden aktuelle Themen wie **Aspektorientierung**, **Agilität**, **Eclipse**, **Embedded** vertieft. **JavaSPEKTRUM** führte mit dem **IntegrationsSPEKTRUM** (I-Spektrum) eine Rubrik für Berichte über die Integration von Java mit anderen Plattformen ein.
Als Kolumnen für den Alltag des Lesers liefern **„Effective Java“** und **„Der Praktiker“** Beispiele für die erfolgreiche Softwareentwicklung. In der einzigartigen **Java 360 Grad** Kolumne werden nicht nur Java Specific Requests vorgestellt, sondern auch andere Einflussfaktoren auf den Java Community Process beleuchtet.
Regelmäßige Berichte über **Plug-Ins** runden das Bild ab.

Ja, ich will JavaSPEKTRUM testen:

Schicken uns diesen Bestellschein ausgefüllt per Fax (+49 (0)2241/2341-199) zurück oder gehen Sie online unter www.sigs.de/publications/aboservice.htm

Hiermit bestelle ich

❑ JavaSPEKTRUM (kostenloses Probeheft)

❑ JavaSPEKTRUM Mini-Abo (15 Euro)
(Das Mini-Abo wandelt sich automatisch in Jahres-Abo um, welches jederzeit kündbar ist.)

Firma: _____ Abteilung: _____

Name: _____ Vorname: _____

Straße: _____ L/PLZ/Ort: _____

Telefon: _____ Telefax: _____

E-Mail: _____ Position: _____

Zahlungswunsch: ❑ per Rechnung ❑ per Kreditkarte – wir rufen Sie an unter Tel.-Nr.: _____

❑ per Bankeinzug Kto.-Nr.: _____ BLZ: _____ Bank: _____

SWIFT: _____ IBAN: _____

Datum: _____ Unterschrift: _____

Die vollständigen AGBs finden Sie unter www.sigs-datacom.de/agb/

Verlagsanschrift: SIGS-DATACOM GmbH, Lindlaustr. 2c, 53842 Troisdorf,
Tel.-Nr.: +49 (0) 2241/2341-100, Fax-Nr.: +49 (0) 2241/2341-199, info@sigs-datacom.de

Hanser_Oechsle

Oechsle

Java-Komponenten

Rainer Oechsle

Java-Komponenten

Grundlagen, prototypische Realisierung und Beispiele für Komponentensysteme

Mit 103 Listings und zahlreichen Beispielen

HANSER

Prof. Dr. Rainer Oechsle
lehrt an der Hochschule Trier am Fachbereich Informatik.

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

Lektorat: Mirja Werner, M.A.
Herstellung: Dipl.-Ing. Franziska Kaufmann
Coverconcept: Marc Müller-Bremer, www.rebranding.de, München
Coverrealisierung: Stephan Rönigk
Satz, Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Printed in Germany

ISBN: 978-3-446-43176-8
E-Book-ISBN: 978-3-446-43591-9

www.hanser-fachbuch.de

Vorwort und Einleitung

Der Begriff Komponente kommt vom lateinischen Wort „componere“, was so viel wie „zusammensetzen“ heißt. Eine Komponente ist also ein Teil eines zusammengesetzten Ganzen, wobei die Einzelteile nicht nur irgendwie zusammengestellt, sondern in funktional sinnvoller Weise miteinander verbunden worden sind. Wenn beispielsweise die Einzelteile eines Fahrrads wie Rahmen, Bremse, Kette usw. einfach auf einen Haufen geworfen werden, kann man bekanntlich wenig mit dem Fahrrad anfangen – es wird in diesem Fall von den Menschen noch nicht einmal als Fahrrad wahrgenommen.

Die Vorstellung von Komponenten im Software-Bereich entstand bereits Ende der 60er Jahre im Zusammenhang mit der sogenannten Software-Krise. Man hoffte damals, dass eine Software-Industrie für die Massenproduktion von Komponenten entstehen würde und dass diese Komponenten in den unterschiedlichsten Anwendungen wiederverwendbar wären. So wie Kinder mit denselben Bausteinen Häuser, Schiffe oder Flugzeuge bauen, sollte eine neue Anwendung lediglich durch das „Zusammenstecken“ vorproduzierter Software-Komponenten entwickelt werden, wobei man dazu im Idealfall nicht einmal programmieren muss. Diese Hoffnung hat sich zumindest bis jetzt nicht erfüllt. Dennoch ist Komponenten-Software immer noch ein aktuelles Thema. So kann man beispielsweise die momentan weit verbreiteten Software-Plattformen für Enterprise Java Beans oder auch Android als Komponenten-Software sehen. Die Tatsache, dass sich das Thema Komponenten-Software schon so lange hält und kein vorübergehender „Hype“ ist, wie manche andere Themen in der Informatik, zeigt, dass es wohl eine gewisse Substanz haben muss.

Analog zum Begriff objektorientierte Programmierung gibt es den Ausdruck komponentenorientierte (manchmal auch komponentenbasierte) Programmierung. Der Komponentenbegriff ist allerdings nicht ganz unproblematisch: Während es einen breiten Konsens gibt, was ein Objekt ist (bei Unklarheiten empfehle ich ganz pragmatisch, einfach mal ein Java-Objekt mit Hilfe der Serialisierung abzuspeichern und sich die abgespeicherten Daten anzusehen), ist dies für eine Komponente weit weniger klar. Eine Ursache dafür dürfte u. a. sein, dass das, was eine Komponente ist, vom verwendeten Komponenten-Framework abhängt, und davon gibt es allein im Java-Umfeld eine ganze Reihe, wie dieses Buch zeigt.

In vielen Veröffentlichungen werden Komponenten dadurch charakterisiert, dass sie ausschließlich über Schnittstellen genutzt werden können (häufig werden auch „klar definierte“ Schnittstellen angeführt, was immer das auch heißen mag) und dass sie wiederverwendbar sind. Ich will nicht bestreiten, dass es Komponentensysteme gibt, auf die diese Beschreibung

zutrifft. Allerdings halte ich sie nicht für typisch. Für manche Komponentensysteme müssen keine Java-Schnittstellen implementiert werden, sondern Klassen aus vorgegebenen Klassen abgeleitet oder alternativ Methoden mit gewissen Annotationen versehen werden. Auch die Wiederverwendbarkeit ist oftmals keine bedeutsame Eigenschaft. Eine webbasierte Anwendung mit Servlets oder eine EJB-Anwendung, welche jeweils eine Komponente für das dazugehörige Komponenten-Framework darstellen, sind häufig nicht auf Wiederverwendbarkeit ausgelegt. Es geht in den genannten Fällen viel mehr um die Nutzung der Funktionen, die vom Komponenten-Framework bereitgestellt werden und die Anwendungsentwicklung dadurch erleichtern.

In diesem Buch soll eine klarere Vorstellung davon vermittelt werden, was Komponenten-Software im Java-Umfeld bedeutet. Im Einzelnen hoffe ich, dass Sie nach dem Durcharbeiten dieses Buches die folgenden Lernziele erreicht haben:

- Sie sollen die charakteristischen Merkmale von Java-Komponentensystemen kennen. Sie sollen verstanden haben, in welchen Ausprägungen diese Merkmale in den Beispielsystemen, die in diesem Buch behandelt werden, vorkommen. Ihnen sollen damit die Ähnlichkeiten, aber auch die Unterschiede der Frameworks deutlich geworden sein.
- Sie sollen ein Grundverständnis für den Umgang mit den Frameworks dieses Buches haben und einfache Komponenten dafür entwickeln können.
- Sie sollen sich mit diesem Grundverständnis leichter in eines der hier behandelten Frameworks in größerer Tiefe einarbeiten können. Auch sollen Sie sich mit einem anderen hier nicht vorgestellten Java-Framework schneller zurecht finden, indem Sie die Prinzipien dieses neuen Frameworks identifizieren, Ähnlichkeiten mit anderen Frameworks erkennen und davon profitieren können.
- Sie sollen aber nicht nur als Anwendungsentwickler und -entwicklerin die behandelten Frameworks nutzen können, sondern Sie sollen darüber hinaus auch verstehen, wie diese Frameworks ihre Funktionen realisieren. Aus diesem Verständnis heraus sollen Sie dann auch in der Lage sein, eigene Frameworks in Java zu entwickeln.

Nachdem Sie nun die Ziele kennen, die Sie erreichen sollen, wird die Gliederung des Buches vorgestellt. Das Buch besteht aus drei Teilen:

- Teil 1: Java-Grundlagen: Im ersten Teil werden die Grundlagen der Sprache Java behandelt, deren Kenntnis für das Thema Komponenten-Software unverzichtbar ist. Zwar wendet sich das Buch an Personen, die schon einigermaßen gute Java-Kenntnisse haben sollten. Aus Erfahrung weiß ich aber, dass manchmal auch Personen, die sich schon Jahre mit Java beschäftigen, die im Teil 1 behandelten Grundlagen nicht oder nur ungenügend beherrschen. Im Einzelnen geht es um Reflection in Kapitel 2, Annotationen in Kapitel 3, dynamische Proxies in Kapitel 4 sowie um das Klassenladen und das sogenannte Hot Deployment in Kapitel 5. Da Reflection einen guten Teil komplizierter geworden ist durch Generics und ich Reflection doch einigermaßen umfassend behandeln will, wird dem Kapitel über Reflection ein Kapitel über Generics vorangestellt (Kapitel 1).
- Teil 2: Java-Komponenten: Der zweite Teil ist zwar der kleinste der drei Teile. Er bildet aber den Kern des Buches, weil darin in Kapitel 7 erläutert wird, was in diesem Buch unter einer Java-Komponente und einem Java-Komponenten-Framework verstanden werden soll. Damit sich die Leserinnen und Leser bei dieser allgemeinen Charakterisierung etwas Konkretes dazu vorstellen können, wird im Kapitel davor (Kapitel 6) ein einfaches,

selbst entwickeltes Komponenten-Framework mit einigen einfachen Beispielkomponenten vorgestellt.

- Teil 3: Beispiele für Java-Komponentensysteme: Im dritten und letzten Teil des Buches wird eine Reihe von konkreten Java-Komponentensystemen präsentiert. Es handelt sich dabei um Java Beans (Kapitel 8), OSGi (Kapitel 9), Eclipse (Kapitel 10), Applets (Kapitel 11), Servlets (Kapitel 12), Enterprise Java Beans (Kapitel 13), Spring (Kapitel 14), Ereignisbusse (Kapitel 15) und Android (Kapitel 16). Für jedes Framework werden neben einigen allgemeinen Bemerkungen konkrete Beispielkomponenten entwickelt. Dabei geht es nicht nur um das Zusammenwirken jeder einzelnen Komponente mit dem Framework, sondern auch und vor allem um das Zusammenspiel der Komponenten untereinander. Jedes Kapitel endet mit einer Bewertung, inwiefern das gerade besprochene Komponentensystem der allgemeinen Charakterisierung aus Kapitel 7 entspricht. Selbstverständlich werden Komponentensysteme in Kapitel 7 so charakterisiert, dass die Beispiele im dritten Teil des Buches einigermaßen gut dazu passen.

Nachdem Sie jetzt wissen, was der Inhalt dieses Buches ist, möchte ich auch darauf eingehen, was in diesem Buch nicht vorkommt. Wie der Titel des Buches angibt, geht es in diesem Buch ausschließlich um Komponentensysteme, die auf Java basieren. Somit wird das Beispiel, das in anderen Veröffentlichungen über Komponenten-Software immer erwähnt wird, nämlich COM/DCOM aus der Windows-Welt, hier nicht behandelt. Auch die zu einem gewissen Teil sprachunabhängigen Frameworks CORBA und Web Services, welche üblicherweise als Beispiel-Frameworks angeführt werden, sind in diesem Buch nicht zu finden. Ich habe mir die Freiheit genommen, nur Frameworks vorzustellen, die mir gefallen. Außerdem sollten Sie nicht erwarten, dass Sie nach dem Durcharbeiten des Buches für alle vorgestellten Frameworks Experte bzw. Expertin sind. Bitte machen Sie sich klar, dass es zu den meisten Frameworks jeweils separate Bücher gibt, deren Seitenumfang doppelt so groß ist wie der Umfang dieses Buches. Somit finden Sie in diesem Buch beispielsweise nichts zu der Installation der einzelnen Frameworks oder zum Vorgehen bei der Entwicklung und Installation (Deployment) eigener Komponenten für diese Frameworks mit Hilfe von Entwicklungsumgebungen wie Eclipse. Auch werden viele Funktionen, die diese Frameworks haben, verschwiegen. Das Ziel dieses Buches ist, nur das „Komponentenhafte“ jedes behandelten Frameworks herauszustellen. Insofern ist dieses Buch ein klassisches Lehrbuch: Es geht nicht um das tiefe Eindringen in ein spezielles Framework, sondern es geht darum, den Leserinnen und Lesern Grundprinzipien zu vermitteln und sie in die Lage zu versetzen, Ähnlichkeiten zwischen den Frameworks zu erkennen. Wenn Sie irgendwann beim Lesen denken: „das ist ja genauso wie bei …“, dann ist das ein durchaus beabsichtigter Effekt. Durch die Betonung der Prinzipien erhoffe ich mir eine Vermittlung von Kenntnissen, die über einen etwas längeren Zeitraum bedeutsam sind und nicht mit dem Erscheinen der nächsten Version eines bestimmten Frameworks nutzlos werden.

Das Buch wendet sich sowohl an bereits im Berufsleben stehende Java-Software-Entwicklerinnen und -Entwickler als auch an Studierende der Informatik und verwandter Studiengänge. Die Studierenden sollten nach Möglichkeit schon mindestens zwei Jahre Erfahrungen mit Java gesammelt haben. Insofern ist das Buch für Bachelor-Studierende ab dem dritten Studienjahr oder für Master-Studierende geeignet. Neben einem guten Grundverständnis der Sprache Java, das man in der Regel im ersten Studienjahr eines Informatikstudiums erwirbt, sollten insbesondere Kenntnisse im Bereich der parallelen Programmierung

(Stichwort: Java-Threads), der Programmierung grafischer Benutzeroberflächen mit Java (Stichwort: Java-Swing) und der Programmierung verteilter Anwendungen mit Hilfe des Fernmethodenaufrufs (Stichwort: Java-RMI) vorhanden sein. Es ist mir zwar etwas peinlich, aber dennoch möchte ich erwähnen, dass diejenigen Leserinnen und Leser, die mein anderes Buch „Parallele und verteilte Anwendungen in Java“ (ebenfalls im Hanser-Verlag erschienen) durchgearbeitet haben, das für dieses Buch nötige Vorwissen mitbringen sollten, das man aber selbstverständlich auch auf andere Weise erworben haben kann.

Damit sich Personen beiderlei Geschlechts angesprochen fühlen, verwende ich an einigen Stellen im Buch sowohl die männliche als auch die weibliche Form. An anderen Stellen verwende ich dagegen nur die männliche oder nur die weibliche Form. Ich habe nicht gezählt, wie oft welche Form vorkommt. Ich hoffe aber sehr, dass die Verwendung in etwa ausgewogen ist.

Auf der Web-Seite http://jk.hochschule-trier.de finden Sie weitere Informationen zu diesem Buch. Insbesondere können Sie von dieser Seite den Beispielcode des Buches in Form einer Zip-Datei herunterladen. Für nachträglich entdeckte Fehler, die leider nicht vermeidbar sind, gebe ich auf dieser Web-Seite sowohl die entdeckende Person als auch die Fehlerkorrektur an. Wenn auch Sie Fehler entdecken, so teilen Sie mir diese bitte in Form einer E-Mail mit, die Sie an oechsle@hochschule-trier.de adressieren. Ich bin an allen Arten von Fehlern interessiert. Dies schließt einfache Fehler wie ausgelassene oder vertauschte Buchstaben und Kommafehler genauso ein wie inhaltliche Fehler. Gerne können Sie auch Ihre positiven oder negativen Kommentare an die angegebene Adresse senden und mit mir dadurch ins Gespräch kommen.

Zum Abschluss dieses Vorworts bleibt mir die angenehme Pflicht, denjenigen Personen zu danken, die mich in irgendeiner Weise bei der Entstehung dieses Buches unterstützt haben: Ich danke den Studierenden des Master-Studiengangs Informatik an der Hochschule Trier, die meine Komponenten-Vorlesung gehört und kritisch begleitet haben, der Kollegin Gaby Elenz und den Kollegen Patrick Fries und Prof. Dr. Andreas Künkler vom Fachbereich Informatik der Hochschule Trier für ihre Unterstützung in vielfältiger Weise sowie vom Hanser-Verlag Herrn Dr. Martin Feuchte, Frau Mirja Werner und Frau Franziska Kaufmann für ihre Hilfe. Ganz besonders danke ich auch meiner Familie. Zur Fertigstellung des Buches musste nicht nur der Text geschrieben werden, sondern ich musste auch die Beispielprogramme entwickeln und ausprobieren, was ebenfalls beträchtliche Zeit in Anspruch genommen hat. Ich danke euch für eure Geduld.

Konz-Oberemmel im Januar 2013
Rainer Oechsle

Inhalt

I Teil 1: Java-Grundlagen

In diesem ersten Teil des Buchs geht es um diejenigen Grundlagen der Programmiersprache Java, die essenziell für Java-Komponenten sind, die aber in den ersten zwei Jahren einer Informatikausbildung nicht oder nur am Rande behandelt werden. Somit könnte dieser erste Teil auch für Personen, die glauben, Java gut zu kennen, unter Umständen überraschende und interessante Inhalte vermitteln. Unter anderem geht es um Reflection (Kapitel 2), Annotations (Kapitel 3) und Dynamic Proxies (Kapitel 4). Das Thema Reflection ist für sich genommen wichtig, es bildet aber auch eine wichtige Basis für das Verständnis von Annotations und Dynamic Proxies. Da ein tieferes Verständnis von Reflection ohne Generics nicht möglich ist, wird der Behandlung von Reflection ein Kapitel über Generics (Kapitel 1) vorangestellt. Ein letzter für Komponenten wichtiger Bereich ist Hot Deployment (Kapitel 5), also das Laden neuer Klassenversionen während der Ausführung eines Programms.

1 Generics

Generics werden in diesem Buch besprochen, da das Thema Reflection, welches im folgenden Kapitel behandelt wird, zum einen Generics benutzt, zum anderen aber durch sie ein gutes Stück umfangreicher und komplizierter geworden ist und ohne Kenntnisse von Generics nicht in der Tiefe verstanden werden kann. Generics sind ein schönes Beispiel dafür, wie eine kleine, scheinbar harmlose Erweiterung einer Programmiersprache eine ganze Reihe von Konsequenzen hat, die auf den ersten Blick ganz und gar nicht offensichtlich sind. Insofern ist es sehr lehrreich, sich mit Generics näher auseinanderzusetzen.

1.1 Einleitung und Motivation

Vor der Einführung des Generics-Sprachkonzepts durch Java 5 sah die Klasse ArrayList beispielsweise so aus (stark verkürzt):

```
public class ArrayList
{
    public boolean add(Object o) {...}
    public Object get(int index) {...}
}
```

Damit konnten in einer ArrayList Objekte jeder beliebigen Klasse gespeichert werden. Es ist aber z. B. nicht möglich, die Benutzung so einzuschränken, dass in einem bestimmten ArrayList-Objekt nur Integers und in einem anderen nur Strings gespeichert werden dürfen. Seit Java 5 ist dies möglich. Eine ArrayList sieht jetzt so aus (wieder stark verkürzt):

```
public class ArrayList<T>
{
    public boolean add(T o) {...}
    public T get(int index) {...}
}
```

T ist dabei ein (formaler) Typparameter für die Klasse ArrayList. Beim Erzeugen eines ArrayList-Objekts muss T konkretisiert werden (z. B. durch die Klasse Integer):

```
ArrayList<Integer> intArray = new ArrayList<Integer>();
```

Seit Java 7 muss der Typ beim new-Operator nicht wiederholt werden, falls dieser aus dem Kontext gefolgert werden kann:

```
ArrayList<Integer> intArray = new ArrayList<>();
```

Die leeren, spitzen Klammern <> werden als Diamant-Operator bezeichnet. Wir werden in diesem Buch den Diamant-Operator nicht benutzen, sondern der Deutlichkeit halber den konkreten Typ immer explizit angeben.

Nachdem die ArrayList intArray nun auf die eine oder andere Art (mit oder ohne Diamant-Operator) erzeugt wurde, muss beim Aufruf der Methode add ein Integer-Objekt als Parameter angegeben werden, denn für dieses ArrayList-Objekt ist der Typparameter T jetzt auf Integer festgelegt:

```
intArray.add(new Integer(3));
```

Analog ist der Rückgabetyp der Methode get, wenn man sie auf unser ArrayList-Objekt anwendet, Integer. Das heißt, man kann ohne Casting den Rückgabewert einer Variablen des Typs Integer zuweisen:

```
Integer i1 = intArray.get(0);
```

Wegen des Auto-Boxing- bzw. Auto-Unboxing-Sprachkonzepts von Java kann statt Integer auch der primitive Datentyp int verwendet werden:

```
intArray.add(3);
int i2 = intArray.get(0);
```

Der Java-Compiler macht im Fall des Auto-Boxings (1. Zeile oben) aus der primitiven 3 ein Integer-Objekt, das den Wert 3 enthält. Ebenso fügt der Compiler in der 2. Zeile oben den Aufruf einer Methode (intValue) ein, die aus dem von get zurückgelieferten Integer-Objekt den primitiven Wert des Typs int extrahiert (Auto-Unboxing). Das heißt, der Compiler ändert die beiden obigen Zeilen in diese:

```
intArray.add(new Integer(3));
int i2 = intArray.get(0).intValue();
```

Objekte eines anderen Typs als Integer können in intArray nicht gespeichert werden. Theoretisch wären noch Objekte vorstellbar, deren Klasse aus Integer abgeleitet ist. Da aber die Klasse Integer final ist, kann aus Integer keine Klasse abgeleitet werden. Es geht also nur Integer und die folgenden Zeilen führen beide zu einem Syntaxfehler:

```
intArray.add("hallo"); //Syntaxfehler
intArray.add(new Object());//Syntaxfehler
```

1.2 Erste Beispiele

Somit können wir nun eine erste eigene Klasse mit Generics schreiben, welche lediglich ein Container für ein Objekt eines parametrisierten Typs T ist und entsprechende Getter- und Setter-Methoden bereitstellt (siehe Listing 1.1).

Listing 1.1 Einfaches Beispiel einer Generics-Klasse

```
package javacomp.basics;

public class GenericClass<T>
{
    private T content;

    public T getContent()
    {
        return content;
    }

    public void setContent(T content)
    {
        this.content = content;
    }
}
```

Die Klasse kann nun so verwendet werden, dass man ein Objekt erzeugt, in dem nur Strings, und ein anderes Objekt, in dem nur Integers abgelegt werden können:

```
GenericClass<String> stringContainer = new GenericClass<String>();
stringContainer.setContent("hallo");
String s = stringContainer.getContent();
System.out.println(s);

GenericClass<Integer> intContainer = new GenericClass<Integer>();
intContainer.setContent(3); //Auto-Boxing
int i = intContainer.getContent(); //Auto-Unboxing
System.out.println(i);
```

In einem zweiten Beispiel wollen wir mit Generics eine Klasse entwickeln, die nicht nur ein einziges Objekt eines parametrisierten Typs T, sondern beliebig viele Objekte des Typs T in einem Feld (Array) speichern kann. Es soll ein Keller (Stack) realisiert werden, der die Methoden push zum Hinzufügen und pop zum Entfernen eines Elements besitzt. Ein Keller funktioniert bekanntlich nach dem LIFO-Prinzip (Last In First Out), was bedeutet, dass immer das zuletzt hinzugefügte Element als erstes wieder entfernt wird. Wir starten mit einer generischen Klasse mit Typparameter T, die ein Attribut des Typs T[] zum Speichern der Objekte hat (außerdem ein Attribut vom Typ int, das den Index angibt, an dem das nächste Element im Feld gespeichert wird; dieser Wert ist immer auch die Anzahl der aktuell vorhandenen Elemente). Der erste Ansatz, das Attribut vom Typ T[] im Konstruktor durch `new T[length]` zu initialisieren, scheitert, da eine solche Anweisung nicht möglich ist und zu einem Syntaxfehler führt („Cannot create a generic array of T“ – eine Begründung dafür folgt später). Wir verwenden stattdessen `new Object[length]` und casten auf T[]. Das Feld wird in Listing 1.2 mit einer initialen Größe erzeugt, die als Konstruktorpara-

meter angegeben wird. Sollte in der Methode push festgestellt werden, dass das Feld voll ist, wird ein neues Feld der doppelten Größe angelegt und alle vorhandenen Elemente werden in das neue Feld kopiert.

Listing 1.2 Keller (Stack) als Generics-Klasse (1. Variante)

```
package javacomp.basics;

public class GenericStack<T>
{
    private int top;
    private T[] stack;

    public GenericStack(int initialLength)
    {
        top = 0;
        stack = (T[])new Object[initialLength];//Warnung
    }

    public void push(T o)
    {
        if(top == stack.length)
        {
            //Keller auf das Doppelte vergrößern:
            T[] newStack = (T[])new Object[2*stack.length];//Warnung
            for(int i = 0; i < stack.length; i++)
            {
                newStack[i] = stack[i];
            }
            stack = newStack;
        }
        stack[top++] = o;
    }

    public T pop()
    {
        if(top > 0)
        {
            T result = stack[top-1];
            stack[top-1] = null; //garbage collection
            top--;
            return result;
        }
        else
        {
            return null;
        }
    }
}
```

Die Implementierung funktioniert zwar, erzeugt aber beim Casten auf T[] im Konstruktor und in der Methode push jeweils eine Warnung („Type safety: Unchecked cast from Object[] to T[]“). Diese Problematik wird in Listing 1.2 durch einen entsprechenden Kommentar am Zeilenende gekennzeichnet.

Um den Warnungen zu entgehen, kann man versuchen, den Typ des Attributs der Klasse von T[] auf Object[] zu ändern (siehe Listing 1.3). Dadurch benötigt man im Konstruktor und

in der Methode push kein Casting mehr und die Warnungen sind weg. Prima! Will man aber den Rückgabetyp T in der Methode pop beibehalten, was ja sinnvoll ist, so kommt man an einem Casting in dieser Methode nicht vorbei. Und siehe da: Die Warnung („Type safety: Unchecked cast from Object to T“) ist wieder da!

Listing 1.3 Keller (Stack) als Generics-Klasse (2. Variante)

```
package javacomp.basics;

public class GenericStack<T>
{
    private int top;
    private Object[] stack;

    public GenericStack(int initialLength)
    {
        top = 0;
        stack = new Object[initialLength];
    }

    public void push(T o)
    {
        if(top == stack.length)
        {
            // Keller vergroessern auf das Doppelte
            Object[] newStack = new Object[2 * stack.length];
            for(int i = 0; i < stack.length; i++)
            {
                newStack[i] = stack[i];
            }
            stack = newStack;
        }
        stack[top++] = o;
    }

    public T pop()
    {
        if(top > 0)
        {
            T result = (T) stack[top-1];//Warnung
            stack[top-1] = null; //garbage collection
            top--;
            return result;
        }
        else
        {
            return null;
        }
    }
}
```

Auch diese Variante funktioniert. Allerdings müssen wir auch bei dieser Variante eine Warnung in Kauf nehmen. Eine echte Lösung gibt es in diesem Fall nicht. Als Ausweg kann ich nur empfehlen, sich für die erste oder zweite Variante zu entscheiden und die Annotation `@SuppressWarnings("unchecked")` vor die Methoden zu schreiben, die eine Warnung erzeugen.

1.3 Mehrere Typparameter

In den bisherigen Beispielen hatten die Klassen immer nur einen einzigen Typparameter. Selbstverständlich kann eine Klasse beliebig viele solcher Typparameter haben:

```
class C<T1, T2, T3> {...}
C<Integer,Double,String> c1 = new C<Integer,Double,String>();
C<Integer,String,String> c2 = new C<Integer,String,String>();
```

Auch sollte selbstredend sein, dass man nicht nur generische Klassen, sondern auch generische Schnittstellen definieren kann.

Weiterhin können generische Typen geschachtelt werden, indem als konkreter Typ für einen Typparameter wieder eine generische Klasse eingesetzt wird:

```
class A {...}
class B {...}
class C<T1, T2, T3> {...}
C<A,B,C<A,B,C<A,B,C<A,B,B>>>> c = new C<A,B,C<A,B,C<A,B,C<A,B,B>>>>();
```

Das ist nicht besonders übersichtlich, aber möglich.

Parametrisierte Typen von Klassen dürfen übrigens im Zusammenhang mit static nicht verwendet werden. So ist das Folgende nicht möglich:

```
class GenericClass<T>
{
    private static T t; //Syntaxfehler
    public static void m(T t) //Syntaxfehler
    {
    }
}
```

1.4 Einschränkung parametrisierter Typen

Bisher war es so, dass beim Erzeugen eines Objekts einer generischen Klasse jede beliebige Klasse für den Typparameter T angegeben werden konnte. Man kann dies aber bei Bedarf auch auf eine bestimmte Klasse und alle von ihr abgeleiteten Klassen einschränken. Betrachten wir als Beispiel dazu die folgenden vier Klassen:

```
class MyClass1 {...}
class MyClass2 extends MyClass1 {...}
class MyClass3 extends MyClass2 {...}
class GenericClass<T extends MyClass1> {...}
```

Es ist damit möglich, folgende Objekte der Klasse GenericClass zu erzeugen:

```
GenericClass<MyClass1> a = new GenericClass<MyClass1>();
GenericClass<MyClass2> b = new GenericClass<MyClass2>();
GenericClass<MyClass3> c = new GenericClass<MyClass3>();
```

Es ist aber nicht möglich, eine konkrete Klasse als Typparameter anzugeben, die außerhalb des Vererbungsbaums liegt, der MyClass1 als Wurzel hat. Die folgende Anweisung ist zum Beispiel syntaktisch falsch („Bound mismatch: The type String is not a valid substitute …“):

```
GenericClass<String> d = new GenericClass<String>();
```

Wenn wir also bisher eine Klasse C<T> definiert haben, dann war das eine vereinfachende Schreibweise für C<T extends Object>.

Es ist auch möglich, mehrfache Einschränkungen anzugeben:

```
class A {...}
interface I1 {...}
interface I2 {...}
class GenericClass<T extends A & I1 & I2> {...}
```

Die zuerst angegebene Einschränkung (im obigen Beispiel A) kann eine Klasse oder eine Schnittstelle sein. Alle weiteren Einschränkungen (nach &) müssen Schnittstellen sein.

1.5 Umsetzung des Generics-Konzepts

Um einige Einschränkungen der Generics in Java zu verstehen, ist es hilfreich zu wissen, wie dieses Sprachkonzept realisiert wird. Generics gehörten nicht von Anfang an zum Sprachumfang von Java, sondern wurden erst in Version 5 eingeführt. Ein Ziel dabei war, dass die JVM (Java Virtual Machine), welche den vom Java-Compiler generierten Byte-Code interpretiert, dafür nicht geändert werden sollte. Mit anderen Worten: Die JVM weiß nichts von Generics und der Byte-Code enthält keine Angaben, wo im Quellcode Generics verwendet wurden. Man bezeichnet die Entfernung der Generics-Eigenschaften aus dem Byte-Code als Typlöschung (Type Erasure). Bitte verwechseln Sie den Byte-Code nicht mit den Informationen über Klassen, Methoden usw., die über Reflection erfragt werden können; über Reflection kann man sehr wohl die Generics-Eigenschaften herausfinden, wie im folgenden Kapitel zu sehen sein wird. Der Compiler prüft zwar, ob die Syntax auch bzgl. Generics korrekt ist (Beispiele für Warnungen und Syntaxfehler im Zusammenhang mit Generics wurden zuvor schon präsentiert). Im erzeugten Byte-Code einer Klasse werden aber alle Vorkommen eines parametrisieren Typs T durch Object (für eine Klasse C<T>) bzw. durch A (für eine Klasse C<T extends A>) ersetzt. Zu einer parametrisierten Klasse gibt es auch nur ein einziges Class-Objekt (s. Kapitel über Reflection), und es wird dafür auch nur einmal Byte-Code erzeugt (nicht für jedes Einsetzen einer konkreten Klasse als Typparameter).

Wegen der oben beschriebenen Typlöschung (Type Erasure) kann man einem GenericStack-Objekt nicht mehr ansehen, für welchen konkreten Typ es erzeugt wurde. Man kann zwar überprüfen, ob ein GenericStack-Objekt beispielsweise nur String-Elemente enthält. Aber man kann daraus nicht folgern, ob es als GenericStack<String> angelegt wurde und nur Strings enthalten darf oder ob es als GenericStack<Object> angelegt wurde, momentan nur zufällig Strings enthält und zukünftig auch mit Objekten jeder beliebigen anderen Klasse

befüllt werden darf. Die Überprüfung von instanceof auf einen generischen Typ ist deshalb nicht möglich:

```
c instanceof GenericStack<String> //Syntaxfehler
```

Ebenso problematisch ist das Casten. Ein Casten eines Objekts auf z. B. GenericStack<String> erzeugt im Gegensatz zu instanceof lediglich eine Warnung, ist aber immerhin möglich (wie zuvor beschrieben können solche Warnungen durch eine entsprechende Annotation unterdrückt werden). Aber auch in diesem Fall kann eine vollständige Überprüfung des Castens, die auch den Typ des Typparameters untersucht, zur Laufzeit nicht erfolgen:

```
Object obj1 = new GenericStack<Integer>(10);
GenericStack<String> obj2 = (GenericStack<String>) obj1; //Warnung
```

Der Operator new kann auf Typparameter nicht angewendet werden, denn man kann ja dafür nicht einfach Code erzeugen, in dem der Konstruktor von Object aufgerufen wird. Man kann auch keine Parameterüberprüfung vornehmen, da man nicht weiß, welche Konstruktoren später die konkrete Klasse mitbringt:

```
class GenericClass<T>
{
   private T t = new T(); //Syntaxfehler
   ...
}
```

Dass die Anweisung new T[length] ebenso nicht möglich ist, wurde schon erwähnt. Die Begründung dafür gibt es aber erst in Abschnitt 1.6.

Will man ein Objekt eines Typs erzeugen, den man zur Programmierzeit noch nicht kennt, so ist dies nicht grundsätzlich unmöglich in Java. Mit Hilfe der Reflection lässt sich dies realisieren, wie aber erst im folgenden Kapitel gezeigt werden wird.

Auf einen Typparameter T (Attribut, Parameter, lokale Variable) können aufgrund der Ersetzung durch Object (für eine Klasse C<T>) bzw. durch A (für eine Klasse C<T extends A>) alle Methoden der Klasse Object bzw. der Klasse A angewendet werden:

```
public class GenericClass<T>
{
    private T content;
    public void m()
    {
        String s = content.toString();
    }
}
```

Der Typ des Attributs content ist zwar der Typparameter T. Dies ist aber kein Hindernis dafür, dass die Methode toString der Klasse Object darauf angewendet wird.

Da im Byte-Code Informationen im Zusammenhang von Generics wegen der Typlöschung (Type Erasure) nicht mehr vorhanden sind, ist ein Überladen von Methoden wie im folgenden Beispiel nicht möglich:

```
class A {}
class B {}
class C
```

```
{
    public void m(ArrayList<A> list)
    {
    }
    public void m(ArrayList<B> list)
    {
    }
}
```

Der Compiler meldet für diesen Code folgenden Fehler: “Method m(ArrayList<A>) has the same erasure m(ArrayList<T>) as another method in type C”.

1.6 Typenkompatibilität und Wildcards

Eine interessante Frage ist, welche generischen Typen kompatibel zueinander sind. Diese Frage spielt bei einer Zuweisung eine Rolle (Typen auf der linken und rechten Seite einer Zuweisung) oder bei einem Methodenaufruf (Typen der formalen Parameter in einer Methodendeklaration und Typen der konkreten Parameter bei einem Methodenaufruf). Betrachten wir zunächst die Situation ohne Generics. Wie Sie sicher wissen, sind die folgenden zwei Code-Zeilen unproblematisch:

```
String s = new String("hallo");
Object o = s; //okay
```

In der zweiten Zeile ist der Typ links des Zuweisungszeichens Object, rechts davon String. Dies ist aber kein Problem, da jedes Objekt des Typs String auch ein Objekt des Typs Object ist. Das Umgekehrte gilt bekanntlich nicht:

```
Object o = new Object();
String s = o; //Syntaxfehler
```

Einer Variablen des Typs String kann kein Objekt des Typs Object zugewiesen werden (Syntaxfehler „Type mismatch: cannot convert from Object to String“). Es wäre zwar möglich auf String zu casten. Dann wäre man zwar den Syntaxfehler los, würde sich aber beim Ausführen des Programms eine ClassCastException einfangen.

Man würde nun vermutlich erwarten, dass die gezeigte Verträglichkeit im Zusammenhang mit Generics ähnlich ist, dass also einer Variablen eines Object-Stacks ein String-Stack zugewiesen werden könnte:

```
GenericStack<String> gss = new GenericStack<String>(10);
GenericStack<Object> gso = gss; //Syntaxfehler
```

Dies geht aber nicht, denn sonst könnte man über die Variable gso ein beliebiges Objekt auf den Keller legen. Wenn man über die Variable gss dann von demselben Keller dieses Objekt wieder entfernen würde, würde man fälschlicherweise einen String erwarten:

```
gso.push(new Object()); //nicht möglich, da oben Syntaxfehler
String s = gss.pop();//nicht möglich, da oben Syntaxfehler
```

Eine ähnliche Problematik hat man übrigens bei Feldern, wobei die Situation hier jedoch eine andere ist. Die folgenden Zuweisungen sind nämlich durchaus problemlos möglich:

```
String[] strArray = new String[10];
Object[] objArray = strArray;
```

Man könnte nun versucht sein, über die Variable objArray ein Objekt, das kein String ist, in dem String-Feld zu speichern und über die Variable strArray auf den vermeintlichen String zuzugreifen. Allerdings scheitert dies bereits beim Versuch des Abspeicherns eines Objekts, das kein String ist, in objArray:

```
objArray[0] = new Integer(1); //ArrayStoreException
String s = strArray[0]; //wird nicht mehr ausgeführt wegen Ausnahme
```

Obwohl dies kein Syntaxfehler ist, passiert beim Ausführen ein Laufzeitfehler in Form einer ArrayStoreException. Dies liegt daran, dass ein Feldobjekt, obwohl es nur eine Folge von Objektreferenzen enthält, den Typ dieser Feldelemente kennt, so dass zur Laufzeit geprüft und erkannt werden kann, ob die im Feld abzulegende Objektreferenz passend zum Typ des Felds ist. Dies dürfte vermutlich der Grund sein, warum die Anweisung `new T[]` für einen Typparameter T nicht möglich ist (vgl. Abschnitt 1.2), denn wenn dafür Code der Form `new Object[]` erzeugt würde und zur Laufzeit für T die Klasse String gesetzt wird, dann hat man eben kein String[]-Objekt, sondern ein Object[]-Objekt, was nicht das ist, was man erwarten würde.

Ein Casten eines Objekts auf einen generischen Typ wie z. B. GenericStack<Object> erzeugt zwar eine Warnung, ist aber möglich (wie zuvor beschrieben können solche Warnungen durch eine entsprechende Annotation unterdrückt werden):

```
GenericStack<String> gss = new GenericStack<String>(10);
Object o = gss;
GenericStack<Object> gso = (GenericStack<Object>) o;
```

Auf diese Weise kann das obige Szenario, das durch die Inkompatibilität von GenericStack<String> und GenericStack<Object> verhindert wurde, doch realisiert werden:

```
gso.push(new Integer(3));
String s = gss.pop(); //ClassCastException
```

Im Unterschied zum Beispiel oben wird beim Ablegen eines Integer-Objekts im GenericStack<String> über die Referenz gso keine Ausnahme erzeugt, da man dies wegen der Typlöschung zur Laufzeit nicht feststellen kann (im Gegensatz zu Feldern). Allerdings wird bei der Zuweisung des von pop zurückgelieferten Integer-Objekts an eine String-Variable eine ClassCastException geworfen.

Aufgrund der bisherigen Beschreibungen könnte man vermuten, dass es im Zusammenhang mit Generics keine Typkompatibilität gibt. Dies stimmt jedoch nicht, da es zum einen rohe Typen und zum anderen Wildcards gibt.

1.6.1 Rohe Typen

Der rohe Typ (Raw Type) zu einer Generics-Klasse liegt vor, wenn die Typinformation ganz weggelassen wird, also zum Beispiel so:

```
GenericStack gs;
```

Man kann Objekte dieses Typs erzeugen, erhält aber eine Warnung, dass man besser Generics verwenden sollte:

```
gs = new GenericStack(10); //Warnung
```

Außerdem kann an eine Variable des rohen GenericStack-Typs jedes GenericStack-Objekt zugewiesen werden (ohne Warnung, Syntax- oder Laufzeitfehler):

```
gs = new GenericStack<String>(10);
gs = new GenericStack<Integer>(20);
```

Auch kann instanceof auf den rohen Typ angewendet werden. Daran kann man erkennen, dass der rohe Typ nicht einem Typ entspricht, bei dem für den Typparameter Object eingesetzt wurde, denn mit Angabe eines konkreten Typparameters sind solche Zuweisungen und solche Tests mit instanceof nicht möglich, wie oben bereits dargelegt wurde.

Eine weitere Situation, bei der rohe Typen eingesetzt werden können, ist das Anlegen von Feldern generischer Objekte. Die folgende Anweisung ist z. B. nicht möglich:

```
GenericClass<String>[] a = new GenericClass<String>[10]; //Fehler
```

Der Grund ist hierbei auf der rechten Seite der Zuweisung zu suchen. Die Deklaration von a ist unproblematisch:

```
GenericClass<String>[] a;
```

Mit Hilfe des rohen Typs kann das Feld aber erzeugt werden, wenn auch mit einer Warnung:

```
GenericClass<String>[] a = new GenericClass[10]; //Warnung
```

Unter Verwendung der im Folgenden vorgestellten Wildcards lassen sich auch Felder generischer Objekte erzeugen.

1.6.2 Wildcards

Eine andere Form der Kompatibilität im Zusammenhang mit Generics ergibt sich durch Wildcards, die durch ein Fragezeichen notiert werden. Mit Wildcards hat man die Möglichkeit, dass man den Typ eines generischen Typparameters nicht angeben muss. Die folgenden Codezeilen sind in Ordnung und erzeugen weder eine Warnung noch einen Syntax- oder Laufzeitfehler:

```
GenericStack<String> gss = new GenericStack<String>(10);
GenericStack<?> gs = gss;
```

Die zweite Zeile könnte etwas ausführlicher auch so geschrieben werden:

```
GenericStack<? extends Object> gs = gss;
```

Das zu Beginn dieses Abschnitts beschriebene Szenario (push über Variable gso, pop über Variable gss) kann in diesem Fall allerdings nicht vorkommen, weil auf Wildcard-Variablen keine Methoden angewendet werden dürfen, deren Parametertyp ein Typparameter ist. Die folgende Zeile ist somit syntaktisch nicht korrekt:

```
gs.push(new Object()); //Syntaxfehler
```

Die eben angegebene Regel kennt allerdings eine Ausnahme: Wenn das Argument null ist, kann die Methode angewendet werden:

```
gs.push(null);
```

Falls nur der Rückgabetyp einer Methode ein Typparameter ist, so ist dies kein Problem:

```
Object o = gs.pop();
```

Der Rückgabetyp kann weiter eingegrenzt werden, falls nicht `? extends Object` angegeben wurde, sondern etwas Spezielleres:

```
class A {...}
class B extends A {...}
GenericStack<A> gsa = new GenericStack<A>(10);
gsa.push(new B());
GenericStack<? extends A> gs = gsa;
A a = gs.pop();
```

Wildcards können für die Typen lokaler Variablen, Attribute und Parameter sowie als Rückgabetypen verwendet werden. Eine Benutzung im Zusammenhang mit new ist problematisch. Die folgende Codezeile ist beispielsweise nicht möglich:

```
GenericStack<?> gs = new GenericStack<?>(); //Syntaxfehler
```

Allerdings sind die folgenden Codezeilen in Ordnung und ergeben keinerlei Probleme:

```
class A<T> {...}
GenericStack<A<?>> gs = new GenericStack<A<?>>();
gs.push(new A<String>());
A<?> a = gs.pop();
```

Sollte eine Klasse mehrere Typparameter haben, so können für einen oder für mehrere Wildcards angegeben werden:

```
class A<T1, T2> {...}
A<?, ?> a = new A<String,Integer>();
```

Einschränkungen von Wildcards können auch parametrisierte Typen sein:

```
class A {...}
class C<T>
{
    public T get() {...}
}
```

```
C<C<A>> cca = new C<C<A>>();
C<? extends C<A>> c = cca;
C<A> ca = c.get();
```

Auch parametrisierte Typen mit Wildcards können als Einschränkungen von Wildcards verwendet werden:

```
class A {...}
class C<T>
{
    public T get() {...}
}
C<C<A>> cca = new C<C<A>>();
C<? extends C<? extends A>> c = cca;
C<? extends A> ca = c.get();
```

Wie oben schon erwähnt wurde, kann man zwar keine Felder generischer Objekte mit konkretem Typ erzeugen. Mit Wildcards ist es aber möglich:

```
class X<T> {}
X<Integer>[] xArray1 = new X<Integer>[10]; //Syntaxfehler
X<?>[] xArray2 = new X<?>[10];
```

Schließlich soll noch erwähnt werden, dass Wildcards nicht nur über extends, sondern über das Schlüsselwort super auch auf die gegenteilige Weise eingeschränkt werden können. Man kann damit angeben, dass nur die in super angegebene Klasse oder deren Basisklassen typkompatibel sind:

```
class A {...}
class B extends A {...}
class C extends B {...}
class GenericClass<T> {...}
GenericClass<? super B> gc;
gc = new GenericClass<A>(); //okay
gc = new GenericClass<B>(); //okay
gc = new GenericClass<C>(); //Syntaxfehler
```

Die Einschränkung, dass auf eine mit einem Wildcard definierte Variable keine Methoden angewendet werden dürfen, die einen Typparameter haben (Ausnahme null), bezog sich auf Variablen, deren Wildcard mit extends verknüpft war. Bei super sieht die Sache anders aus (gc bezieht sich auf die oben mit super definierte Variable):

```
gc.setContent(new B());
gc.setContent(new A()); //Syntaxfehler
gc.setContent(new C()); //Syntaxfehler
```

Als Rückgabetyp bei mit super definierten Wildcard-Variablen kann man aber mit nicht mehr als Object rechnen:

```
A a = gc.getContent(); //Syntaxfehler
B b = gc.getContent();//Syntaxfehler
Object o = gc.getContent();
```

1.7 Vererbung

Im Folgenden beschäftigen wir uns mit Vererbungs- und Implementierungsbeziehungen im Zusammenhang mit Generics. Wenn eine Klasse aus einer generischen Klasse abgeleitet wird, dann kann der Typparameter für die Basisklasse festgelegt werden, so dass die abgeleitete Klasse keinen Typparameter mehr haben muss (aber haben kann, wenn sie nämlich einen neuen Typparameter definiert, der mit der Basisklasse nichts zu tun hat):

```
class A {...}
class B<T> {...}
class C extends B<A> {...}
```

Auf diese Weise kann man übrigens die Einschränkung, dass keine Felder für generische Objekte erzeugt werden dürfen, umgehen, denn Felder der Klasse, die keinen Typparameter mehr enthält, dürfen natürlich erzeugt werden (A, B und C beziehen sich auf die obigen Klassen):

```
B<A>[] bArray = new B<A>[10]; //Syntaxfehler
C[] c = new C[10];
```

Statt der Festlegung des Typparameters bei der Ableitung kann der Typparameter der Basisklasse auch offen bleiben. In diesem Fall muss dann die abgeleitete Klasse ebenfalls einen Typparameter haben:

```
class B<T> {...}
class C<T> extends B<T> {...}
```

Für die abgeleitete Klasse muss mindestens dieselbe Typeinschränkung gelten wie für die Basisklasse:

```
class A {...}
class B<T extends A> {...}
class C<T extends A> extends B<T> {...}
```

In den folgende Codezeilen ist der Typparameter in der abgeleiteten Klasse weniger stark eingeschränkt wie in der Basisklasse. Dies führt deshalb zu einem Syntaxfehler:

```
class A {...}
class B<T extends A> {...}
class C<T> extends B<T> {...} //Syntaxfehler
```

Natürlich sind auch Mischformen im Rahmen von Ableitungen möglich, wobei einige Typparameter der Basisklasse festgelegt werden, andere aber offen bleiben:

```
class A {...}
class B<T1, T2> {...}
class C<T> extends B<T, A> {...}
```

Es ist auch möglich, dass die abgeleitete Klasse zusätzliche Typparameter einführt. In den folgenden Codezeilen sind T1 und T3 neue Typparameter der Klasse C, während T2 der zweite Typparameter von C und gleichzeitig der erste Typparameter der Klasse B ist:

```
class A {...}
class B<T1, T2> {...}
class C<T1, T2, T3> extends B<T2, A> {...}
```

Ein Wildcard in der Basisklasse ist nicht möglich:

```
class A<T> {...}
class B extends A<?> {...} //Syntaxfehler
```

Bitte folgern Sie aus den Erläuterungen in Abschnitt 1.6 nicht, dass es grundsätzlich keine Typkompatibilität bei der Vererbung im Zusammenhang mit Generics gibt. Wie schon erläutert wurde, geht zwar nicht:

```
class A<T> {...}
A<Object> a = new A<String>(); //Syntaxfehler
```

Aber die „normale“ Kompatibilität ist bei Ableitung einer generischen Klasse B aus einer generischen Klasse A weiterhin gegeben:

```
class A<T> {...}
class B<T> extends A<T> {...}
A<String> a = new B<String>();
```

Selbstverständlich gelten alle Bemerkungen auch im Zusammenhang mit Schnittstellen, also bei der Ableitung einer Schnittstelle aus einer oder mehreren anderen Schnittstellen und bei der Implementierung einer oder mehrerer Schnittstellen durch eine Klasse. Eine parametrisierte Schnittstelle kann allerdings nur einmal implementiert werden und nicht mehrfach für unterschiedliche Typen, wie das folgende Beispiel zeigt:

```
interface I<T> {...}
class A {...}
class B {...}
class C implements I<A>, I<B> {...} //Syntaxfehler
```

Eigene Ausnahmeklassen, die z. B. aus Exception abgeleitet werden, dürfen übrigens keine generischen Klassen sein:

```
class MyException<T> extends Exception {} //Snytaxfehler
```

Ausnahmen dürfen aber Typparameter von Klassen oder Schnittstellen sein:

```
interface I <E extends Exception>
{
    public void m1() throws E;
    public void m2() throws E;
}
```

1.8 Generische Methoden

Nicht nur Klassen und Schnittstellen, sondern auch Methoden und Konstruktoren können Typparameter haben. Wir besprechen hier nur generische Methoden; das Gesagte lässt sich leicht auf generische Konstruktoren übertragen. Um zu verstehen, warum generische Methoden eingeführt wurden, betrachten wir als Beispiel die Aufgabe, den Inhalt von zwei GenericClass-Objekten (GenericClass siehe Listing 1.1) auszutauschen. Dies ist z. B. für ein GenericClass<String>- und ein GenericClass<Integer>-Objekt nicht möglich. Oder anders formuliert: Dies macht nur Sinn für zwei GenericClass-Objekte, die denselben Typ haben. Eine Lösung dieses Problems mit Hilfe einer generischen Klasse zeigt Listing 1.4.

Listing 1.4 Generische Klasse zum Austausch des Inhalts zweier GenericClass-Objekte

```
public class Exchange<T>
{
    public void exchange(GenericClass<T> o1,
                         GenericClass<T> o2)
    {
        T content1 = o1.getContent();
        T content2 = o2.getContent();
        o1.setContent(content2);
        o2.setContent(content1);
    }
}
```

Damit kann man z. B. den Inhalte zweier GenericClass<String>- und zweier GenericClass<Integer>-Objekte austauschen:

```
GenericClass<String> hallo = new GenericClass<String>();
hallo.setContent("hallo");
GenericClass<String> welt = new GenericClass<String>();
welt.setContent("welt");
Exchange<String> eString = new Exchange<String>();
eString.exchange(hallo, welt);

GenericClass<Integer> i1 = new GenericClass<Integer>();
i1.setContent(47);
GenericClass<Integer> i2 = new GenericClass<Integer>();
i2.setContent(11);
Exchange<Integer> eInteger = new Exchange<Integer>();
eInteger.exchange(i1, i2);
```

Ein Nachteil ist, dass man für jeden Typ, der als Typargument von GenericClass verwendet wird (im Beispiel String und Integer), ein eigenes Exchange-Objekt benötigt. So lässt sich das Exchange-Objekt eString nicht für GenericClass<Integer>-Objekte verwenden:

```
eString.exchange(i1, i2); //Syntaxfehler
```

Anders sieht es aus, wenn wir generische Methoden einsetzen. Bei generischen Methoden ist der Typparameter mit der Methode verknüpft. Listing 1.5 zeigt, wie sich das Beispiel aus Listing 1.4 mit Hilfe einer generischen Methode umsetzen lässt.

Listing 1.5 Generische Methode zum Austausch des Inhalts zweier GenericClass-Objekte

```
public class Exchange
{
    public <T> void exchange(GenericClass<T> o1,
                             GenericClass<T> o2)
    {
        T content1 = o1.getContent();
        T content2 = o2.getContent();
        o1.setContent(content2);
        o2.setContent(content1);
    }
}
```

In diesem Fall reicht ein einziges Objekt aus, um die Inhalte für unterschiedliche GenericClass-Objekte zu realisieren:

```
GenericClass<String> hallo = new GenericClass<String>();
hallo.setContent("hallo");
GenericClass<String> welt = new GenericClass<String>();
welt.setContent("welt");

GenericClass<Integer> i1 = new GenericClass<Integer>();
i1.setContent(47);
GenericClass<Integer> i2 = new GenericClass<Integer>();
i2.setContent(11);

Exchange e = new Exchange();
e.exchange(hallo, welt);
e.exchange(i1, i2);
```

Hier kann das Objekt e sowohl zum Tauschen der Inhalte zweier GenericClass<String>- als auch zweier GenericClass<Integer>-Objekte verwendet werden.

Übrigens könnte man im vorliegenden Fall, in dem keine Attribute in der Klasse Exchange vorhanden sind, die Methode exchange static machen, wie Listing 1.6 zeigt.

Listing 1.6 Generische Static-Methode zum Austausch des Inhalts zweier GenericClass-Objekte

```
public class Exchange
{
    public static <T> void exchange(GenericClass<T> o1,
                                    GenericClass<T> o2)
    {
        T content1 = o1.getContent();
        T content2 = o2.getContent();
        o1.setContent(content2);
        o2.setContent(content1);
    }
}
```

In diesem Fall ist gar kein Objekt nötig, um einen Tausch durchzuführen:

```
GenericClass<String> hallo = new GenericClass<String>();
hallo.setContent("hallo");
GenericClass<String> welt = new GenericClass<String>();
```

```
welt.setContent("welt");

GenericClass<Integer> i1 = new GenericClass<Integer>();
i1.setContent(47);
GenericClass<Integer> i2 = new GenericClass<Integer>();
i2.setContent(11);

Exchange.exchange(hallo, welt);
Exchange.exchange(i1, i2);
```

Nach wie vor ist es aber nicht möglich, zueinander unverträgliche Objekte als Parameter in einem Aufruf von exchange anzugeben:

```
Exchange.exchange(hallo, i1); //Syntaxfehler
```

Bei einem Aufruf einer generischen Methode sind alle Typen kompatibel, die zu beiden Parametern passen. Betrachten wir dazu die folgende Klasse X mit der generischen Methode m:

```
class X
{
    public static <T> T m(T t1, T t2)
    {
        ...
    }
}
```

Selbstverständlich ist Object als Rückgabetyp für T immer verträglich:

```
Object result = X.m(1, "hallo");
```

Wenn man nur Vererbung betrachten würde, dann könnte man auch eindeutig die spezifischste Klasse bestimmen, die zu beiden Parametern verträglich ist (im obigen Beispiel ist das Object). Da in Java aber auch Implementierungsbeziehungen vorkommen, kann auch eine Schnittstelle ein passender Typ sein. So kann als Rückgabetyp im obigen Beispiel die Schnittstelle Serializable verwendet werden, weil sowohl Integer als auch String diese Schnittstelle implementieren:

```
Serializable result = X.m(1, "hallo");
```

Im Allgemeinen kann es neben der spezifischsten Klasse mehrere Schnittstellen geben, die „am besten“ passen.

Die folgenden Codezeilen zeigen ein Beispiel dafür, dass die am besten passende Klasse nicht Object sein muss:

```
class A {...}
class B extends A {...}
A a = X.m(new A(), new B());
```

Übrigens sind bei generischen Methoden auch Einschränkungen für die Typparameter möglich:

```
class A {...}
class X
{
    public static <T extends A> T m(T t1, T t2)
```

```
    {
        ...
    }
}
```

Selbstverständlich kann eine generische Methode genau wie eine generische Klasse mehr als einen Typparameter haben.

Ein interessantes Beispiel einer generischen Methode stellt das folgende Beispiel dar:

```
class X<T> {}
class Y
{
    public static <T> X<T> newInstance()
    {
        return new X<T>();
    }
}
```

In diesem Beispiel kann der konkrete Typparameter nicht aus den Parametern gefolgert werden, da die Methode newInstance parameterlos ist. Möglich sind dann z. B. die folgenden Methodenaufrufe:

```
X<Integer> x1 = Y.newInstance();
X<String> x2 = Y.newInstance();
```

Vielleicht überraschend ist auch dieses möglich:

```
X<?> x3 = Y.newInstance();
```

Wir haben nun einige Beispiele generischer Methoden gesehen, die static waren. Dies ist kein Widerspruch zu den Bemerkungen am Ende von Abschnitt 1.3, wonach Typparameter im Zusammenhang mit static nicht verwendet werden dürfen. Diese Bemerkung bezog sich auf Typparameter von Klassen, während es sich bei den letzten Beispielen um Typparameter von Methoden handelte.

1.9 Überladen und Überschreiben

Ich hatte zu Beginn dieses Kapitels erwähnt, dass Generics eine Reihe von Konsequenzen für Java bringen. Sollte Sie das bisher Gesagte noch nicht genügend überzeugt haben, soll nun als weitere Argumentationshilfe für die durch Generics entstehende Komplexität zum Abschluss dieses Kapitels ein kleiner Blick auf das Überladen und Überschreiben von Methoden im Zusammenhang mit Generics geworfen werden. Zum Einstieg soll das Thema Überladen und Überschreiben kurz wiederholt werden. Im ersten Beispiel wird die Methode m in der Klasse X2 überladen, denn es gibt zwei Methoden namens m (eine wird von X1 geerbt, eine ist in X2 definiert), die sich durch die Typen ihrer Parameter unterscheiden:

```
class A {}
class B extends A {}
```

```
class X1
{
    public void m(A a)
    {
        System.out.println("X1.m(A)");
    }
}

class X2 extends X1
{
    public void m(B b)
    {
        System.out.println("X2.m(B)");
    }
}
```

Im folgenden Beispiel dagegen handelt es sich um Überschreiben, da in der abgeleiteten Klasse Y2 die Methode m mit genau derselben Signatur definiert wird, wie sie in der Basisklasse Y1 schon vorhanden ist:

```
class A {}
class B extends A {}

class Y1
{
    public void m(A a)
    {
        System.out.println("Y1.m(A)");
    }
}

class Y2 extends Y1
{
    public void m(A a)
    {
        System.out.println("Y2.m(A)");
    }
}
```

Um den unterschiedlichen Effekt zwischen Überladen und Überschreiben darzustellen, betrachten wir zunächst die folgenden Zeilen, in denen es um die überladene Methode m geht:

```
X2 x2 = new X2();
x2.m(new B());
X1 x1 = x2;
x1.m(new B()); //entscheidender Aufruf: X1.m
```

Im obigen Beispiel wird ein X2-Objekt erzeugt. Ruft man die Methode m mit einem Objekt der Klasse B als Parameter auf, so wird das m aus X2 aufgerufen. Wiederholt man denselben Aufruf für dasselbe Objekt, verwendet aber eine Referenz auf das X2-Objekt vom Typ X1, so wird das m aus X1 verwendet.

Anders ist es beim Überschreiben:

```
Y2 y2 = new Y2();
y2.m(new B());
Y1 y1 = y2;
y1.m(new B()); //entscheidender Aufruf: Y2.m
```

Hier wird bei beiden Aufrufen von m die Methode m der Klasse Y2 benutzt.

Kombiniert man die für sich nicht einfachen Konzepte des Überladens und Überschreibens einerseits und der Generics andererseits, so kann man leicht Programmcode schreiben, bei dem nur noch die absolut intimsten Java-Kenner zweifelsfrei wissen, was wirklich passiert. Wir wollen diesen Bereich nicht vollständig und systematisch behandeln, sondern nur einige Beispiele ohne nähere Erklärungen angeben, die unterstreichen sollen, wie schwierig das Thema Generics ist.

Betrachten wir zunächst die folgenden Codezeilen:

```
class A {}
class B<T extends A> {public void m(T t) {}}
class C extends B<A> {public void m(Object obj) {}}
-----------------------------------------------------
B<A> b = new C();
b.m(new A());
```

Welche Methode m wird aufgerufen? Um das zu entscheiden, muss man wissen, ob m in der Klasse C überladen oder überschrieben wird. Die Antwort lautet: m wird überladen. Deshalb wird das m der Klasse B aufgerufen. Überschreiben würde so gehen:

```
class A {}
class B<T extends A> {public void m(T t) {}}
class C extends B<A> {public void m(A a) {}}
```

Betrachten wir ein ähnliches, aber doch entscheidend anderes Beispiel:

```
class A {}
class B<T> {public void m(T t) {}}
class C extends B<A> {public void m(Object obj) {}} //Syntaxfehler
```

Der Java-Compiler meldet für diese Codezeilen den Syntaxfehler “Name clash: The method m(Object) of type C has the same erasure as m(T) of type B<T> but does not override it”.

Zum Schluss ein letztes, wieder leicht anderes Beispiel:

```
class A {}
class B<T> {public void m(T t) {}}
class C extends B<A> {public void m(A a) {}}
-----------------------------------------------------
B<A> b = new C();
b.m(new A());
```

In diesem Fall handelt es sich nun tatsächlich um ein Überschreiben, so dass in diesem Fall die Methode m der Klasse C aufgerufen wird. Die Umsetzung in Byte-Code, in dem keine Informationen über Generics mehr vorhanden sind, ist allerdings nicht ganz trivial. Denn wenn man die Typlöschung naiv anwenden würde, ergäbe sich Byte-Code, der zurückübersetzt in Java so aussehen würde:

```
class A {}
class B {public void m(Object t) {}}
class C extends B {public void m(A a) {}}
```

Das wäre dann aber ein Überladen und kein Überschreiben. Deshalb ist die Umsetzung in Byte-Code doch etwas trickreicher als hier gezeigt. Auch wenn hier nicht mehr erläutert

werden soll, wie dieses Problem tatsächlich gelöst wird, denke ich, dass trotzdem klar geworden sein sollte, was mit diesen Beispielen ausgedrückt werden sollte: Generics haben ihre Tücken.

1.10 Fazit

Im ersten Abschnitt dieses Kapitels wurde zur Motivation für die Einführung von Generics das Beispiel der Klasse ArrayList herangezogen: Der Programmcode dieser Klasse muss nur einmal geschrieben werden; man kann aber mehrere Objekte dieser Klasse anlegen, wobei in einem ArrayList-Objekt beispielsweise nur Integer-Objekte und in einem anderen ArrayList-Objekt nur String-Objekte gespeichert werden dürfen. Dies ist eine durchaus vernünftige Idee, deren Umsetzung nicht allzu schwierig erscheint. Sie haben dann im weiteren Verlauf des Kapitels aber hoffentlich einen kleinen Eindruck davon bekommen, wie kompliziert der Umgang mit Generics werden kann, wobei noch nicht einmal alle Stolpersteine im Zusammenhang mit Generics erwähnt wurden. Auch wenn ich mir damit vielleicht keine Freunde mache, sage ich aus diesem Grund: Meine ganz persönliche Einschätzung ist, dass der Aufwand für das sichere Beherrschen des Themas Generics weitaus größer ist als sein Nutzen. Wenn es nach mir gegangen wäre, hätte man dieses Sprachkonzept in Java nicht gebraucht. Selbstverständlich dürfen Sie, liebe Leserinnen und Leser, eine davon abweichende Meinung haben. Wie immer man darüber denken mag: Generics sind nun mal ein Teil von Java, und sie spielen für dieses Buch eine Rolle. Wir müssen uns also mit diesem Konzept auseinandersetzen, ob wir es mögen oder nicht.

2 Reflection

Mit Hilfe von Reflection kann man zur Laufzeit unterschiedliche Informationen über eine Klasse erfragen: Welche Attribute hat eine Klasse? Wie heißen ihre Attribute, welchen Typ und welche Sichtbarkeit haben sie? Welche Konstruktoren (Parametertypen, Ausnahmen, Sichtbarkeit) hat eine Klasse? Welche Methoden (Namen, Parametertypen, Rückgabetyp, Ausnahmen, Sichtbarkeit) hat eine Klasse? Welche Schnittstellen implementiert eine Klasse und von welcher Klasse ist sie abgeleitet? Dies ist für jede beliebige Klasse möglich, ohne dass man die betreffende Klasse beim Schreiben seines Codes schon kennen muss. Auch muss der Quellcode der zu untersuchenden Klasse zur Laufzeit nicht vorhanden sein; was man zur Laufzeit braucht, ist die entsprechende Class-Datei, die sich auch in einem Jar-Archiv befinden kann. Der Name der Klasse muss im Programm nicht „fest eingebrannt" sein („fest eingebrannt" ist beispielsweise die Klasse X, wenn man in seinem Programm `new X()` benutzt; der Klassenname kann nach dem Übersetzen des Programms zur Laufzeit nicht mehr dynamisch geändert werden). Es ist im Gegenteil möglich, dass man den Namen der zu untersuchenden Klasse dynamisch als String, z. B. aus einer Konfigurationsdatei oder von der Tastatur, einliest und damit die oben beschriebenen Informationen erfragen und anzeigen kann.

Mit Hilfe von Reflection kann man aber nicht nur Informationen auslesen, sondern man kann damit auch die Attribute eines Objekts einer Klasse ändern, ohne dass man beim Programmieren schon weiß, welche Attribute die Klasse hat. Ferner kann man in seinem Programm Objekte einer Klasse erzeugen, ohne dass man beim Programmieren weiß, wie die Klasse heißt und welche Parametertypen der Konstruktor benötigt. Entsprechendes gilt für das Aufrufen von Methoden.

Im Zusammenhang mit Java-Komponenten kann ein Komponenten-Framework mit Hilfe von Reflection zum Beispiel ein Objekt einer neuen Komponente erzeugen, deren Klasse beim Schreiben des Komponenten-Framework-Codes noch nicht bekannt war. Dem Komponenten-Framework muss zur Laufzeit lediglich der Name der Komponentenklasse mitgeteilt werden, dann kann das Framework Objekte davon erzeugen. Wie Sie im Laufe dieses Kapitels sehen werden, spielen Generics im Zusammenhang mit Reflection eine gewisse Rolle.

2.1 Grundlagen von Reflection

In diesem ersten Abschnitt über Reflection berücksichtigen wir die Tatsache, dass die durch Reflection beschriebenen Klassen, Schnittstellen und Methoden Generics verwenden können, zunächst noch nicht. Wie Reflection für Generics erweitert wurde, erfahren Sie dann in Abschnitt 2.2.

2.1.1 Die Klasse Class

Die zentrale Klasse von Reflection ist Class. Ein Objekt der Klasse Class beschreibt eine Klasse. Es gibt eine ganze Reihe von Möglichkeiten, wie man an ein Class-Objekt gelangt. Wenn man schon ein Objekt der Klasse hat, dann kann man sich mit getClass einfach das dazugehörige Class-Objekt geben lassen (X sei irgendeine Klasse):

```
X x = new X();
Class<?> c = x.getClass();
```

Wie oben zu sehen ist, ist die Klasse Class eine generische Klasse. Der Typparameter ist die Klasse, die vom Class-Objekt beschrieben wird (in obigem Beispiel also Class<X>). Da getClass eine Methode der Klasse Object ist, muss der Rückgabetyp für alle Objekte aller möglichen Klassen passen; er ist deshalb Class<?>. Es wäre zwar oben möglich, auf Class<X> zu casten. Dies hat aber eine (unterdrückbare) Warnung zur Folge, da eine vollständige Typüberprüfung zur Laufzeit nicht möglich ist, wie im vorherigen Kapitel schon erläutert wurde. Weil man aber weiß, dass die Variable x vom Typ X ist, muss das Objekt, das durch x referenziert wird, vom Typ X oder einer aus X abgeleiteten Klasse sein. Der Typparameter kann deshalb zumindest auf X eingeschränkt werden:

```
Class<? extends X> c = x.getClass();
```

Eine weitere Möglichkeit, an ein Class-Objekt zu gelangen, ist in folgendem Beispiel zu sehen:

```
Class<X> c = X.class;
```

In diesem Fall kann das Class-Objekt einer Variablen des Typs Class<X> ohne Warnungen zugewiesen werden, da der Compiler überprüfen kann, dass es sich in diesem Fall tatsächlich um ein Class<X>-Objekt handeln muss. (Bitte beachten Sie, dass dies im ersten Fall bei der Verwendung von getClass im Allgemeinen nicht geht, da bei der Zuweisung an die Variable x nicht immer die Anwendung des Operators new im Spiel sein muss, sondern es könnte z. B. auch sein, dass der Rückgabewert irgendeiner Methode an x zugewiesen wird. Diese Methode könnte auch ein Objekt einer von X abgeleiteten Klasse zurückgeben.) Bei der Verwendung von .class muss sich die Programmiererin allerdings bereits beim Schreiben ihres Programms auf die Klasse X festlegen; dies kann zur Laufzeit nicht mehr geändert werden (X ist statisch in den Programmcode „eingebrannt“). Wenn man sich das Class-Objekt von der statischen Methode forName der Klasse Class geben lässt, muss die Festlegung auf eine bestimmte Klasse nicht zur Programmierzeit erfolgen. Als Parameter benötigt die Methode forName den vollständigen Klassennamen (d. h. einschließlich des Package-

Namens) in Form eines Strings. Diesen String kann sich das Programm dynamisch beschaffen (z. B. aus einer Eingabe des Benutzers oder einer Konfigurationsdatei); zur Zeit des Programmierens muss die Programmiererin die Klasse nicht kennen:

```
try
{
    Class<?> c = Class.forName("javax.swing.JButton");
}
catch(ClassNotFoundException e)
{
    ...
}
```

Wie durch den Try-Catch-Block angedeutet, kann die Methode forName eine ClassNotFoundException werfen, falls sich die als String angegebene Klasse bzw. Schnittstelle nicht im Classpath des ausführenden Prozesses befindet.

Will man auf das Class-Objekt für ein Feld zugreifen, kann man sowohl die Variante mit .class als auch Class.forName einsetzen. Die folgenden beiden Anweisungen, in denen ein Zugriff auf ein Class-Objekt für ein zweidimensionales String-Feld beschafft wird, sind äquivalent, wobei in dem String-Argument der Methode Class.forName die Notation verwendet wird, die Java intern zur Beschreibung des Typs von Feldern nutzt:

```
Class<?> c1 = String[][].class;
Class<?> c2 = Class.forName("[[Ljava.lang.String;");
```

Übrigens erhält man auf beide Arten nicht nur das gleiche, sondern sogar dasselbe Class-Objekt (d. h. nach Ausführung der beiden Zeilen gilt c1 == c2 und nicht nur c1.equals(c2)).

Auch für Schnittstellen existieren Class-Objekte. Diese lassen sich über .class oder Class.forName besorgen. Die Variante über ein Objekt und getClass funktioniert nicht, weil hier immer die Klasse des Objekts zurückgegeben wird (selbst wenn die Variable, auf die getClass angewendet wird, den Typ der Schnittstelle hat).

Und auch für die primitiven Datentypen wie boolean, byte und int gibt es entsprechende Class-Objekte. Der Zugriff darauf lässt sich aber auch in diesem Fall nicht über getClass bewerkstelligen:

```
boolean b = true;
Class<?> c = b.getClass(); //Syntaxfehler!!!!
```

Es funktioniert aber die Variante mit .class:

```
Class<?> c = boolean.class;
```

Für die Beschaffung der Class-Objekte von Feldern primitiver Datentypen gibt es mehrere Varianten. Sowohl getClass als auch .class sowie Class.forName funktionieren. Die folgenden Zeilen zeigen ein Beispiel für die Beschaffung des Class-Objekts eines dreidimensionalen Felds des Typs boolean:

```
boolean[][][] boolArray = new boolean[10][20][30];
Class<?> c1 = boolArray.getClass();
Class<?> c2 = boolean[][][].class;
Class<?> c3 = Class.forName("[[[Z");
```

Auch hier gilt: c1 == c2 und c2 == c3 (und damit auch c1 == c3). Beachten Sie bitte, dass B schon für den primitiven Datentyp byte vergeben ist, so dass für boolean Z verwendet wird.

Hat man ein Class-Objekt, kann man z. B. unterschiedliche Eigenschaften der damit beschriebenen Klasse abfragen, z. B. ob die Klasse public ist, ob die Klasse final ist und ob es sich überhaupt um eine Klasse oder um eine Schnittstelle handelt. Auch kann man sich von einem Class-Objekt die Basisklasse sowie die implementierten Schnittstellen geben lassen. Sowohl die Basisklasse als auch die implementierten Schnittstellen werden wiederum als Class-Objekte zurückgeliefert. Weiterhin können die Attribute, Konstruktoren und Methoden einer Klasse erfragt werden.

2.1.2 Die Klasse Field

Mit Hilfe der Methoden getField, getDeclaredField, getFields und getDeclaredFields der Klasse Class kann man Informationen über die Attribute einer Klasse abfragen. Ein Attribut ist durch ein Objekt der Klasse Field repräsentiert, die ebenfalls eine der zu Reflection gehörenden Klassen ist. Wie die Methodennamen vermuten lassen, haben die Methoden getField und getDeclaredField den Rückgabetyp Field, weil sie genau ein Field-Objekt zurückgeben, während getFields und getDeclaredFields mehrere Field-Objekte in Form eines Field-Feldes (Field[]) zurückgeben. Bei den Methoden getField und getDeclaredField muss man den Namen des Attributs bereits kennen; dieser wird als String-Argument angegeben. Ein Field-Objekt wird zurückgeliefert, falls der Name des Attributs gültig war (andernfalls wird eine Ausnahme geworfen). Die Methoden getFields und getDeclaredFields sind dagegen parameterlos. Die Methoden, die „Declared“ in ihrem Namen haben, beziehen sich nur auf solche Attribute, die genau in der betrachteten Klasse definiert wurden (geerbte Attribute werden also nicht berücksichtigt), unabhängig von deren Sichtbarkeit (also auch einschließlich nicht-öffentlicher Attribute). Im Gegensatz dazu beziehen sich die Methoden, in deren Namen „Declared“ nicht vorkommt, auf alle, also auch auf die geerbten Attribute, allerdings nur auf die öffentlichen.

So wie ein Class-Objekt eine Klasse repräsentiert, beschreibt ein Field-Objekt ein Attribut einer Klasse. Dies sind einige wichtige Methoden der Klasse Field:

```
public Class<?> getType();
public Object get(Object obj) throws IllegalArgumentException,
                                     IllegalAccessException;
public void set(Object obj, Object value)
                               throws IllegalArgumentException,
                                     IllegalAccessException;
public void setAccessible(boolean flag) throws SecurityException;
public int getModifiers();
```

Über die Methode getType der Klasse Field kann man den Typ des Attributs erfragen (zur Erinnerung: es gibt auch für primitive Datentypen Class-Objekte). Hat man ein Objekt der Klasse, die das Attribut besitzt, das durch ein Field-Objekt repräsentiert wird, kann man mit Hilfe der Methoden get und set den Attributwert lesen und ändern. Dazu muss natürlich das Objekt, dessen Attributwert gelesen bzw. geändert werden soll, als Parameter angegeben werden; die Methoden get und set werden ja auf das Field-Objekt angewendet und nicht auf das Objekt, dessen Attribut gelesen oder geändert werden soll. Beim Zugriff auf ein

Attribut wird die Sichtbarkeit des Attributs standardmäßig respektiert. Das heißt, wenn auf ein privates Attribut von außerhalb der betreffenden Klasse zugegriffen wird, werfen die Methoden get und set die Ausnahme IllegalAccessException. Entsprechendes gilt für Attribute mit Standard- bzw. Protected-Sichtbarkeit. Wenn man will, kann man diesen Schutzmechanismus aber einfach abschalten: Wenn man setAccessible mit dem Parameter true auf das Field-Objekt vor dem Zugriff anwendet, sind alle Zugriffe auf das Attribut möglich, ganz gleich, welche Sichtbarkeit das Attribut besitzt.

Über die Methode getModifiers kann neben der Sichtbarkeit abgefragt werden, welche Java-Schlüsselwörter noch bei der Deklaration des Attributs verwendet wurden (z. B. transient).

2.1.3 Die Klasse Method

Ganz analog zu den Methoden für Attribute besitzt die Klasse Class die Methoden getMethod, getDeclaredMethod, getMethods und getDeclaredMethods für Methoden. Ähnlich wie bei den Attributen muss bei den Singularvarianten der Name einer Methode angeben werden und der Rückgabetyp ist Method, während die Pluralvarianten parameterlos sind und ein Method-Feld (Method[]) zurückliefern. Wegen der Konzepts des Überladens ist ein Name für eine Methode nicht eindeutig. Deshalb müssen bei den Singularvarianten neben dem Methodennamen auch die Typen der Parameter (in Form von Class<?>-Objekten) angegeben werden. Auch der Unterschied zwischen den Methoden mit und ohne „Declared“ im Namen ist derselbe wie bei den Methoden für die Attribute. So wie ein Objekt der Klasse Field für ein Attribut einer Klasse steht, repräsentiert ein Method-Objekt eine Methode. Die Klasse Method besitzt u. a. die folgenden Methoden:

```
public Class<?>[] getParameterTypes();
public Class<?> getReturnType();
public Class<?>[] getExceptionTypes();
public Object invoke(Object obj, Object... args)
          throws IllegalAccessException, IllegalArgumentException,
                 InvocationTargetException;
public void setAccessible(boolean flag) throws SecurityException;
public int getModifiers();
```

Wie aufgrund der Methodennamen erahnt werden kann, kann man mit getParameterTypes die Typen der Parameter, mit getReturnType den Rückgabetyp und mit getExceptionTypes die möglicherweise geworfenen Ausnahmen der Methode abfragen (die Parametertypen, der Rückgabetyp und die Ausnahmen werden alle durch Class-Objekte repräsentiert, wobei der Rückgabetyp void vom Class-Objekt void.class repräsentiert wird). Mit invoke kann die Methode, für die das Method-Objekt steht, aufgerufen werden. Da invoke auf das Method-Objekt angewendet wird und nicht auf das Zielobjekt, auf das die eigentliche Methode angewendet werden soll, muss das Zielobjekt als Parameter angegeben werden (auch dies ist ganz analog zu den Get- und Set-Methoden der Klasse Field). Weitere Parameter von invoke sind die aktuellen Parameter des Methodenaufrufs, der durch Method beschrieben wird. Diese Parameter müssen kompatibel sein zu den durch getParameterTypes beschriebenen Typen. Wegen der Allgemeinheit können es beliebig viele Parameter des Typs Object sein. Die variable Anzahl wird durch das Sprachkonzept Varargs von Java (notiert durch drei Punkte) realisiert. Falls ein Parametertyp ein primitiver Datentyp ist, muss das Argument

als ein Objekt der entsprechenden Wrapper-Klasse (für int z. B. ein Integer-Objekt) übergeben werden. Der Rückgabetyp ist im Allgemeinen ebenfalls Object. Falls das Method-Objekt zu einer Void-Methode gehört, ist der Rückgabewert immer null.

Bezüglich der Sichtbarkeitsüberprüfung und setAccessible gilt dasselbe wie bei der Klasse Field. Auch getModifiers entspricht der Methode desselben Namens der Klasse Field.

2.1.4 Die Klasse Constructor

Ganz analog zu den Attributen und Methoden kann man von einem Class-Objekt die Konstruktoren über die Methoden getConstructor, getDeclaredConstructor, getConstructors und getDeclaredConstructors erfragen. Rückgabetyp ist Constructor bzw. Constructor[]. Einige Besonderheiten der Konstruktoren spiegeln sich in der Reflection-Schnittstelle wider: Da alle Konstruktoren denselben Namen haben, muss bei den Singularvarianten getConstructor und getDeclaredConstructor kein Name, sondern es müssen nur die Typen der Parameter angegeben werden. Aufgrund der Tatsache, dass Konstruktoren nicht vererbt werden, ist der Unterschied zwischen den Methoden mit und ohne „Declared“ im Namen geringer als für Attribute und Methoden; es geht immer nur um Konstruktoren dieser Klasse: Die Methoden getDeclaredConstructor und getDeclaredConstructors beziehen sich auf alle Konstruktoren der Klasse, die Methoden getConstructor und getConstructors nur auf die öffentlichen Konstruktoren der Klasse.

Die Klasse Konstruktor ist eine generische Klasse. Der Typparameter entspricht der Klasse, um deren Konstruktor es geht. Wichtige Methoden der Klasse Constructor<T> sind:

```
public Class<?>[] getParameterTypes();
public Class<?>[] getExceptionTypes();
public T newInstance(Object... initArgs)
         throws InstantiationException, IllegalAccessException,
                IllegalArgumentException, InvocationTargetException;
public void setAccessible(boolean flag) throws SecurityException;
public int getModifiers();
```

Die Methode newInstance entspricht der Methode invoke der Klasse Method. Ein Unterschied besteht darin, dass es kein Objekt gibt, auf das der Konstruktor angewendet wird. Folglich sind die Parameter von newInstance nur die entsprechenden Konstruktorparameter. Ferner ist der Rückgabetyp nicht der allgemeine Typ Object, sondern T, der Typparameter der Klasse Constructor. Alle anderen Methoden sollten aufgrund der Erläuterungen zur Klasse Method selbsterklärend sein.

Falls eine Klasse einen parameterlosen Konstruktor besitzt und man ein Objekt mit Hilfe dieses Konstruktors erzeugen möchte, muss man übrigens die Klasse Constructor nicht notwendigerweise verwenden. In diesem Fall kann man auch die parameterlose Methode newInstance der Klasse Class benutzen. Auch diese Methode hat als Rückgabetyp den Typparameter T.

2.1.5 Beispiel

Mit Hilfe der beschriebenen Klassen und Methoden kann man ein Programm schreiben, dem als Kommandozeilenargumente beliebig viele vollständige Klassennamen (inklusive Package-Namen) übergeben werden und das dann u. a. die Basisklasse, die implementierten Schnittstellen, die Attribute, die Konstruktoren und die Methoden (für die letzten drei jeweils mit und ohne Declared) ausgibt. Um nicht zu viel Platz zu verbrauchen, wird in Listing 2.1 nur ein Ausschnitt des Programms gezeigt; das vollständige Programm ist von der angegebenen Web-Seite erhältlich.

Listing 2.1 Programm zur Ausgabe der Eigenschaften einer Klasse (ohne Generics)

```
package javacomp.basics;

import java.lang.reflect.*;

public class Reflector
{
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("arguments: list of class names");
        }
        for(String className: args)
        {
            analyze(className);
            System.out.println("================================");
        }
    }

    private static void analyze(String className)
    {
        try
        {
            String headline = className + ":";
            System.out.println(headline);
            printUnderlined(headline.length());

            Class<?> c = Class.forName(className);

            ...

            System.out.println("  declared methods:");
            Method[] declaredMethods = c.getDeclaredMethods();
            handleMethods(declaredMethods);

            System.out.println("  methods:");
            Method[] methods = c.getMethods();
            handleMethods(methods);
        }
        catch(ClassNotFoundException e)
        {
            System.out.println(className + ": not found");
        }
    }
```

```
    private static void handleMethods(Method[] methods)
    {
        for(Method method: methods)
        {
            System.out.println("    " + method.getName() + ":");
            int mod = method.getModifiers();
            System.out.println("      modifiers: " +
                                       Modifier.toString(mod));
            System.out.print("      parameters:");
            Class<?>[] params = method.getParameterTypes();
            for(Class<?> param : params)
            {
                System.out.print(" " + param.getName());
            }
            System.out.println();
            System.out.println("      return type: " +
                               method.getReturnType().getName());
            System.out.print("      exceptions:");
            Class<?>[] exceptions = method.getExceptionTypes();
            for(Class<?> exception : exceptions)
            {
                System.out.print(" " + exception.getName());
            }
            System.out.println();
        }
    }
}
```

In Listing 2.1 werden beispielhaft die Teile des Programmcodes gezeigt, die Informationen über die Methoden einer Klasse ausgeben. Die Methode handleMethods wird pro Klasse zwei Mal aufgerufen, einmal für das von getDeclaredMethods und einmal für das von getMethods zurückgelieferte Method-Feld. Die Methode handleMethods gibt für jedes Element eines als Argument übergebenen Method-Felds den Namen der Methode, ihre Modifiers, die Typen ihrer Parameter, den Rückgabetyp und die Typen der möglicherweise der von ihr geworfenen Ausnahmen aus. Zu den sogenannten Modifiers gehören u. a. public, private, abstract, final usw. Die Modifiers sind durch eine ganze Zahl des Typs int codiert, wobei unterschiedliche Bits für unterschiedliche Eigenschaften stehen. Ist das betreffende Bit gesetzt, so ist die Eigenschaft vorhanden. Die statische Methode toString der Klasse Modifier liefert alle vorhandenen Eigenschaften als String zurück, wobei die einzelnen Eigenschaften durch ein Leerzeichen voneinander getrennt sind. Es ist wesentlich bequemer, diese Methode zu benutzen als für jedes einzelne Bit abzuprüfen, ob es gesetzt ist oder nicht, um dann im positiven Fall einen entsprechenden String auszugeben.

2.1.6 Anwendungen

Auch wenn es den Leserinnen und Lesern dieses Buches nicht immer bewusst ist, so haben sie vermutlich schon Software verwendet, in der Reflection eingesetzt wird. Wenn Sie nämlich eine Entwicklungsumgebung (IDE: Integrated Development Environment) wie Eclipse verwenden, dann werden Sie wissen, dass die Entwicklungsumgebung beim Tippen Vorschläge macht, auf welche Methoden oder Attribute über eine bestimmte Variable zugegrif-

fen werden kann. Dies funktioniert auch für Klassen, die nicht im Quellcode vorliegen, sondern nur als übersetzter Code zum Beispiel in einer Jar-Datei. Über Reflection kann die Entwicklungsumgebung mühelos herausfinden, welche Methoden und Attribute eine Klasse besitzt und diese dann zur Auswahl anbieten.

Noch intensiver wird Reflection in BlueJ verwendet. In BlueJ kann man interaktiv eine beliebige Klasse auswählen und sich ein Objekt davon erzeugen lassen; BlueJ bietet den Anwendern dazu eine Liste möglicher Konstruktoren und fragt die Parameterwerte ab, nachdem man sich für einen Konstruktor entschieden hat. Für alle so erzeugten Objekte kann man sich die aktuellen Werte der Attribute anzeigen lassen (ohne dass es dafür Getter-Methoden geben muss). Man kann auf ein zuvor erzeugtes Objekt auch jede mögliche Methode anwenden, wobei dazu wie im Falle des Konstruktors die Parameterwerte interaktiv erfragt werden.

Ein weiteres Beispiel für den Einsatz von Reflection ist RMI (Remote Method Invocation). Wenn Sie RMI kennen, dann werden Sie wissen, dass man damit Methoden auf Objekte, die sich in einem anderen Prozess und eventuell sogar auf einem anderen Rechner befinden, anwenden kann. Die Java-Laufzeitumgebung stellt dafür u. a. ein sogenanntes RMI-Skeleton bereit. Das ist Software, die über eine Netzverbindung Daten empfängt, die angeben, welche Methode auf welchem Objekt mit welchen Argumenten aufgerufen werden soll. Das RMI-Skeleton führt den entsprechenden Aufruf dann auch aus. Da die Skeleton-Software unabhängig von einer konkreten RMI-Anwendung programmiert wurde, kann ein solcher Methodenaufruf nur über Reflection (das heißt über die Methode invoke der Klasse Method) realisiert werden.

Zur Abrundung dieser Aufzählung über Anwendungen, die Reflection einsetzen, wollen wir noch einmal auf die Problematik zurückkommen, die wir im vorhergehenden Kapitel über Generics besprochen haben, dass nämlich der Ausdruck `new T()` für einen Typparameter T einen Compilerfehler verursacht. Mit Reflection erhält man als Software-Entwickler jedoch die Möglichkeit, Programmcode zu schreiben, der ein Objekt, dessen Typ zur Programmierzeit noch nicht bekannt ist, erzeugt. Im Folgenden gehen wir der Einfachheit halber davon aus, dass jeder konkrete Typ, der für T eingesetzt wird, einen parameterlosen Konstruktor besitzen wird. Dann kann man z. B. folgende generische Methode zum Erzeugen eines Objekts programmieren:

```
class X
{
    public static <T> T newInstance(Class<T> c) throws Exception
    {
        return c.newInstance();
    }
}
```

Die Methode kann dann zum Beispiel zum Ausprobieren so aufgerufen werden:

```
Class<?> c = Class.forName("java.lang.String");
String s = (String) X.newInstance(c);

c = Class.forName("java.lang.StringBuffer");
StringBuffer s2 = (StringBuffer) X.newInstance(c);
```

Der Code verursacht weder Compiler- noch Laufzeitfehler, also insbesondere auch keine ClassCastExceptions. Das heißt, dass tatsächlich Objekte des Typs String bzw. StringBuffer erzeugt werden. Würde man als String allerdings „java.lang.Integer“ angeben, würde eine InstantiationException geworfen, da Integer keinen parameterlosen Konstruktor hat.

In diesem Zusammenhang soll auch noch kurz auf die Problematik eingegangen werden, dass auch kein Feld für einen Typparameter T erzeugt werden kann und der Ausdruck `new T[length]` ebenfalls als syntaktisch falsch vom Compiler gekennzeichnet wird. Mit der statischen Methode newInstance der Klasse Array, die ebenfalls zu den Reflection-Klassen gehört, lässt sich aber trotzdem ein Feld erzeugen, dessen Typ zur Zeit des Programmierens noch nicht bekannt sein muss:

```
Class<?> c = Class.forName("java.lang.String");
String[] sArray = (String[]) Array.newInstance(c, 10);
System.out.println(sArray.getClass().getName());
c = Class.forName("java.lang.Integer");
Integer[] iArray = (Integer[]) Array.newInstance(c, 20);
System.out.println(iArray.getClass().getName());
```

Die Ausgabe dazu lautet:

```
[Ljava.lang.String;
[Ljava.lang.Integer;
```

Es wurden also tatsächlich Objekte des Typs String[] und Integer[] erzeugt.

2.2 Reflection mit Generics

Zwar ist die Klasse Class selbst eine generische Klasse, wie wir im vorigen Abschnitt gesehen haben. Dennoch wurde in der Beschreibung bisher außer Acht gelassen, dass die Klassen, Schnittstellen, Methoden und Konstruktoren, die durch die Reflection-Klassen Class, Method und Constructor beschrieben werden, selbst generisch sein können und dass dadurch Attribut-, Parameter- und Rückgabetypen generische Typen oder Typvariablen sein können. Derartige Sachverhalte können mit den bisher besprochenen Klassen und Methoden nicht herausgefunden werden. Um dies zu erläutern, betrachten wir die folgenden Beispielklassen:

Listing 2.2 Beispielklassen mit Generics

```
package javacomp.basics;

class A<T1, T2> {}
class B<T1, T2> extends A<T1, C>
{
    private A<T2,D> a;
    public A<T2,D> m(B<C,D> b)
    {
        return a;
    }
}
```

```
class C {}
class D {}
```

Wenn man das Programm aus Listing 2.1 auf die Klasse B anwendet, dann wird für die Methode m ausgegeben, dass diese Methode einen Parameter des Typs B und einen Rückgabewert des Typs A besitzt. Wenn man sich jetzt daran erinnert, dass es die Typlöschung (Type Erasure) gibt, könnte man meinen, dass Informationen über Generics-Eigenschaften zur Laufzeit grundsätzlich nicht vorhanden sind. Dies stimmt aber nicht. Die Typlöschung bezieht sich auf den vom Compiler erzeugten Byte-Code. Im Gegensatz dazu kann man über Reflection durchaus Generics-Informationen herausfinden. In Reflection wurden zu diesem Zweck neue Klassen und Schnittstellen eingeführt. Außerdem wurden die vorhandenen Klassen wie Class, Field, Constructor und Method um entsprechende Methoden erweitert. Dies macht Reflection deutlich komplizierter.

2.2.1 Reflection-Typsystem

In der Zeit vor Generics war ein Typ ein primitiver Typ, eine Klasse oder eine Schnittstelle. Wie Sie in Abschnitt 2.1 gesehen haben, können alle diese Typen durch ein Class-Objekt repräsentiert werden. Mit Generics erweitert sich der Begriff Typ. Betrachten wir dazu die Klasse X:

```
class X<T>
{
    public void m(T t) ...
}
```

Der Parametertyp der Methode m ist im obigen Beispiel der Typparameter T der Klasse X. Dieser Typ T kann nicht durch ein Class-Objekt repräsentiert werden. Der Begriff Typ wurde deshalb verallgemeinert. Zu diesem Zweck wurde die Schnittstelle Type in Reflection eingeführt. Type ist eine leere Schnittstelle (sie besitzt keine Methoden). Sie dient lediglich dazu, unterschiedliche Arten von Typen zusammenzufassen. Folgende Arten von Typen gibt es:

- Class: Was mit einem Class-Objekt beschrieben werden kann, ist selbstverständlich eine Variante eines Typs. Es ist deshalb wenig überraschend, dass die Klasse Class die Schnittstelle Type implementiert.
- TypeVariable: Allerdings gibt es auch die Schnittstelle TypeVariable, die Type erweitert und mit der Typparameter (wie T im obigen Beispiel) repräsentiert werden. Da Typvariable durch extends eingeschränkt sein können, besitzt die Schnittstelle TypeVariable eine Methode namens getBounds, mit der die beschränkenden Typen herausgefunden werden können (Rückgabetyp Type[]).
- ParameterizedType: Für generische Typen wie z. B. ArrayList<String> gibt es die Schnittstelle ParameterizedType, die ebenfalls Type erweitert. Die Typargumente (im Allgemeinen können es mehrere sein), die durch die Methode getActualTypeArguments erfragt werden können, können nicht in allen Fällen durch Class-Objekte beschrieben werden, denn z. B. ist bei ArrayList<ArrayList<String>> das Typargument wieder ein ParameterizedType. Aber auch ArrayList<T> für einen Typparameter T ist möglich. Im Allgemeinen sind die Typargumente somit wieder Types; die Methode getActualTypeArguments liefert

folglich ein Type-Feld (Type[]) zurück. Die „eigentliche“ Klasse eines ParameterizedType bekommt man durch Aufruf der Methode getRawType (Rückgabetyp Type).

- WildcardType: Bei ArrayList<?> ist das Typargument ein Wildcard, das durch die bisherigen Typarten nicht erfasst wird. Aus diesem Grund gibt es die Schnittstelle WildcardType, die ebenfalls aus Type abgeleitet ist. Um zu unterscheiden, ob es sich um „? extends …“ oder um „? super …“ handelt, besitzt die Schnittstelle die Methoden getLowerBounds und getUpperBounds. Damit können die Typgrenzen des Wildcard-Typs abgefragt werden. Der Rückgabetyp beider Methoden ist Type[]. Bei „? extends X“ ist X die obere Grenze, während die untere Grenze nicht existiert (getLowerBounds liefert ein Feld der Länge 0 zurück). Bei „? super X“ ist X die untere Grenze, während Object logischerweise die obere Grenze darstellt.
- GenericArrayType: Wie im Kapitel über Generics beschrieben wurde, gibt es gewisse Besonderheiten bei Feldern generischer Datentypen. Diesen Besonderheiten ist die Existenz der Schnittstelle GenericArrayType (ebenfalls aus Type abgeleitet) geschuldet. Wir wollen in diesem Buch auf diese Typart nicht näher eingehen.

Zum Einstieg in dieses Typsystem beginnt man in aller Regel auch mit einem Class-Objekt, das man so erhalten kann, wie dies in Abschnitt 2.1 bereits beschrieben wurde. Es wurde ebenfalls schon erwähnt, dass es zu einer Klasse nur ein einziges Class-Objekt gibt. Wenn man also in seinem Programm beispielsweise Variablen hat, die Objekte referenzieren, die als ArrayList<String> und ArrayList<Integer> angelegt wurden, so liefert eine Anwendung der Methode getClass auf die beiden Objekte dasselbe Class-Objekt für ArrayList zurück. Ein Ausdruck der Form ArrayList<String>.class ist konsequenterweise syntaktisch gar nicht möglich. Wie bei instanceof benötigt man hier immer den rohen Typ (also ArrayList.class in diesem Fall).

Hat man ein Class-Objekt, so kann man darauf die Methode getTypeParameters anwenden. Diese Methode gibt ein Feld von TypeVariables (TypeVariable[]) zurück. Hat dieses Feld die Länge 0, handelt es sich um eine „normale“ (d. h. nicht-generische) Klasse. Da nicht nur Klassen (und Schnittstellen), sondern auch Konstruktoren und Methoden Typparameter haben können, besitzen die Klassen Constructor und Method ebenfalls diese Methode. Um diesen Sachverhalt deutlich herauszustellen, wurde die Schnittstelle GenericDeclaration definiert, welche genau diese Methode enthält. Die Klassen Class, Constructor und Method implementieren alle diese Schnittstelle und haben somit die Methode getTypeParameters.

Die beschriebenen Sachverhalte sind in Bild 2.1 noch einmal zusammengefasst (ohne GenericArrayType).

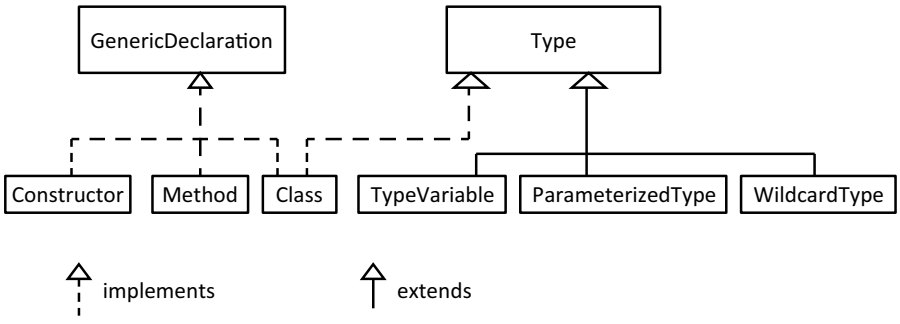

Bild 2.1 UML-Diagramm zum Reflection-Typsystem (ohne GenericArrayType)

Die Pfeile in Bild 2.1 bedeuten zum Beispiel, dass die Klasse Class die Schnittstellen GenericDeclaration und Type implementiert, während die Schnittstelle TypeVariable aus der Schnittstelle Type abgeleitet ist.

In Bild 2.2 ist zusätzlich dargestellt, welchen Rückgabetyp die im Text erwähnten Methoden besitzen (wieder ohne GenericArrayType). Die Pfeile sind mit den Methodennamen beschriftet. Ein Pfeil geht von der Klasse oder Schnittstelle weg, in der sich die Methode befindet. Sie endet an der Schnittstelle, die den Rückgabetyp der betreffenden Methode darstellt. Wenn der Rückgabetyp ein Feld der entsprechenden Schnittstelle ist, so ist ein Stern in der Nähe der Pfeilspitze zu finden. Man sieht also beispielsweise, dass die Schnittstelle TypeVariable die Methode getBounds mit dem Rückgabetyp Type[] besitzt.

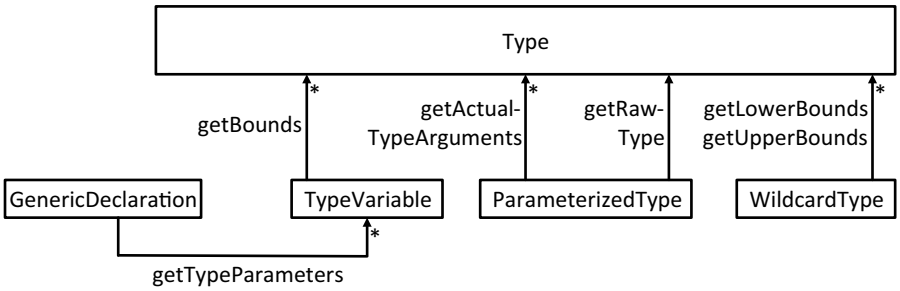

Bild 2.2 Einige Methoden des Reflection-Typsystems und deren Rückgabetyp

2.2.2 Zusätzliche Methoden in Reflection-Klassen

Die Klasse Class besitzt zusätzlich zu den „alten“ Methoden getSuperclass und getInterfaces (Rückgabetyp Class<?> bzw. Class<?>[]), die es aus Kompatibilitätsgründen weiterhin gibt, nun auch die Methoden getGenericSuperclass und getGenericInterfaces. Entsprechendes gilt für die Klassen Field, Constructor und Method. Field hat außer getType jetzt auch noch getGenericType. Constructor und Method haben neben getParameterTypes und getExceptionTypes jetzt auch noch getGenericParameterTypes und getGenericExceptionTypes. Method hat zusätzlich noch zu getReturnType die Methode getGenericReturnType (der Rückgabetyp aller erwähnten Methoden ist Type bzw. Type[]). Dass die beiden Klassen Constructor und Method die Methode getTypeParameters (Rückgabetyp TypeVariable[]) besitzen, weil sie die Schnittstelle GenericDeclaration implementieren, wurde zuvor schon erwähnt.

2.2.3 Beispiel

Das Programm aus Listing 2.1 kann nun so abgeändert werden, dass bei der Untersuchung einer Klasse immer die generischen Varianten der Reflection-Methoden verwendet werden (also z. B. getGenericParameterTypes statt getParameterTypes). Diese Änderung ist relativ einfach vorzunehmen. Da der Unterschied zwischen den Methoden mit und ohne „Declared“ im Namen schon durch die Ausgaben des Programms von Listing 2.1 deutlich geworden sein sollte, werden im folgenden Programm in Listing 2.3 nur noch die Declared-Varianten

benutzt. Durch diese Änderungen ist das Programm nicht aufwändiger geworden. Was das Programm nun aber doch aufwändiger macht, ist die Darstellung der Informationen eines Type-Objekts. In Listing 2.3 wird zu diesem Zweck die Methode printType verwendet. Diese Methode stellt nicht nur dar, um welche Typart (WildcardType, TypeVariable usw.) es sich handelt, sondern z. B. für einen WildcardType auch die unteren und oberen Grenzen. Da diese Grenzen wieder durch Type-Objekte beschrieben sind, wird zur Ausgabe dieser Type-Objekte wieder die Methode printType verwendet. Damit die Darstellung auf dem Bildschirm einigermaßen übersichtlich aussieht, wird mit Einrückungen gearbeitet. Der Methode printType wird deshalb als Argument nicht nur das Type-Objekt übergeben, über das Informationen ausgegeben werden sollen, sondern auch eine ganze Zahl, welche angibt, um wie viele Leerzeichen die ausgegebenen Informationen eingerückt sein sollen. Die Methode printType ist relativ aufwändig, da sie für die unterschiedlichen Arten von Typen eine Fallunterscheidung durchführt und dann jede einzelne Form eines Typs näher behandelt. In Listing 2.3 geben wir wieder nur einen Ausschnitt an; der vollständige Programmcode kann von der Web-Seite zum Buch heruntergeladen werden.

Listing 2.3 Programm zur Ausgabe der Eigenschaften einer Klasse (mit Generics)

```
package javacomp.basics;

import java.lang.reflect.*;

public class ReflectorForGenerics
{
    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("arguments: list of class names");
        }
        for(String className: args)
        {
            analyze(className);
            System.out.println("============================");
        }
    }

    private static void analyze(String className)
    {
        try
        {
            String headline = className + ":";
            System.out.println(headline);
            printNTimes('-', headline.length());
            System.out.println();

            Class<?> c = Class.forName(className);

            System.out.println("  type parameters: ");
            Type typeParams[] = c.getTypeParameters();
            for(Type typeParam : typeParams)
            {
                printType(typeParam, 4);
            }
```

```
            System.out.println("  generic base class: ");
            Type genericBase = c.getGenericSuperclass();
            if(genericBase != null)
            {
                printType(genericBase, 4);
            }

            ...

            System.out.println("  declared methods:");
            Method[] declaredMethods = c.getDeclaredMethods();
            handleMethods(declaredMethods);
        }
        catch(ClassNotFoundException e)
        {
            System.out.println(className + ": not found");
        }
    }

    private static void printType(Type t, int indentation)
    {
        printNTimes(' ', indentation);
        if(t instanceof Class<?>)
        {
            Class<?> cls = (Class<?>) t;
            if(cls.isInterface())
            {
                System.out.print("interface: ");
            }
            else
            {
                System.out.print("class: ");
            }
            System.out.println(cls.getName());
        }
        else if(t instanceof TypeVariable<?>)
        {
            TypeVariable<?> typeVar = (TypeVariable<?>) t;
            System.out.println("type variable: " +
                               typeVar.getName());
            printNTimes(' ', indentation+2);
            System.out.println("bounds:");
            Type[] bounds = typeVar.getBounds();
            if(bounds.length == 1)
            {
                printType(bounds[0], indentation+4);
            }
            else
            {
                int i = 0;
                for(Type bound : bounds)
                {
                    i++;
                    printNTimes(' ', indentation+4);
                    System.out.println(i + ". bound:");
                    printType(bound, indentation+6);
                }
```

```
            }
        }
        else if(t instanceof ParameterizedType)
        {
            ...
        }
        else if(t instanceof WildcardType)
        {
            ...
        }
        else if(t instanceof GenericArrayType)
        {
            ...
        }
        else
        {
            System.out.println("<<<unknown type!!!!!>>>");
        }
    }

    ...

    private static void handleMethods(Method[] methods)
    {
        for(Method method: methods)
        {
            System.out.println("    " + method.getName() + ":");
            int mod = method.getModifiers();
            System.out.println("      modifiers: " +
                                       Modifier.toString(mod));
            System.out.println("      generic parameters: ");
            Type[] params = method.getGenericParameterTypes();
            int i = 0;
            for(Type param : params)
            {
                i++;
                printNTimes(' ', 8);
                System.out.println(i + ". parameter:");
                printType(param, 10);
            }
            System.out.println("      generic return type: ");
            Type returnType = method.getGenericReturnType();
            printType(returnType, 8);
            System.out.println("      generic exceptions: ");
            Type[] exceptions = method.getGenericExceptionTypes();
            i = 0;
            for(Type exception : exceptions)
            {
                i++;
                printNTimes(' ', 8);
                System.out.println(i + ". exception:");
                printType(exception, 10);
            }
        }
    }
}
```

Führen wir dieses Programm mit dem Kommandozeilenargument javacomp.basics.B (siehe Listing 2.2) aus, so erhalten wir diese Ausgabe (gekürzt):

```
javacomp.basics.B:
------------------
  type parameters:
    type variable: T1
      bounds:
        class: java.lang.Object
    type variable: T2
      bounds:
        class: java.lang.Object
  generic base class:
    parameterized type:
      raw type:
        class: javacomp.basics.A
      actual type arguments:
        1. argument:
          type variable: T1
            bounds:
              class: java.lang.Object
        2. argument:
          class: javacomp.basics.C
  ...
  declared methods:
    m:
      modifiers: public
      generic parameters:
        1. parameter:
          parameterized type:
            raw type:
              class: javacomp.basics.B
            actual type arguments:
              1. argument:
                class: javacomp.basics.C
              2. argument:
                class: javacomp.basics.D
      generic return type:
        parameterized type:
          raw type:
            class: javacomp.basics.A
          actual type arguments:
            1. argument:
              type variable: T2
                bounds:
                  class: java.lang.Object
            2. argument:
              class: javacomp.basics.D
      generic exceptions:
```

Der Übersicht wegen ist in Bild 2.3 der Beginn der Programmausgabe (bis zu der Stelle …) grafisch dargestellt. Man sieht zum Beispiel, dass die Basisklasse von B ein ParameterizedType ist, dessen roher Typ die Klasse A ist, und deren Typargumente die Typvariable T1 und die Klasse C ist.

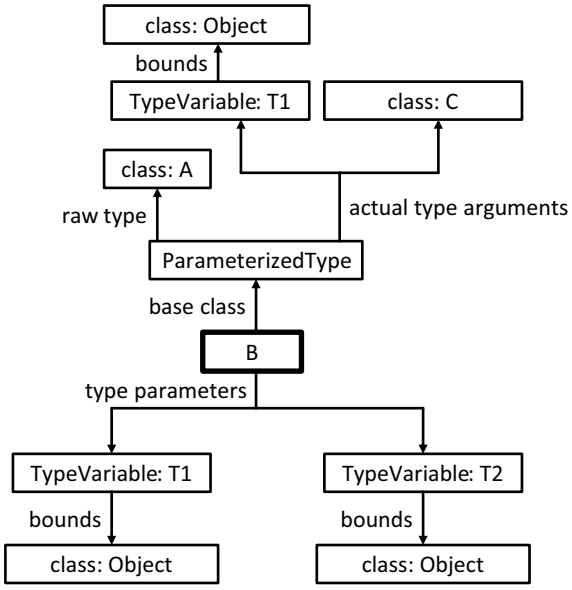

Bild 2.3 Grafische Darstellung der generischen Klasse B aus Listing 2.2

Zur Vertiefung empfehle ich, das Programm aus Listing 2.3 für weitere Klassen (u. a. auch mit Wildcards) auszuprobieren und die Ausgabe sorgfältig zu studieren.

Es soll abschließend noch darauf hingewiesen werden, dass das Programm, das Sie von der Web-Seite beziehen können, noch einige Zeilen Code enthält, die im obigen Listing nicht gezeigt sind. Es wird nämlich noch geprüft, ob eine Typvariable schon einmal behandelt wurde. Betrachten wir dazu folgendes Beispiel:

```
class X<T> {}
class Y<T extends X<T>> {}
```

Bei der Untersuchung der Klasse Y würde das Programm ohne diese Prüfung in eine unendliche Rekursion geraten, da beim Ausgeben der Typvariable T auch die obere Grenze X ausgegeben wird, die die Typvariable T als Parameter besitzt, die dann auch wieder ausgegeben würde. Dies würde sich unendlich oft wiederholen. Dies wird durch die zusätzlich eingebaute Prüfung jedoch verhindert.

3 Annotationen

Annotationen sind Zusatzinformationen, mit denen Klassen, Attribute, Methoden usw. kommentiert werden können. Wie Generics spielen sie für die Erzeugung des Byte-Codes durch den Compiler keine Rolle; sie können also keinen Einfluss auf die Byte-Code-Interpretation durch die Java Virtual Machine (JVM) haben. In dieser Hinsicht sind Annotationen tatsächlich ähnlich wie Kommentare. Allerdings sind Annotationen Teil der Sprache Java. Im Gegensatz zu reinen Kommentaren wird deshalb die syntaktische Korrektheit der Annotationen durch den Compiler überprüft. Annotationen können so deklariert werden, dass sie zur Laufzeit über Reflection ausgelesen und entsprechend interpretiert werden können. Auch dies ist eine Parallele zu Generics: Sie spielen zwar keine Rolle für den Byte-Code, aber unter Umständen für Reflection. Man kann Annotationen auch so definieren, dass sie zur Laufzeit nicht mehr sichtbar sind. Solche Annotationen sind für Programme gedacht, die den Quelltext eines Programms wie Compiler analysieren und basierend darauf z. B. zusätzlichen Programmcode produzieren.

Im Zusammenhang mit Komponenten-Software geht es meistens um Annotationen, die zur Laufzeit noch vorhanden sind. Man kann damit zum Beispiel das Konzept der Dependency Injection (siehe Abschnitt 3.6) realisieren.

3.1 Deklaration und Nutzung von Annotationen

Eine Annotation wird durch das Schlüsselwort `@interface` ähnlich wie eine Schnittstelle deklariert:

```
public @interface Special
{
}
```

Hat man eine solche Annotation definiert, kann man sie in seinem Programm zum Beispiel vor Klassen, Attributen, Konstruktoren, Parametern und Methoden verwenden:

```
@Special
public class AnnotatedClass
```

```
{
    @Special
    private int i;

    private String s;

    @Special
    public AnnotatedClass(int i, @Special String s)
    {
        this.i = i;
        this.s = s;
    }

    public int getI()
    {
        return i;
    }

    @Special
    public String getS()
    {
        return s;
    }
}
```

Statt `@Special` ist auch die Schreibweise `@Special()` möglich.

Annotationen können parametrisiert sein. Dazu definiert man wie in einer Schnittstelle Methoden:

```
public @interface Special
{
    public String scope();
    public int priority();
}
```

Jetzt kann die Annotation @Special nicht mehr so verwendet werden wie im ersten Beispiel (das wäre ein Syntaxfehler), sondern beispielsweise so:

```
@Special(scope="Class", priority=2)
public class AnnotatedClass
{
    @Special(priority=19, scope="Field")
    private int i;

    private String s;

    @Special(scope="Constructor", priority = 3)
    public AnnotatedClass(int i, String s)
    {
        this.i = i;
        this.s = s;
    }
    ...
}
```

In der Deklaration von @Special sind scope und priority wie Methoden definiert. Bei der Nutzung muss ihnen aber wie Attributen ein zum Rückgabetyp passender Wert zugewiesen werden. Die folgende Zeile ist z. B. ein Syntaxfehler, da priority einen Wert des Typs int verlangt:

```
@Special(scope="Class", priority="2") //Syntaxfehler
```

Die Fehlermeldung des Compilers ist so, wie wenn man eine Methode, die ein Argument des Typs int erwartet, mit einem String aufruft. Die Reihenfolge von scope und priority ist aber im Vergleich zu den Parametern bei einem Methodenaufruf beliebig, wie oben im Beispiel schon zu sehen ist. Es darf aber keines der Attribute fehlen. Diese Verwendung der Annotation @Special führt zu einem Syntaxfehler:

```
@Special(scope="Class") //Syntaxfehler
```

Falls es nur einen Parameter gibt und dieser value heißt, kann der Name des Parameters weggelassen werden (muss aber nicht):

```
public @interface Special
{
    public String value();
}
-----------------------------------------------------------------------
@Special("Class")
public class AnnotatedClass
{
    @Special(value="Field")
    private int i;
    ...
}
```

Als Rückgabetyp für die Methoden einer Annotation sind auch Felder möglich. Wie für andere Annotationen kann der Methodenname auch bei Feldern beliebig gewählt werden; der Name value ist nicht nötig, aber auch in diesem Fall möglich:

```
public @interface Special
{
    public String[] value();
}
-----------------------------------------------------------------------
@Special({"Class"})
public class AnnotatedClass
{
    @Special("Field")
    private int i;

    @Special({"Field", "private", "String", "s"})
    private String s;
    ...
}
```

Wie dem Beispiel entnommen werden kann, müssen bei der Angabe von mehr als einem Element die Elemente in geschweiften Klammern eingeschlossen sein. Bei einem Argument kann man die geschweiften Klammern ebenfalls angeben; man muss es aber nicht.

Wir haben oben gesehen, dass man bei der Nutzung einer Annotation alle Attribute angeben muss. Dies gilt nicht, falls Standardwerte für die Attribute einer Annotation festgelegt wurden:

```
public @interface Special
{
    public String scope() default "no scope";
    public int priority() default 5;
}
-----------------------------------------------------------------------
@Special(scope="Class", priority=2)
public class AnnotatedClass
{
    @Special(scope="Field")
    private int i;

    @Special(priority=2)
    private String s;

    @Special
    public void m() {}
}
```

Im obigen Beispiel gibt es Standardwerte für beide Attribute. Deshalb kann eines der beiden oder können beide Attribute weggelassen werden.

3.2 Meta-Annotationen

Die bisher definierten Annotationen sind alle zur Laufzeit nicht mehr vorhanden. Will man aber, dass eine Annotation zur Laufzeit über Reflection ausgelesen werden kann (s. Abschnitt 3.4), so muss sie durch eine entsprechende Annotation gekennzeichnet werden:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface Special
{
    public String scope();
    public int priority();
}
```

Man bezeichnet Annotationen für Annotationen als Meta-Annotationen. Die Meta-Annotation @Retention ist im Package java.lang.annotation definiert. Deshalb wurde die Import-Anweisung oben verwendet. Mögliche Werte für die @Retention-Annotation sind:

- Retention.RUNTIME: Wie oben dargestellt sind die derart gekennzeichneten Annotationen noch zur Laufzeit über Reflection auslesbar.
- Retention.CLASS: Dies ist der Standardwert, der verwendet wird, wenn keine @Retention-Annotation angegeben wird. Die Annotation wird mit in die vom Compiler erzeugte Class-Datei geschrieben. Würde man die Class-Datei analysieren, so würde man die Annotation finden. Über Reflection ist die Annotation aber nicht zu sehen.

- Retention.SOURCE: Die Annotation ist wie ein Kommentar nur im Quellcode vorhanden. Solche Annotationen sind nur für Programme sinnvoll, die den Quellcode analysieren.

Annotationen können auch auf bestimmte Stellen (nur vor Klassen, nur vor Methoden usw.) eingeschränkt werden; dies erfolgt ebenfalls über eine entsprechende Meta-Annotation:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface SpecialClass
{
    public String scope();
    public int priority();
}
-----------------------------------------------------------------
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface SpecialField
{
    public String name();
}
```

Die Meta-Annotation @Target definiert, vor welchen Elementen die betreffende Annotation vorkommen darf. Mögliche Werte für @Target sind einer oder mehrere dieser Werte (mehrere Werte sind möglich, da @Target ein Feld solcher Werte akzeptiert):

- ElementType.ANNOTATION_TYPE: Meta-Annotation (d. h. Annotation darf vor einer Annotation stehen)
- ElementType.CONSTRUCTOR: Annotation für Konstruktoren
- ElementType.FIELD: Annotation für Attribute
- ElementType.LOCAL_VARIABLE: Annotation für lokale Variable
- ElementType.METHOD: Annotation für Methoden
- ElementType.PACKAGE: Annotation für Packages
- ElementType.PARAMETER: Annotation für Parameter von Konstruktoren und Methoden
- ElementType.TYPE: Annotation für Klassen, Schnittstellen und Enumerations

Ist keine @Target-Annotation vorhanden, gibt es keine Einschränkung für das Vorkommen der betreffenden Annotation.

Die oben definierte Annotation @SpecialClass ist eine Annotation, die vor Klassen, Schnittstellen und Enumerations stehen darf, während die Annotation @SpecialField nur vor Attributen vorkommen darf. Der folgende Code ist damit fehlerhaft:

```
@SpecialField(name="Class") //Syntaxfehler
public class AnnotatedClass
{
    @SpecialClass(priority=19, scope="Field") //Syntaxfehler
    public void m() {}
    ...
}
```

Wie bei der Deklaration der Annotationen @SpecialClass und @SpecialField zu sehen ist, kann vor einem annotierten Element (Klasse, Methode oder Annotation) mehr als eine Annotation stehen (im Beispiel sind die Annotationen @SpecialClass und @SpecialField mit den Annotationen @Retention und @Target versehen). Dieses Mehrfachvorkommen von Annotationen gilt nicht nur für Meta-Annotationen, sondern allgemein für alle Annotationen. Es ist allerdings nicht möglich, dass dieselbe Annotation vor demselben Element mehrfach vorkommt, ganz gleich, ob die Attributwerte dieselben oder unterschiedlich sind:

```
@SpecialClass(priority=1, scope="Class") //Syntaxfehler
@SpecialClass(priority=2, scope="Class") //Syntaxfehler
public class AnnotatedClass
{
    ...
}
```

Zwei weitere Meta-Annotationen aus dem Package java.lang.annotation sind:

- @Documented: Die mit dieser Meta-Annotation gekennzeichneten Annotationen werden, wenn sie vorkommen, von Javadoc in die generierten HTML-Dateien übernommen.
- @Inherited: Die mit dieser Meta-Annotation gekennzeichneten Annotationen werden vererbt. Wenn eine Basisklasse mit einer solchen Annotation gekennzeichnet ist, dann wird diese Annotation in die abgeleitete Klasse übernommen.

3.3 Regeln für Annotationsdeklarationen

Allgemein gilt für die Deklaration von Annotationen:

- Die Methoden dürfen keine Parameter haben.
- Die Methoden dürfen nicht mit throws versehen sein.
- Als Rückgabetyp der Methoden sind nur erlaubt: alle primitiven Datentypen (wie int, short und long), String, Class, Enumerations, andere Annotationen sowie eindimensionale Felder der zuvor genannten Typen.

Der Fall, dass der Rückgabetyp einer Annotationsmethode eine andere Annotation ist, ist vielleicht nicht so ganz naheliegend. Wir geben deshalb ein Beispiel dafür an:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@interface AnnotationParameter
{
    public boolean value();
}

@Retention(RetentionPolicy.RUNTIME)
@interface AnnotationExample
{
    public String stringParam();
```

```
    public int intParam();
    public AnnotationParameter annoParam();
}

@AnnotationParameter(false)
public class NestedAnnotations
{
    @AnnotationExample(stringParam="hello",
                       intParam=19,
                       annoParam=@AnnotationParameter(true))
    public void m()
    {
    }
}
```

Wie in diesem Beispiel zu sehen ist, hat die Annotation @AnnotationExample die Methode annoParam, deren Rückgabetyp die Annotation @AnnotationParameter ist, wobei bei der Deklaration das @-Zeichen weggelassen werden muss. Bitte beachten Sie, dass die Annotation @AnnotationParameter so deklariert ist, dass sie nur vor Klassen, Schnittstellen und Enumerations vorkommen darf (ElementType.TYPE). Dies gilt aber nur für die Annotation selbst. Die Einschränkung hat keine Bedeutung, wenn die Annotation @AnnotationParameter als Attributwert einer anderen Annotation vorkommt. Im Beispiel wird @AnnotationParameter innerhalb von @AnnotationExample vor einer Methode verwendet. Ob dies möglich ist, hängt nur vom @Target der Annotation @AnnotationExample ab. Da diese nicht mit einer @Target-Annotation gekennzeichnet ist, ist ihr Vorkommen nicht eingeschränkt. Deshalb ist der obige Programmcode fehlerfrei.

3.4 Reflection für Annotationen

Sie haben bisher gesehen, wie man Annotationen definiert und in seinem Programm verwenden kann. Dabei ist hoffentlich deutlich geworden, dass für diese Annotationen kein ausführbarer Programmcode erzeugt werden kann („Annotationen tun nichts“). Die Annotationen bekommen erst dadurch Bedeutung, dass andere Programme die annotierten Programme betrachten und aufgrund der vorhandenen Annotationen irgendwelche Aktionen durchführen. Die Annotationen kann man im Quellcode, in der Class-Datei oder während der Laufzeit über Reflection auswerten. Wir beschränken uns in diesem Buch auf die zuletzt genannte Möglichkeit.

Die im Reflection-Kapitel vorgestellten Klassen Class, Field, Constructor und Method besitzen die Methoden getAnnotations und getDeclaredAnnotations zum Auslesen von Annotationen (Rückgabetyp für beide Methoden ist Annotation[]). Damit können die Annotationen einer Klasse bzw. einer Schnittstelle, eines Attributs, eines Konstruktors oder einer Methode abgefragt werden. Ähnlich wie bei GenericDeclaration werden die Methoden getAnnotations und getDeclaredAnnotations (und zwei andere, die später noch besprochen werden) in eine Schnittstelle gepackt, die AnnotatedElement heißt und von den genannten Klassen implementiert wird. Der Unterschied zwischen den beiden Varianten mit und ohne Declared ist wie bei den bisher besprochenen Reflection-Methoden, z. B. getMethods und getDeclared-

Methods; die Declared-Variante beschränkt sich auf die Annotationen in der betrachteten Klasse, während die Variante ohne Declared auch vererbte Annotationen mit einbezieht (wie in Abschnitt 3.2 kurz angesprochen wurde, werden Annotationen, die mit der Meta-Annotation @Inherited markiert sind, vererbt). Wie schon besprochen wurde, kann eine Klasse, Attribut, Methode usw. mit mehreren Annotationen versehen sein. Deshalb ist der Rückgabetyp der Methoden getAnnotations und getDeclaredAnnotations ein Feld des Typs Annotation.

Parameter von Konstruktoren und Methoden können ebenfalls annotiert werden. Deshalb haben die beiden Klassen Constructor und Method auch noch die Methode getParameter-Annotations. Der Rückgabetyp ist in diesem Fall ein zweidimensionales Feld des Typs Annotation (Annotation[][]), denn ein Konstruktor und eine Methode können mehrere Parameter haben, und jeder Parameter kann mehrfach annotiert sein.

Die von den erwähnten Methoden zurückgelieferten Annotationsobjekte sind Objekte irgendwelcher nicht näher interessierenden Klassen. Eine solche Klasse implementiert eine Schnittstelle, die man aus einer Annotationsdeklaration erhält, wenn man das Schlüsselwort @interface gedanklich durch interface ersetzt. Über diese Schnittstelle kann man dann durch Aufruf ihrer Methoden die Attributwerte der betreffenden Annotation auslesen. Alle diese Schnittstellen sind abgeleitet aus der Schnittstelle Annotation, die in den soeben beschriebenen Methoden bei der Rückgabe vorkommt. Die Schnittstelle Annotation ist im Package java.lang.annotation definiert.

Diese Sachverhalte sind vermutlich ohne Beispiel schwer zu verstehen. Betrachten wir die zuvor schon einmal gezeigte Annotation @SpecialClass für Klassen und Schnittstellen:

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface SpecialClass
{
    public String scope();
    public int priority();
}
```

Diese Annotation werde für eine Klasse X wie folgt verwendet:

```
package javacomp.basics;

@SpecialClass(priority=19, scope="Class")
public class X
{
}
```

Wenn wir getAnnotations auf das Class-Objekt, welches die Klasse X repräsentiert, aufrufen, erhalten wir ein Feld des Typs Annotation mit der Länge 1, da die Klasse X genau eine Annotation hat. Das einzige Objekt in diesem Feld implementiert die Schnittstelle SpecialClass. Die Deklaration der Schnittstelle SpecialClass müssen wir uns so vorstellen, als ob in der Deklaration der Annotation @SpecialClass @interface durch interface ersetzt worden wäre. SpecialClass ist aus der Schnittstelle Annotation abgeleitet, so dass der Rückgabetyp Annotation[] passend ist.

In den folgenden Codezeilen gehen wir davon aus, dass die Klasse X genau diese eine Annotation @SpecialClass hat. Wir casten das erste (und einzige) Annotationsobjekt im von getAnnotations zurückgelieferten Feld ohne weitere Prüfung auf SpecialClass. Damit können wir dann die Attributwerte 19 und „Class“ für diese Annotation auslesen:

```
Class<?> c = Class.forName("javacomp.basics.X");
Annotation[] classAnnotations = c.getAnnotations();
SpecialClass s = (SpecialClass) classAnnotations[0];
System.out.print("priority = " + s.priority());
System.out.println(", scope = " + s.scope());
```

Die Ausgabe dieser Codezeilen ist:

```
priority = 19, scope = Class
```

Wenn man nicht weiß, welche Annotationen vorhanden sind, kann man auf jedes Annotation-Objekt die parameterlose Methode annotationType anwenden. Diese liefert ein Objekt des Typs Class<? extends Annotation> zurück. Dieses Class-Objekt repräsentiert die zu der Annotation gehörende Schnittstelle. Das heißt, für das obige Beispiel erhalten wir ein Class-Objekt, das die Schnittstelle SpecialClass mit den Methoden priority und scope repräsentiert. Mit Hilfe von Reflection lassen sich die Methoden jeder Annotation aufrufen (ohne dass man beim Programmieren wissen muss, welche Methoden die Annotation mitbringt). Damit lassen sich dann die Werte der Annotation bestimmen. Der Aufruf erfolgt durch das Anwenden der Methode invoke auf das dazugehörige Method-Objekt. Als Parameter von invoke muss das Objekt angegeben werden, auf das die durch das Method-Objekt repräsentierte Methode angewendet werden soll; das ist in diesem Fall das Annotationsobjekt. Ferner müssen die Argumente für den Methodenaufruf angegeben werden. Da alle Methoden einer Annotation parameterlos sein müssen, wird deshalb als zweites Argument von invoke ein Object-Feld der Länge 0 übergeben. Das Programm in Listing 3.1 illustriert diesen Sachverhalt:

Listing 3.1 Programm zum Auslesen von Annotationen mit Reflection

```
package javacomp.basics;

public class Main
{
    public static void main(String[] args)
    {
        try
        {
            Class<?> c = Class.forName("javacomp.basics.X");
            Annotation[] classAnnotations = c.getAnnotations();
            for(Annotation anno : classAnnotations)
            {
                printAnnotation(anno);
            }
        }
        catch(Exception e)
        {
        }
    }

    private static void printAnnotation(Annotation anno)
```

```
                        throws Exception
    {
        Class<?> annoClass = anno.annotationType();
        if(annoClass.isInterface())
        {
            System.out.print("interface ");
        }
        else
        {
            //dieser Fall kommt nie vor
            System.out.print("class ");
        }
        System.out.println(annoClass.getName());
        System.out.println("{");
        Method[] declaredMethods1 = annoClass.getDeclaredMethods();
        for(Method m : declaredMethods1)
        {
            System.out.print("   " + m.getReturnType().getName());
            System.out.print(" " + m.getName() + "(): ");
            System.out.println(m.invoke(anno, new Object[]{}));
        }
        System.out.println("}");
    }
}
```

Dieser Code erzeugt die folgende Ausgabe:

```
interface javacomp.basics.SpecialClass
{
   int priority(): 19
   java.lang.String scope(): Class
}
```

Statt durch einen Aufruf von annotationType kann man genau dasselbe Class-Objekt erhalten, indem man auf das von getAnnotations zurückgelieferte Annotation-Objekt zunächst die Methode getClass aufruft (dies liefert das Class-Objekt der nicht näher interessierenden Klasse) und darauf dann die Methode getInterfaces. Der Aufruf von getInterfaces gibt ein Class-Feld der Länge 1 zurück, das genau das Class-Objekt erhält, das man durch den Aufruf von annotationType erhält.

So wie es zu der Reflection-Methode getMethods auch die Singularvariante getMethod gibt, enthält die Schnittstelle AnnotatedElement, die von Class, Field, Constructor und Method implementiert wird, auch eine Methode namens getAnnotation. Wie bei getMethod, wo man den Namen der Methode und ihre Parametertypen angeben muss, muss man die Annotation, nach der man sucht, bereits kennen, da man sie als Parameter vorgeben muss. Dies dürfte in der Praxis der Normalfall sein: In der Regel wird man nämlich überprüfen, ob eine Klasse, ein Attribut oder ein Methode eine speziell vorgegebene Annotation besitzt, um dann entsprechend darauf zu reagieren. Die Angabe der Annotation, nach der man sucht, erfolgt in Form eines Class-Objekts. Es handelt sich dabei genau um das oben diskutierte Class-Objekt, das die Schnittstelle repräsentiert, die der Annotation entspricht. Die Signatur der Methode getAnnotation ist nur mit Generics-Kenntnissen zu verstehen:

```
public <T extends Annotation> T getAnnotation(Class<T> annoClass)
```

Die generische Methode getAnnotation benötigt als Parameter ein Class<T>-Objekt und liefert dann ein Objekt des Typs T zurück. Hat das Element (Klasse, Attribut, Methode usw.), auf das man die Methode getAnnotation anwendet, die betreffende Annotation nicht, wird null zurückgeliefert. Im folgenden Beispiel, in dem wir noch einmal die Klasse X verwenden, gehen wir davon aus, dass X mit @SpecialClass annotiert ist. Wir prüfen deshalb nicht auf null ab:

```
Class<?> c = Class.forName("javacomp.basics.X");
SpecialClass s = c.getAnnotation(SpecialClass.class);
System.out.print("priority = " + s.priority());
System.out.println(", scope = " + s.scope());
```

Im Beispiel hat der Parameter beim Aufruf von getAnnotation den Typ Class<SpecialClass>. Der Rückgabetyp ist deshalb SpecialClass. Man muss also bei der Zuweisung des Rückgabewerts an die lokale SpecialClass-Variable nicht casten.

Will man nur wissen, ob eine Annotation vorhanden ist oder nicht, so gibt es noch die Methode isAnnotationPresent. Dies ist die vierte Methode der Schnittstelle AnnotatedElement, die von Class, Field, Constructor und Method implementiert wird:

```
public boolean isAnnotationPresent(Class<? extends Annotation> cl)
```

Zum Abschluss dieses Abschnitts soll noch erwähnt werden, dass sich Annotationen lokaler Variablen mit Reflection nicht auslesen lassen.

3.5 Beispiel

Wie im Kapitel über Reflection kann man ein Programm schreiben, das alle Annotationen von Klassen, deren Namen als Kommandozeilenargumente dem Programm mitgegeben werden, mitsamt ihren Werten ausgibt. Der Abdruck des Programms würde wieder sehr viel Platz beanspruchen, den wir einsparen wollen. Der Kern des Programms ist die Methode printAnnotation aus Listing 3.1. Diese Methode wird für alle Annotationen der Klasse sowie jedes Attributs, jedes Konstruktors und jeder Methode der Klasse aufgerufen. Als Beispiel betrachten wir die annotierte Klasse aus Listing 3.2, wobei die Deklaration der Annotationen weggelassen wird.

Listing 3.2 Beispielprogramm zum Auslesen von Annotationen

```
package javacomp.basics;

@SpecialClass(scope="Class", priority=12)
public class AnnotatedClass
{
    @SpecialField(name="Field")
    private int i;

    private String s;

    @SpecialConstructor(priority = 3)
```

```
    public AnnotatedClass(int i, String s)
    {
        this.i = i;
        this.s = s;
    }

    public AnnotatedClass()
    {
        this.i = 0;
        this.s = "";
    }

    public int getI()
    {
        return i;
    }

    @SpecialMethod("Method")
    public String getS()
    {
        return s;
    }
}
```

Die Ausführung unseres Analyseprogramms mit dem Kommandozeilenargument javacomp.basics.AnnotatedClass erzeugt die folgende Ausgabe:

```
javacomp.basics.AnnotatedClass:
-------------------------------
  class annotations:
    interface javacomp.basics.SpecialClass
    {
      public abstract int priority(): 12
      public abstract java.lang.String scope(): Class
    }
  annotations for attribute i:
    interface javacomp.basics.SpecialField
    {
      public abstract java.lang.String name(): Field
    }
  annotations for attribute s:
  annotations for 1. constructor:
    interface javacomp.basics.SpecialConstructor
    {
      public abstract int priority(): 3
    }
  annotations for 2. constructor:
  annotations for method getI:
  annotations for method getS:
    interface javacomp.basics.SpecialMethod
    {
      public abstract java.lang.String value(): Method
    }
```

Auch das Programm zur Ausgabe aller Annotationen kann von der Web-Seite zum Buch heruntergeladen werden.

3.6 Anwendung: Dependency Injection

In der Einleitung wurde erwähnt, dass Annotationen u. a. für Dependency Injection eingesetzt werden können. Unter Dependency Injection versteht man, dass in einer Klasse ein Attribut definiert, dieses mit einer bestimmten Annotation versehen, aber diesem Attribut kein Wert zugewiesen wird. Durch eine weitere Software, die einen Ausführungsrahmen (eine Art Basis-Software, in der Regel auch Framework genannt) für die oben erwähnte Klasse bildet, wird mittels Reflection untersucht, welche Attribute der Klasse die gewisse Annotation tragen. Diesen Attributen wird dann über Reflection ein vom Framework und vom Typ des Attributs abhängiger Wert zugewiesen. Diese Zuweisung von „außerhalb der Klasse“, also durch das Framework, wird als Injektion bezeichnet. Mit Dependency (Abhängigkeit) wird darauf Bezug genommen, dass ein Objekt a von einem anderen Objekt b abhängig ist, wenn a eine Referenz auf b hat. Dependency Injection könnte man also übersetzen durch das Zuweisen von Werten an Attribute durch eine Art von Framework-Software. Häufig spricht man in diesem Zusammenhang auch von der „Verdrahtung“ von Objekten.

Bevor wir uns den Code, der die Dependcy Injection realisiert, ansehen, wird zunächst erläutert, was diese Software leisten soll. Zunächst einmal definieren wir die Annotation @Inject (s. Listing 3.3):

Listing 3.3 Deklaration der Annotation @Inject

```
package javacomp.basics;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Inject
{
}
```

Unser Dependency-Injection-Programm erwartet als Kommandozeilenargumente die Namen von beliebig vielen Klassen. Es erzeugt zunächst zu jeder der angegebenen Klassen ein Objekt. Es wird davon ausgegangen, dass zu diesem Zweck jede Klasse einen parameterlosen Konstruktor hat. Ist dies nicht der Fall, so kann kein Objekt erzeugt werden und die betreffende Klasse wird im weiteren Verlauf nicht mehr berücksichtigt. Dann werden für die Klasse jedes so erzeugten Objekts alle Attribute untersucht. Falls ein Attribut die Annotation @Inject trägt und ihr Datentyp eines der angegebenen Kommandozeilenargumente ist, wird dem Attribut eine Referenz auf das zu dieser Klasse erzeugte Objekt zugewiesen (falls die Erzeugung möglich war). Welche Sichtbarkeit die Attribute haben, spielt dabei keine Rolle, da eventuell vorhandene Sichtbarkeitseinschränkungen durch setAccessible(true) aufgehoben werden. Zusammengefasst heißt das, dass unser Dependency-Injection-Beispielprogramm je ein Objekt jeder angegebenen Klasse erzeugt, die dann entsprechend der Annotation @Inject „verdrahtet“ werden.

Listing 3.4 zeigt einige Beispielklassen, in denen die Annotation @Inject verwendet wird:

Listing 3.4 Beispielklassen für die Anwendung von Dependency Injection

```
package javacomp.basics;

class A
{
    @Inject
    private B b;

    @Inject
    private C c;
}

class B
{
    @Inject
    private A a;
}

class C
{
}
```

Wenn unser Dependency-Injection-Programm mit den drei Kommandozeilenargumenten javacomp.basics.A, javacomp.basics.B und javacomp.basics.C gestartet wird, wird je ein Objekt der Klasse A, B und C erzeugt, die dann so „verdrahtet" werden, wie es in dem UML-Objektdiagramm in Bild 3.1 zu sehen ist. Ein Objekt wird in einem Objektdiagramm durch einen optionalen Bezeichner beschriftet, der auch weggelassen werden kann, wie dies in Bild 3.1 zu sehen ist. Dann folgt ein Doppelpunkt und danach der Name der Klasse. Die Beschriftung ist für Objekte immer unterstrichen (im Gegensatz zu Klassen und Schnittstellen in einem UML-Klassendiagramm).

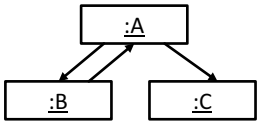

Bild 3.1 „Verdrahtung" der Beispielobjekte zu den Klassen aus Listing 3.4

In Listing 3.5 ist der komplette Programmcode für dieses einfache Beispiel einer Dependency Injection zu finden.

Listing 3.5 Beispielprogramm zur Dependency Injection

```
package javacomp.basics;

import java.lang.reflect.*;
import java.util.*;

public class DependencyInjection
{
    public static void main(String[] args)
    {
        if(args.length == 0)
```

```
        {
            System.out.println("arguments: list of class names");
            return;
        }
        HashMap<String,Object> map = new HashMap<String,Object>();
        createObjects(args, map);
        wireObjects(map);
    }

    private static void createObjects(String[] args,
                                      HashMap<String,Object> map)
    {
        for(String className: args)
        {
            try
            {
                if(!map.containsKey(className))
                {
                    Class<?> c = Class.forName(className);
                    Object o = c.newInstance();
                    map.put(className, o);
                }
            }
            catch(Exception e)
            {
                System.out.println("Error in createObjects: " +
                                   e.getMessage());
            }
        }
    }

    private static void wireObjects(HashMap<String,Object> map)
    {
        Collection<Object> objects = map.values();
        for(Object object: objects)
        {
            Class<?> c = object.getClass();
            Field[] attributes = c.getDeclaredFields();
            for(Field attribute: attributes)
            {
                if(attribute.isAnnotationPresent(Inject.class))
                {
                    String attClassName =
                                       attribute.getType().getName();
                    Object target = map.get(attClassName);
                    if(target != null)
                    {
                        attribute.setAccessible(true);
                        try
                        {
                            attribute.set(object, target);
                        }
                        catch(Exception e)
                        {
                            System.out.println("Error in " +
                                               "wireObjects: " +
                                               e.getMessage());
                        }
```

```
                        }
                    }
                }
            }
        }
    }
```

Das Programm verwendet eine Hash-Tabelle. Als Schlüssel wird der Name der Klasse verwendet, während der Wert das dazu erzeugte Objekt ist. Die Objekterzeugung erfolgt in der Methode createObjects. Falls ein Klassenname mehr als einmal angegeben wird, wird er nur einmal berücksichtigt. Falls eine Klasse keinen parameterlosen Konstruktor besitzt, wird beim Aufruf der Methode newInstance eine Ausnahme geworfen. Die „Verdrahtung“ wird in der Methode wireObjects vorgenommen. Es werden für jedes in der Hash-Tabelle eingetragene Objekt alle Attribute der dazugehörigen Klasse untersucht. Ist die Annotation @Inject bei einem Attribut vorhanden, wird in der Hash-Tabelle gesucht, ob es für den Typ des Attributs einen Eintrag in der Hash-Tabelle gibt. Falls dies der Fall ist (`target != null`), wird die eigentliche Dependency Injection durch die Anweisung `attribute.set(object, target)` realisiert.

4 Dynamische Proxies

Der Begriff Proxy, den man mit Stellvertreter(in) übersetzen kann, soll zunächst anhand eines Beispiels aus dem täglichen Leben erläutert werden, und zwar anhand einer Zeitschaltuhr. Eine Zeitschaltuhr wird in eine Stromsteckdose eingesteckt. Sie liefert selbst wieder die Möglichkeit, dass ein Stecker in die von ihr bereitgestellte Steckdose eingesteckt wird. Das heißt also, dass die Zeitschaltuhr eine Schnittstelle (hier die Steckdose) benutzt und dieselbe Schnittstelle für andere wieder bereitstellt (im Gegensatz zu einem Adapter, der in der Regel eine Schnittstelle benutzt, die verschieden ist von der, die er anbietet). Der Stecker eines Geräts kann somit direkt in die Steckdose gesteckt werden, aber genauso auch in die Zeitschaltuhr, die in der Steckdose steckt. Durch diese Form der Indirektion erhält man eine Zusatzfunktionalität, in diesem Fall die zeitgesteuerte Versorgung der Steckdose mit Strom.

Der Begriff Proxy kommt in der Informatik an vielen Stellen vor. Bekannt sind z. B. Proxies für das World Wide Web: Ein Browser sendet den Befehl zum Abrufen einer Web-Seite nicht an den Web-Server, der die Seite bereitstellt, sondern an einen Proxy. Der Proxy, der einen Cache für abgerufene Web-Seiten besitzt, prüft, ob er die Seite bereits in seinem Cache hat. Wenn ja, wird die Seite an den anfragenden Browser ausgeliefert. Wenn nein, wird der Befehl vom Proxy an den Original-Web-Server weitergeleitet, der mit den Daten der gewünschten Web-Seite antwortet. Die Daten werden in den Proxy-Cache übernommen und dann an den Browser weitergeleitet. Die Zusatzfunktion, die in diesem Fall vom Proxy geliefert wird, ist eine Reduzierung der Antwortzeit für mehrfach abgerufene Web-Seiten. Der Web-Proxy spielt (wie die Zeitschaltuhr) sowohl die Rolle eines Servers als auch eines Clients: Er nimmt wie ein Server Befehle entgegen (die Zeitschaltuhr hat eine Steckdose) und sendet selbst Befehle an andere Server aus (die Zeitschaltuhr hat einen Stecker zur Benutzung von Steckdosen). In der Rolle des Servers stellt ein Proxy eine Schnittstelle bereit (der Web-Proxy in Form einer HTTP-Schnittstelle, die Zeitschaltuhr für eine bestimmte Form einer Stromsteckdose). Genau diese Schnittstelle benutzt der Proxy in der Rolle des Clients selbst. Dieser Sachverhalt ist in Bild 4.1 grafisch dargestellt.

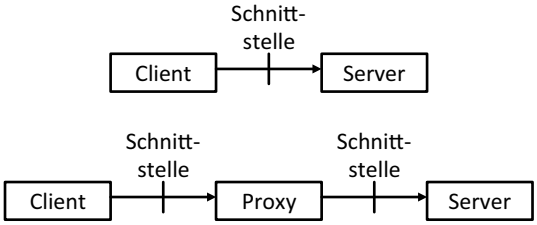

Bild 4.1 Bereitstellung und Nutzung einer Schnittstelle ohne und mit Proxy

Wir haben das Proxy-Prinzip jetzt an einem Beispiel aus dem täglichen Leben und einem Client-Server-Szenario erläutert. Die in diesem Zusammenhang erwähnten Proxies waren statischer Natur. Dies bedeutet, dass sie für eine ganz spezielle Schnittstelle (HTTP, Form einer Steckdose für ein bestimmtes Land) angefertigt worden sind. Im restlichen Teil dieses Kapitels geht es jetzt wieder um das Zusammenwirken von Java-Objekten. Unter Schnittstellen verstehen wir ab jetzt wieder Java-Schnittstellen. In diesem Kontext ist ein dynamischer Proxy ein Stellvertreterobjekt einer Klasse, wobei diese Klasse dynamisch (d. h. zur Laufzeit) aufgrund einer oder mehrerer vorgegebenen Schnittstellen erzeugt wird.

Dynamische Proxies werden im Zusammenhang mit Komponenten-Software z. B. bei der aspektorientierten Programmierung (s. später) eingesetzt. Zum leichteren Verständnis dynamischer Proxies betrachten wir zunächst den statischen Fall.

4.1 Statische Proxies

Statische Proxies werden vor dem Starten eines Programms erzeugt. Wir betrachten als Basis für ein nachfolgendes Beispiel mit statischem Proxy das Beispielprogramm in Listing 4.1.

Listing 4.1 Nutzung eines Objekts über eine Schnittstelle ohne Proxy

```
package javacomp.basics;

interface Counter
{
    public int add(int i);
    public int sub(int i);
}

class CounterImpl implements Counter
{
    private int c;

    public int add(int i)
    {
        c += i;
        return c;
    }

    public int sub(int i)
    {
        c -= i;
        return c;
    }
}

public class Main
{
    private static void useCounter(Counter counter)
    {
        //z.B.
```

```
        counter.add(7);
        counter.sub(6);
        ...
    }

    public static void main(String[] args)
    {
        CounterImpl counterImpl = new CounterImpl();
        useCounter(counterImpl);
    }
}
```

Der Schnittstelle in Bild 4.1 entspricht die Schnittstelle Counter in Listing 4.1, dem Server entspricht die Klasse CounterImpl bzw. das Objekt dieser Klasse, und dem Client entspricht die Methode useCounter. Da wir im Folgenden einen Proxy zwischen Client und Server schalten wollen ohne den Programmcode von Client und Server zu ändern, ist es günstig, wenn in der Methode useCounter der Typ des Parameters die Schnittstelle Counter ist (und nicht die implementierende Klasse CounterImpl). So können wir im nächsten Programmbeispiel die Methode useCounter mit einem Proxy-Objekt als Argument aufrufen, da die Proxy-Klasse die Schnittstelle Counter ebenfalls implementieren wird.

Die Zusatzfunktion unserer manuell erstellten, statischen Proxy-Klasse soll Logging sein. Das bedeutet, dass alle Methodenaufrufe der Schnittstelle Counter samt der von diesen Aufrufen zurückgelieferten Werte protokolliert und auf dem Bildschirm ausgegeben werden. Das Programm mit statischem Proxy findet sich in Listing 4.2.

Listing 4.2 Nutzung eines Objekts über einen statischen Proxy

```
package javacomp.basics;

//interface Counter und class CounterImpl wie zuvor:
...

class StaticProxy implements Counter
{
    private Counter counter;

    public StaticProxy(Counter counter)
    {
        this.counter = counter;
    }

    public int add(int i)
    {
        System.out.println("---> javacomp.basics.Counter.add(" +
                           i + ")");
        int result = counter.add(i);
        System.out.println("<--- return: " + result);
        return result;
    }

    public int sub(int i)
    {
        System.out.println("--->javacomp.basics.Counter.sub(" +
                           i + ")");
        int result = counter.sub(i);
```

```
        System.out.println("<--- return: " + result);
        return result;
    }
}

public class Main
{
    private static void useCounter(Counter counter)
    {
        //wie zuvor:
        ...
    }

    public static void main(String[] args)
    {
        CounterImpl counterImpl = new CounterImpl();
        StaticProxy staticProxy =
                        new StaticProxy(counterImpl);
        useCounter(staticProxy);
    }
}
```

Der statische Proxy wird durch die Klasse StaticProxy verkörpert. Da diese Klasse die Schnittstelle Counter implementiert, ist sie sozusagen ein Server. Gleichzeitig benutzt sie die Counter-Schnittstelle und spielt somit die Rolle eines Clients. Wie bei useCounter ist es von Vorteil, wenn wir auch in StaticProxy wieder mit dem Schnittstellentyp Counter und nicht der Klasse CounterImpl arbeiten. Dadurch erhalten wir die Möglichkeit, mehrere Proxies zwischen den Client und Server zu setzen. Nehmen Sie dazu an, dass es weitere Klassen StaticProxy2 und StaticProxy3 gibt, die eine Protokollierung der Methodenaufrufe in eine Datei bzw. in eine Datenbank realisieren. Man könnte die Funktionalität aller drei Proxies dann kombinieren, wobei in diesem Fall die Reihenfolge der Proxies beliebig ist, da sie nicht voneinander abhängig sind. Wir nehmen für den folgenden Programmcode an, dass die Klasse StaticProxy in StaticProxy1 umbenannt wurde. Das Hintereinanderschalten mehrerer Proxies könnte dann so realisiert werden:

```
public class Main
{
    private static void useCounter(Counter counter)
    {
        //wie zuvor:
        ...
    }

    public static void main(String[] args)
    {
        CounterImpl counterImpl = new CounterImpl();
        StaticProxy staticProxy1 =
                        new StaticProxy1(counterImpl);
        StaticProxy2 staticProxy2 =
                        new StaticProxy2(staticProxy1);
        StaticProxy3 staticProxy3 =
                        new StaticProxy3(staticProxy2);
        useCounter(staticProxy3);
    }
}
```

Die Situation ist in Bild 4.2 noch einmal grafisch dargestellt.

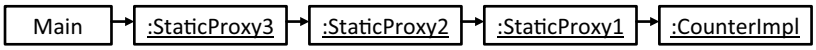

Bild 4.2 UML-Objektdiagramm zur Hintereinanderschaltung von Proxies

Das Hintereinanderschalten ist ein allgemeines Prinzip von Proxies. Es gilt nicht nur für statische, sondern auch für dynamische Proxies. Es kann auch bei den eingangs des Kapitels erwähnten Web-Proxies und sogar bei Zeitschaltuhren angewendet werden; bei Zeitschaltuhren kann man damit wesentlich vielfältigere Zeitmuster erzeugen als mit nur einer einzigen Zeitschaltuhr. Aufgrund der breiten Einsetzbarkeit dieses Prinzips wurde es als Entwurfsmuster klassifiziert. Bevor wir darauf zu sprechen kommen, wollen wir die Referenz auf das Counter-Objekt, die alle StaticProxy-Klassen als Gemeinsamkeit haben, in eine gemeinsame Basisklasse auslagern:

```
abstract class StaticProxy implements Counter
{
    protected Counter counter;

    public StaticProxy(Counter counter)
    {
        this.counter = counter;
    }

    public abstract int add(int i);
    public abstract int sub(int i);
}

class StaticProxy1 extends StaticProxy
{
    public StaticProxy1(Counter counter)
    {
        super(counter);
    }
    public int add(int i)
    {
        ...
    }
    public int sub(int i)
    {
        ...
    }
}

//StaticProxy2 und StaticProxy3 analog:
...
```

In dieser Form entspricht die Implementierung unserer statischen Proxies, die in Bild 4.3 als UML-Klassendiagramm dargestellt ist, dem Entwurfsmuster Decorator.

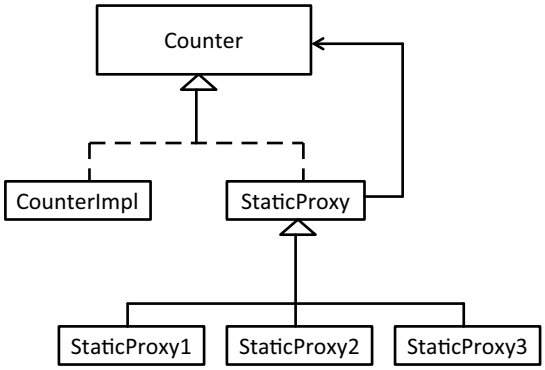

Bild 4.3 UML-Klassendiagramm zur Verkettung von Proxies (Decorator)

Das Entwurfsmuster Decorator kommt zum Beispiel auch in der Eingabe-Ausgabe-Bibliothek von Java vor. Hier gibt es zum Beispiel eine abstrakte OutputStream-Klasse mit mehreren Write-Methoden (entspricht in Bild 4.3 der Schnittstelle Counter). Daraus ist eine Klasse wie FileOutputStream zum Schreiben in Dateien abgeleitet (entspricht in Bild 4.3 der Klasse CounterImpl). Das Pendant zur StaticProxy-Klasse ist die Klasse FilterOutputStream, die ein Attribut des Typs OutputStream besitzt. Aus FilterOutputStream ist dann zum Beispiel CipherOutputStream zur Verschlüsselung und DeflaterOutputStream zur Komprimierung abgeleitet. Man kann dann mit den Methoden der abstrakten Klasse OutputStream komprimiert und verschlüsselt in eine Datei schreiben, indem man vor ein FileOutputStream-Objekt je ein Proxy-Objekt der Klassen CipherOutputStream und DeflaterOutputStream schaltet.

In diesem Abschnitt wurden die statischen Proxy-Klassen manuell erstellt. Dies muss nicht unbedingt so sein. Auch eine automatische Erzeugung einer statischen Proxy-Klasse ist durchaus möglich. So wurde zum Beispiel in früheren RMI-Versionen (vor Java-Version 5) die RMI-Stub-Klasse vom sogenannten RMI-Compiler rmic erzeugt. Der RMI-Compiler bekam eine oder mehrere RMI-Schnittstellen als Eingabe und erzeugte dann eine Klasse, die genau diese Schnittstellen implementierte. In den automatisch generierten Methoden werden die für den Methodenaufruf nötigen Daten wie Objektkennung, Methodenname und Parameter über eine Netzverbindung an einen RMI-Server gesendet, auf eine Antwort von diesem RMI-Server gewartet und der über die Netzverbindung empfangene Rückgabewert von der Methode zurückgegeben.

Egal ob manuell oder automatisch generiert: Das Thema dieses Abschnitts waren statische Proxy-Klassen, die beim Starten eines Programms schon existieren. Im Gegensatz dazu befassen wir uns im restlichen Teil dieses Kapitels mit dynamischen Proxy-Klassen, die erst zur Laufzeit durch das laufende Programm erzeugt werden. Hierzu gibt es zwei Wege: mit Hilfe von Reflection oder mit Hilfe der CGLIB.

4.2 Erzeugung dynamischer Proxies mit Reflection

Mit Hilfe der Reflection-Bibliothek kann eine Klasse zur Laufzeit generiert werden, die eine oder mehrere vorgegebene Java-Schnittstellen implementiert. Voraussetzung dafür ist logischerweise, dass solche Schnittstellen vorhanden sind (im Gegensatz dazu benötigt man zur Erzeugung von Klassen mit Hilfe der CGLIB keine Java-Schnittstellen, wie im folgenden Abschnitt zu sehen sein wird). Zu einer dynamisch erzeugten Dynamic-Proxy-Klasse existiert weder eine Java- noch eine Class-Datei, sondern das Class-Objekt und der Bytecode existieren nur temporär in der JVM (Java Virtual Machine), also nur im Adressraum des Java-Prozesses.

Wenn eine Klasse erzeugt wird, die eine oder mehrere Schnittstellen implementiert, besitzt sie logischerweise eine Menge von Methoden. Es stellt sich nun die Frage, was diese Methoden tun, d. h. welcher Code für diese Methoden generiert wird. Die Antwort dazu lautet: Für alle Methoden wird Code generiert, der den Methodenaufruf an die einzige Methode invoke der Schnittstelle InvocationHandler aus der Reflection-Bibliothek delegiert:

```
package java.lang.reflect;

public interface InvocationHandler
{
    public Object invoke(Object proxy, Method m, Object[] args)
           throws Throwable;
}
```

Die übergebenen Parameter sind das dynamische Proxy-Objekt, von dem der Aufruf kommt (also this aus Sicht des aufrufenden dynamischen Proxy-Objekts), die aufgerufene Methode in Form eines Method-Objekts (Klasse Method aus der Reflection-Bibliothek) sowie die Parameter der Methode m in Form eines Object-Felds. Sie fragen sich nun vielleicht, auf welches Objekt die Methode invoke angewendet wird. Die Antwort darauf lautet: Die dynamisch erzeugte Proxy-Klasse besitzt einen Konstruktor mit einem Parameter des Typs InvocationHandler. Das heißt, dass man eine eigene Klasse programmieren muss, die die Schnittstelle InvocationHandler implementiert. In dieser Klasse schreibt man durch Implementierung der Methode invoke den Code, der die Funktionalität des dynamischen Proxy eigentlich ausmacht. Man erzeugt dann ein Objekt dieser Klasse und übergibt dieses Objekt als Konstruktorargument beim Erzeugen eines Objekts der dynamisch erzeugten Proxy-Klasse. An dieses als Konstruktorargument übergebene Objekt werden alle Methodenaufrufe des dynamischen Proxy delegiert. In Bild 4.4 ist die Situation grafisch dargestellt, wobei mit I die Server-Schnittstelle bezeichnet wird, die auch vom dynamischen Proxy implementiert wird. In dieser Abbildung wird davon ausgegangen, dass der Code der InvocationHandler implementierenden Klasse auf den Server zugreift, vor den das Proxy-Objekt gesetzt wird. Dies ist zwar nicht zwingend notwendig, ist aber für Proxies der Normalfall, wie in der Einleitung zu diesem Kapitel dargestellt wurde.

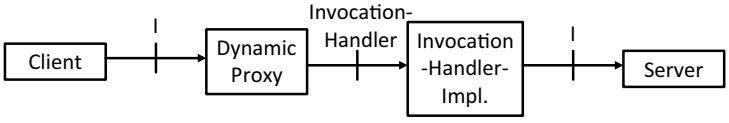

Bild 4.4 Strukturbild für einen mit Reflection erzeugten DynamicProxy

Die programmtechnische Umsetzung der in Bild 4.4 dargestellten Situation erfolgt in drei Schritten:

1. Die Erzeugung der dynamischen Proxy-Klasse erfolgt durch Aufruf der statischen Methode getProxyClass der Klasse Proxy aus dem Reflection-Package:

```
package java.lang.reflect;

public class Proxy
{
    public static Class<?> getProxyClass(ClassLoader loader,
                                         Class<?>... interfaces)
           throws IllegalArgumentException
    {...}
    //weitere Methoden:
    ...
}
```

Als Parameter müssen ein ClassLoader sowie die zu implementierenden Schnittstellen in Form von Class-Objekten angegeben werden. Dass man einen ClassLoader angeben muss, wo doch gar keine Klasse geladen, sondern dynamisch erzeugt wird, mag merkwürdig erscheinen. Die Begründung dafür ist, dass eine Klasse nur zusammen mit ihrem ClassLoader-Objekt eindeutig ist. Sollte Ihnen diese Erläuterung unzureichend sein, so vertröste ich Sie auf das folgende Kapitel, in dem das Thema ausführlicher behandelt wird. Im nachfolgenden Beispiel wird ohne weitere Erklärung der SystemClassLoader verwendet, den man durch `ClassLoader.getSystemClassLoader()` bekommt. Falls die drei Punkte beim zweiten Parameter nicht geläufig sein sollten, sei erwähnt, dass es sich hierbei um das Varargs-Sprachkonstrukt von Java handelt. Es bedeutet, dass man beim Aufruf beliebig viele Class-Argumente oder alternativ ein Feld von Class-Objekten angeben kann (die weiter unten vorkommenden drei Punkte sind dagegen kein Java-Sprachkonstrukt, sondern sie sollen anzeigen, dass an dieser Stelle weiterer Programmcode folgt, der nicht angegeben ist). Die Methode getProxyClass liefert das Class-Objekt zurück, das die dynamisch erzeugte Proxy-Klasse repräsentiert.

2. Um ein Objekt der dynamisch erzeugten Klasse anzulegen, kann man nicht die Methode newInstance auf das von getProxyClass zurückgelieferte Class-Objekt anwenden, da diese Klasse keinen parameterlosen Konstruktor besitzt. Man benötigt einen Konstruktor, der einen Parameter des Typs InvocationHandler hat. Dieser Konstruktor lässt sich über Reflection leicht beschaffen.

3. Im letzten Schritt muss man dann auf den in Schritt 2 beschafften Konstruktor die Methode newInstance anwenden. Als Parameter übergibt man ein Object-Feld der Länge 1, das ein Objekt einer Klasse enthält, die die Schnittstelle InvocationHandler implementiert.

Da man in der Regel mit dem Class-Objekt für die dynamische Proxy-Klasse allein nichts anfangen kann, sondern immer ein Objekt davon braucht, bietet die Klasse Proxy eine weitere statische Methode an, die die drei Schritte zusammenfasst:

```
package java.lang.reflect;

public class Proxy
{
    public static Object newProxyInstance(Classloader cl,
```

```
                                          Class<?>[] interfaces,
                                          InvocationHandler ih)
            throws IllegalArgumentException
    {...}
    //weitere Methoden:
    ...
}
```

Die ersten beiden Parameter von newProxyInstance sind wie bei getProxyClass, der dritte Parameter ist das Argument für den Konstruktor der erzeugten dynamischen Proxy-Klasse. Bitte beachten Sie, dass bei dieser Methode für das zweite Argument das Varargs-Sprachkonstrukt nicht verwendet werden kann, da dies nur für den letzten Parameter möglich ist.

In Listing 4.3 wird ein dynamischer Proxy auf beide beschriebenen Arten erzeugt, einmal mit getProxyClass und einmal mit newProxyInstance.

Listing 4.3 Beispiel für mit Reflection dynamisch erzeugte Proxies

```
package javacomp.basics;

import java.lang.reflect.*;

class MyInvocationHandler implements InvocationHandler
{
    private Object target;

    public MyInvocationHandler(Object target)
    {
        this.target = target;
    }

    public Object invoke(Object proxy, Method method,
                         Object[] args)
    {
        String argsString = "";
        if(args != null)
        {
            for(int i = 0; i < args.length; i++)
            {
                if(i > 0)
                {
                    argsString += ", ";
                }
                argsString += args[i];
            }
        }
        System.out.println("---> " +
                           method.getDeclaringClass().getName() +
                           "." + method.getName() +
                           "(" + argsString + ")");
        try
        {
            Object result = method.invoke(target, args);
            if(method.getReturnType() != void.class)
            {
                System.out.println("<--- return: " + result);
            }
```

```
            else
            {
                System.out.println("<--- return");
            }
            return result;
        }
        catch(Throwable t)
        {
            System.out.println("<--- exception: " + t.getMessage());
            return null;
        }
    }
}

public class Main
{
    private static void useCounter(Counter counter)
    {
        //wie zuvor:
        ...
    }

    public static void main(String[] args)
    {
        CounterImpl counterImpl = new CounterImpl();
        try
        {
            Class<?>[] implInterfaces = new Class<?>[1];
            implInterfaces[0] = Counter.class;
            Class<?> dynProxyClass = Proxy.getProxyClass(
                                ClassLoader.getSystemClassLoader(),
                                implInterfaces);
            Class<?>[] formalArgs = new Class<?>[1];
            formalArgs[0] = InvocationHandler.class;
            Constructor<?> constructor =
                           dynProxyClass.getConstructor(formalArgs);
            Object[] actualArgs = new Object[1];
            actualArgs[0] = new MyInvocationHandler(counterImpl);
            Counter dynamicProxy =
                      (Counter) constructor.newInstance(actualArgs);
            useCounter(dynamicProxy);
        }
        catch(Exception e)
        {
            System.out.println("exception: " + e.getMessage());
        }

        System.out.println("-------------------------");

        try
        {
            Counter dynamicProxy = (Counter) Proxy.newProxyInstance(
                              ClassLoader.getSystemClassLoader(),
                              new Class<?>[] {Counter.class},
                              new MyInvocationHandler(counterImpl));
            useCounter(dynamicProxy);
        }
        catch(Exception e)
```

```
            {
                System.out.println("exception: " + e.getMessage());
            }
        }
    }
```

Die Ausgabe des Programms ist exakt dieselbe wie die beim statischen Proxy in Listing 4.2. Der wesentliche Unterschied besteht darin, dass der Code beim statischen Proxy genau auf die Schnittstelle Counter mit den Methoden add und sub sowie den Parametern dieser Methoden zugeschnitten war. Der Code in diesem letzten Beispiel ist allgemeiner: Die Klasse MyInvocationHandler lässt sich für jede beliebige Schnittstelle verwenden, und bei der Erzeugung des dynamischen Proxy muss man lediglich eine andere Schnittstelle in Form eines anderen Class-Objekts angeben. Mit anderen Worten: Wenn die Anwendung eine andere ist, so muss der statische Proxy aus Listing 4.2 neu programmiert werden, während genau der Code aus Listing 4.3 für den dynamischen Proxy unverändert weiterverwendet werden kann.

Ein Beispiel für eine Benutzung dieser Form dynamischer Proxies ist RMI ab Java-Version 5. Ein RMI-Stub ist ein Proxy, der stellvertretend auf der Client-Seite die Methodenaufrufe entgegennimmt, die für das ferne RMI-Objekt auf dem Server gedacht sind. Die Funktion des RMI-Stubs ist dann die Übertragung der Daten wie Methodenname und Parameter auf den Server über eine Netzverbindung. RMI-Stubs sind dynamisch erzeugte Proxies.

Ein weiteres Beispiel für die Verwendung dynamischer Proxies haben wir im vorigen Kapitel gesehen. Die Methoden getAnnotations, getDeclaredAnnotations und getAnnotation liefern Annotationsobjekte zurück, die die zu einer Annotation gehörenden Schnittstellen implementieren. Es wurde im Text gesagt, dass dies Objekte irgendwelcher nicht näher interessierender Klassen seien. Wie Sie jetzt vielleicht schon vermuten: Damit waren dynamische Proxy-Klassen gemeint.

4.3 Erzeugung dynamischer Proxies mit CGLIB

Wir kommen nochmals auf Listing 4.1 zurück. Wir verändern das Programm nun so, dass es die Schnittstelle Counter nicht mehr gibt, sondern nur noch die Klasse CounterImpl, die dann als Parametertyp in der Methode useCounter der Klasse Main verwendet werden muss. Wie kann für diese Situation ein Proxy realisiert werden? Die Antwort lautet: Die Proxy-Klasse muss in diesem Fall aus CounterImpl abgeleitet werden, dann kann das Proxy-Objekt wie das eigentliche Objekt benutzt werden. Man überschreibt in der abgeleiteten Proyx-Klasse die Methoden der Basisklasse und implementiert darin die Proxy-Funktionalität. Zum Delegieren der Methodenaufrufe benötigte das Proxy-Objekt in den bisherigen Beispielen eine Referenz auf das Server-Objekt. Dies ist in diesem Fall auch möglich, aber nicht unbedingt nötig, denn durch die Vererbung bringt die Proxy-Klasse die „eigentlichen“ Methoden mit, die zwar überschrieben, aber über einen Super-Aufruf immer noch zugänglich sind. Mit anderen Worten: Das Proxy-Objekt ist gleichzeitig auch das Server-Objekt.

Listing 4.4 zeigt eine statische Proxy-Klasse für unser modifiziertes Beispiel aus Listing 4.1 (d. h. nach Entfernen der Schnittstelle Counter). Darin sind die beschriebenen Überlegungen umgesetzt. Die Proxy-Klasse ist aus CounterImpl abgeleitet und benötigt kein Attribut, in dem eine Referenz auf das Server-Objekt gespeichert wird (entsprechend auch keinen expliziten Konstruktor). Die Methoden add und sub werden überschrieben. Die Delegation an die Methoden add und sub von CounterImpl erfolgt durch Aufruf von `super.add()` bzw. `super.sub()`.

Listing 4.4 Abgeleitete, statische Proxy-Klasse

```
package javacomp.basics;

class StaticProxyDerived extends CounterImpl
{
    public int add(int i)
    {
        System.out.println("---> javacomp.basics.CounterImpl.add(" +
                           i + ")");
        int result = super.add(i);
        System.out.println("<--- return: " + result);
        return result;
    }

    public int sub(int i)
    {
        System.out.println("---> javacomp.basics.CounterImpl.sub(" +
                           i + ")");
        int result = super.sub(i);
        System.out.println("<--- return: " + result);
        return result;
    }
}
```

Mit Hilfe der Reflection-Klasse Proxy, die im vorhergehenden Abschnitt vorgestellt und benutzt wurde, ist diese Idee nicht auf dynamische Proxies übertragbar. Wenn nämlich keine Schnittstelle vorhanden ist und man in Proxy.getProxyClass ein Class-Objekt übergibt, das eine Klasse und keine Schnittstelle repräsentiert, wird eine IllegalArgumentException mit der Meldung „is not an interface“ geworfen. Hier stoßen die Reflection-Klassen, die Teil des JDK (Java Development Kit) sind, an ihre Grenzen. Die CGLIB (Code Generation Library) hilft uns aber weiter. Mit Hilfe der CGLIB lässt sich tatsächlich eine Proxy-Klasse durch Vererbung aus einer vorgegebenen Klasse dynamisch erzeugen. Die CGLIB kann in Form einer Jar-Datei aus dem Internet geladen werden (unter dem Stichwort CGLIB kann die entsprechende Web-Seite leicht von jeder Suchmaschine gefunden werden). Bitte beachten Sie, dass Sie diese Jar-Datei dann in ihren CLASSPATH einbinden müssen (in Eclipse über den Menüeintrag „Build Path“).

Die wichtigste Klasse der CGLIB ist die Klasse Enhancer. Sie hat einen parameterlosen Konstruktor und eine parameterlose Methode create, die direkt ein Objekt der dynamischen Proxy-Klasse erzeugt (entspricht der Methode newProxyInstance des vorigen Abschnitts). Vor dem Aufruf von create kann durch die Methode setSuperclass die Klasse festgelegt werden, aus der die Proxy-Klasse abgeleitet werden soll. Es können mit setInterfaces auch mehrere Schnittstellen angegeben werden, die die dynamische Proxy-Klasse implementieren soll. Diese Möglichkeit werden wir im Folgenden nicht verwenden. Ähnlich wie in der

mit Reflection erzeugten dynamischen Proxy-Klasse werden in allen Methoden der Proxy-Klasse die Aufrufe delegiert. Das Objekt, an das delegiert wird, kann mit der Enhancer-Methode setCallback festgelegt werden, das die Schnittstelle MethodInterceptor implementieren sollte:

```
package net.sf.cglib.proxy

import java.lang.reflect.Method;

public interface MethodInterceptor extends Callback
{
    public Object intercept(Object proxy,
                            Method m,
                            Object[] args,
                            MethodProxy mProxy)
           throws Throwable;
}
```

Die ersten drei Argumente der Methode intercept entsprechen denjenigen der Methode invoke der Schnittstelle InvocationHandler aus der Reflection-Bibliothek (siehe Abschnitt 4.2). Der Typ des vierten Parameters ist MethodProxy. MethodProxy ist eine Klasse aus der CGLIB, die vergleichbar ist mit der Klasse Method aus der Reflection-Bibliothek. Zusätzlich zu der Methode invoke besitzt MethodProxy allerdings noch die Methode invokeSuper (mit denselben Argumenten wie invoke), mit der die überschriebene Methode der Basisklasse aufgerufen werden kann. Die Methode invokeSuper ist genau für den Fall vorgesehen, dass das MethodInterceptor-Objekt zum Weiterleiten des Methodenaufrufs an die „eigentliche“ Methode das dynamische Proxy-Objekt verwendet, da dieses die gewünschte Methode in überschriebener Form mitbringt (bei der statischen Proxy-Klasse in Listing 4.4 wurde die „eigentliche“ Methode auch über super aufgerufen). Der dynamische Proxy ist damit gleichzeitig auch wieder der Server (so wie zuvor der statische Proxy auch der Server war). Bild 4.5 veranschaulicht diese Situation.

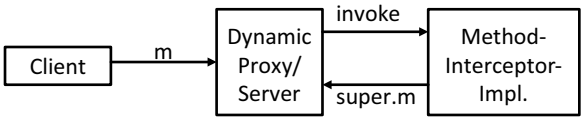

Bild 4.5 Strukturbild für einen mit der CGLIB erzeugten DynamicProxy

Selbstverständlich könnte man alternativ mit einem dynamischen Proxy, der mit Hilfe der CGLIB erzeugt wurde, auch das Szenario aus Bild 4.4 umsetzen. Für unser CounterImpl-Beispiel orientieren wir uns in Listing 4.5 aber an Bild 4.5.

Listing 4.5 Beispiel für einen mit der CGLIB dynamisch erzeugten Proxy

```
package javacomp.basics;

import java.lang.reflect.*;
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

class MyMethodInterceptor implements MethodInterceptor
{
    public Object intercept(Object target,
```

```
                            Method method,
                            Object[] args,
                            MethodProxy alternativeMethod)
           throws Throwable
    {
        String argsString = "";
        if(args != null)
        {
            for(int i = 0; i < args.length; i++)
            {
                if(i > 0)
                {
                    argsString += ", ";
                }
                argsString += args[i];
            }
        }
        System.out.println("---> " +
                           method.getDeclaringClass().getName() +
                           "." + method.getName() +
                           "(" + argsString + ")");
        try
        {
            Object result = alternativeMethod.invokeSuper(target,
                                                          args);
            if(method.getReturnType() != void.class)
            {
                System.out.println("<--- return: " + result);
            }
            else
            {
                System.out.println("<--- return");
            }
            return result;
        }
        catch(Throwable t)
        {
            System.out.println("<--- exception: " + t.getMessage());
            return null;
        }
    }
}

public class Main
{
    private static void useCounter(CounterImpl counter)
    {
        //wie zuvor:
        ...
    }

    public static void main(String[] args)
    {
        try
        {
            Enhancer enhancer = new Enhancer();
            enhancer.setSuperclass(CounterImpl.class);
            enhancer.setCallback(new MyMethodInterceptor());
```

```
            CounterImpl dynamicProxy =
                                    (CounterImpl) enhancer.create();
            useCounter(dynamicProxy);
        }
        catch(Exception e)
        {
            System.out.println("exception: " + e.getMessage());
        }
    }
}
```

Bitte beachten Sie die Anweisung `alternativeMethod.invokeSuper(target, args)` in der Methode intercept der Klasse MyMethodInterceptor. Der Parameter target ist eine Referenz auf den dynamischen Proxy. Auf diesem Objekt wird die überschriebene Methode der Basisklasse aufgerufen.

Noch ein ganz wichtiger Hinweis, falls Sie den Code aus Listing 4.5, den Sie auf der Web-Seite zum Buch finden, tatsächlich ausführen wollen: Sie müssen dazu die Sichtbarkeit der Klasse CounterImpl aus Listing 4.1 auf public ändern, was zur Folge hat, dass Sie CounterImpl in eine eigene Datei mit dem Namen CounterImpl.java auslagern müssen. Andernfalls wird beim Aufruf der Methode create auf dem Enhancer-Objekt die Ausnahme CodeGenerationException ausgelöst.

Logischerweise können Klassen, für die das Ableiten durch das Schlüsselwort final verboten ist, nicht als Basisklassen einer dynamischen Proxy-Klasse verwendet werden. Wenn man es versucht, wird die Ausnahme IllegalArgumentException beim Aufruf von create geworfen (mit der Meldung „Cannot subclass final class“). Kein Problem stellen jedoch Methoden der Basisklasse dar, die final sind. Diese werden einfach nicht überschrieben. Sie können das für unser CounterImpl-Beispiel leicht ausprobieren: Machen Sie z. B. in der Klasse CounterImpl die Methode add final, dann werden nur die Aufrufe von sub protokolliert, aber nicht mehr die von add, denn für sub wird die überschriebene Proxy-Methode aufgerufen, während add nicht überschrieben ist und somit die Methode add von CounterImpl direkt aufgerufen wird.

Wie eingangs des Kapitels schon erwähnt wurde, ist die aspektorientierte Programmierung ein Einsatzgebiet für dynamische Proxies. Bei der aspektorientierten Programmierung wird zusätzlicher Programmcode um Methoden „herumgewoben“. Dies wird mit Hilfe dynamischer Proxies realisiert, wobei sowohl Reflection als auch CGLIB zum Einsatz kommen.

5 ClassLoader und Hot Deployment

In der Unterhaltungssendung „Wetten dass..?“ hat eine Gruppe von Personen vor etlichen Jahren einmal gewettet, dass sie es schaffen, ein Rad eines Autos während der Fahrt zu wechseln. Diese in meiner Erinnerung erfolgreich verlaufene Wette stellt ein schönes Beispiel für Hot Deployment dar. Mit Hot Deployment ist gemeint, dass ein Teil eines Systems im laufenden Betrieb ausgetauscht wird.

Für viele komponentenbasierte Systeme ist Hot Deployment eine wesentliche Eigenschaft: Einzelne Software-Komponenten sollen gestartet, gestoppt und in einer eventuell anderen Version neu gestartet werden können, ohne dass man das Gesamtsystem dazu beenden und wieder hochfahren muss. Manchmal versteht man unter Hot Deployment lediglich das Einbringen neuer Software-Komponenten in ein laufendes System, während der Austausch einer bereits laufenden Komponente als Hot Redeployment oder Hot Update bezeichnet wird. In diesem Buch wird der Begriff Hot Deployment so verwendet, dass darin auch die Bedeutung von Hot Redeployment bzw. Hot Update mit eingeschlossen ist.

Um zu verstehen, wie man Hot Deployment in Java, das sicher weniger spektakulär ist als die Wette bei „Wetten dass..?“, realisieren kann, muss man ein paar Grundlagen über das Laden von Klassen kennen. Wir beginnen deshalb mit dem Thema Klassenladen in Java.

5.1 Klassenladen in Java

Als einleitendes Beispiel sehen wir uns das Programm aus Listing 5.1 an, das sicherlich ohne weitere Erläuterungen zu verstehen ist.

Listing 5.1 Beispielprogramm zur Erläuterung des normalen Klassenladens

```
package javacomp.basics;

class SomeClass
{
    public void call()
    {
        System.out.println("hallo");
    }
```

```
}

public class DynamicClassLoading
{
    public static void main(String[] args)
    {
        if(Math.random() <= 0.5)
        {
            for(int i = 1; i <= 30; i++)
            {
                SomeClass some = new SomeClass();
                some.call();
                try
                {
                    Thread.sleep(1000);
                }
                catch(InterruptedException e)
                {
                }
            }
        }
    }
}
```

Wenn die Datei DynamicClassLoading.java übersetzt wird, werden zwei Dateien erzeugt, für jede Klasse eine, nämlich DynamicClassLoading.class und SomeClass.class. Was passiert, wenn die Datei SomeClass.class gelöscht wird und dann das Programm mit dem Kommando

```
java javacomp.basics.DynamicClassLoading
```

gestartet wird? Wird in jedem Fall die Ausnahme ClassNotFoundException geworfen? Oder passiert das bei häufiger Ausführung nur in ca. 50 % der Fälle?

Wie Sie vielleicht schon vermuten, ist die zweite Antwortmöglichkeit die Richtige. In Java werden nicht alle für die Programmausführung potentiell benötigten Klassen zu Beginn geladen, sondern eine Klasse wird immer erst dann geladen, wenn sie zum ersten Mal gebraucht wird (man bezeichnet dieses Verhalten auch als Lazy Class Loading). Wenn also mit `Math.random()` eine Zufallszahl zwischen 0 und 1 bestimmt wird und wenn die Zahl größer als 0,5 ist (was in 50 % der Programmausführungen der Fall sein dürfte, da man von einer Gleichverteilung ausgehen kann), dann wird die Klasse SomeClass nicht benötigt, da der Anweisungsblock hinter dem if nicht ausgeführt wird. Das Fehlen der Klasse SomeClass fällt in diesen Fällen nicht auf; das Programm läuft somit ohne das Werfen einer Ausnahme zu Ende. Bei Interesse kann man sich übrigens anzeigen lassen, wann welche Klasse geladen wird. Dazu startet man das Programm mit der Option „-verbose“ (zeigt auch z. B. Aktionen des „Abfallsammelns“ [Garbage Collection]) oder „-verbose:class“ (zeigt nur das Laden von Klassen an):

```
java -verbose:class javacomp.basics.DynamicClassLoading
```

Ich empfehle Ihnen, das einmal auszuprobieren. Sie werden vielleicht überrascht sein, wie viele Meldungen Sie sehen werden. Bei der Ausführung des Programms aus Listing 5.1, das aus nur zwei Klassen besteht, werden mehr als 300 Klassen geladen.

Anhand des Programms aus Listing 5.1 soll ein weiterer, Ihnen vermutlich ebenfalls bekannter Effekt beschrieben werden. Dazu gehen wir diese Mal vom Normalfall aus, dass nämlich alle benötigten Class-Dateien vorhanden sind. Nehmen wir nun an, dass die Zufallszahl kleiner als 0,5 ist und der Anweisungsblock mit der For-Schleife ausgeführt wird. Dies dauert ca. 30 Sekunden, da das Programm bei jedem Schleifendurchlauf eine Zeile ausgibt und dann eine Sekunde pausiert. Wenn Sie nun in diesen 30 Sekunden die Methode call in der Klasse SomeClass verändern (z. B. die Ausgabe von „hallo“ in „huhu“ ändern), anschließend neu übersetzen und dadurch die Datei SomeClass.class durch eine andere Version ersetzt wird, werden Sie keinen Effekt beim laufenden Programm wahrnehmen, denn die Klasse SomeClass wird natürlich nicht für jeden Schleifendurchlauf neu geladen, sondern eben nur einmal bei der ersten Benutzung. Es ist allerdings doch möglich, ein und dieselbe Klasse während der Ausführung eines Programms mehrfach – auch in mehreren Versionen – zu laden. Dazu benötigt man aber unterschiedliche ClassLoader-Objekte. Wir kümmern uns im nächsten Abschnitt deshalb um ClassLoader.

5.2 ClassLoader

Bei der Ausführung eines Java-Programms gibt es in der Regel mehrere ClassLoader-Objekte. Ein ClassLoader-Objekt kann eine Klasse nur einmal laden. Man kann aber weitere ClassLoader-Objekte erzeugen, die eine bestimmte Klasse jeweils einmal laden. Insgesamt kann diese Klasse dadurch mehrfach geladen werden.

Die Basis des Klassenladens ist die abstrakte Klasse ClassLoader im Package java.lang. Die entscheidende Methode ist loadClass:

```
public Class<?> loadClass(String name) throws ClassNotFoundException
```

Der String-Parameter muss dabei ein vollständig spezifizierter Klassenname (also einschließlich des Package-Namens) sein.

Aus der abstrakten ClassLoader-Klasse sind mehrere Klassen abgeleitet worden (SecureClassLoader, URLClassLoader). Es ist auch möglich, eine eigene ClassLoader-Klasse zu schreiben. Von dieser Möglichkeit wird aber in diesem Buch kein Gebrauch gemacht. Alle ClassLoader-Objekte (gleich von welcher ClassLoader-Klasse sie sind) sind baumartig vernetzt. Die Standard-ClassLoader-Klassen verhalten sich so, wenn ein ClassLoader-Objekt aufgefordert wird, eine Klasse zu laden:

- Zunächst wird geprüft, ob diese Klasse bereits vorher von diesem ClassLoader-Objekt geladen wurde.
- Falls ja, erfolgt keine weitere Aktion.
- Falls nein, wird die Anfrage an den Vorgänger-ClassLoader im Baum weitergeleitet.
- Der Vorgänger-ClassLoader handelt auch so. Das heißt, wenn dieser ClassLoader die Klasse schon geladen hat, ist der komplette Ladevorgang beendet. Andernfalls wird die Anfrage an den Vorgänger weitergereicht, sofern es einen Vorgänger gibt (sofern also die Wurzel noch nicht erreicht wurde). Die Aufforderung wird also bis zur Wurzel weitergereicht. Der Wurzel-Loader bekommt somit als Erster die Chance zum Laden der Klasse.

- Falls der Wurzel-Server die Klasse lädt, ist die Aktion damit abgeschlossen. Andernfalls bekommt der Nachfolger, von dem der Wurzel-Server kontaktiert wurde, die Chance des Ladens.
- So geht es dann weiter, bis einer der Vorgänger die Klasse geladen hat oder die Anforderung wieder zu dem ClassLoader-Objekt zurückkehrt, das ursprünglich damit beauftragt wurde.
- Dieser ClassLoader hat dann erst die Möglichkeit zum Laden der Klasse. Wenn auch er die Klasse nicht laden kann, ist der Vorgang fehlgeschlagen, andernfalls erfolgreich abgeschlossen.

Das Hochreichen der Aufforderung zum Klassenladen bis zur Wurzel dient dem Schutz der Java-Kernklassen, denn somit kann ein eigener ClassLoader keine Klasse wie z. B. java.lang.String laden.

Der oben geschilderte Ablauf bedeutet, dass sich die Klasse, die man durch ein eigenes ClassLoader-Objekt laden möchte, nicht im CLASSPATH befinden darf, denn sonst wird sie spätestens durch den Wurzel-ClassLoader geladen. Mit anderen Worten: Man muss allen anderen ClassLoader-Objekten die Möglichkeit nehmen, die Klasse laden zu können.

Eine weitere Konsequenz dieser Strategie des Klassenladens ist, dass von einer Klasse A1, die von einem ClassLoader cl1 geladen wurde, nicht auf eine Klasse A2, die von einem anderen ClassLoader cl2 geladen wurde, zugegriffen werden kann, falls cl2 kein direkter oder indirekter Vorgänger von cl1 im ClassLoader-Baum ist. Sollten Sie diese Konsequenz im Moment nicht nachvollziehen können, so spielt dies erst einmal keine Rolle. Wir werden auf diese Problematik bei der prototypischen Implementierung eines eigenen Komponentensystems nochmals zurückkommen. Anhand eines konkreten Beispiels wird der Sachverhalt dann einfacher zu verstehen sein.

Um ein eigenes ClassLoader-Objekt zu erzeugen, braucht man nicht unbedingt eine eigene ClassLoader-Klasse zu implementieren, sondern man kann auf die Klasse URLClassLoader zurückgreifen. Diese hat u. a. die folgenden Konstruktoren:

```
public URLClassLoader(URL[] urls, ClassLoader parent) {...}
public URLClassLoader(URL[] urls) {...}
```

Das Argument urls enthält eine Folge von Verzeichnissen und Jar- bzw. Zip-Dateien, die der Reihe nach durchsucht werden, um die zu ladende Klasse zu finden (wie der Datentyp URL andeutet, können sich diese auch auf anderen Rechnern befinden). Der zweite Parameter des ersten Konstruktors gibt die Stelle an, wo das neue ClassLoader-Objekt im Baum aller ClassLoader angehängt werden soll. Im zweiten Konstruktor entfällt diese Angabe. In diesem Fall wird ein Standard-System-Klassenlader als Vorgänger des neuen ClassLoaders verwendet.

Wenn im Code einer Klasse A die Klasse B benötigt wird, so wird bei automatischem Laden der Klassenlader, der A geladen hat, beauftragt auch B zu laden. Man kann aber das Laden von Klassen auch explizit durch Aufruf von loadClass auf ein spezifisches ClassLoader-Objekt anstoßen.

5.3 Beispiele

5.3.1 Hot Deployment

Als Beispiel für ein Hot Deployment wollen wir das Programm aus Listing 5.1 so verändern, dass man den Code einer Methode, die immer wieder aufgerufen wird, verändern kann und diese Änderung dann direkt im laufenden Programm sieht. Der Typ der lokalen Variablen, die auf das Objekt zeigt, dessen Klasse X aktualisiert werden soll, sollte nicht X sein, da X – wie oben erläutert – sich nicht im CLASSPATH befinden darf. Deshalb führen wir eine Schnittstelle ein für den Typ der lokalen Variablen. Die aus der Übersetzung resultierende Class-Datei sollte sich im CLASSPATH befinden. Die Schnittstelle heißt HotDeployment-Interface und ist in Listing 5.2 zu sehen.

Listing 5.2 Schnittstelle HotDeploymentInterface

```
package javacomp.basics;

public interface HotDeploymentInterface
{
    public void call();
}
```

Die Klasse, welche die Schnittstelle HotDeploymentInterface implementiert, ist HotDeploy-mentClass1 (s. Listing 5.3).

Listing 5.3 Klasse HotDeploymentClass1

```
package javacomp.basics;

public class HotDeploymentClass1 implements HotDeploymentInterface
{
    public void call()
    {
        System.out.println("HotDeploymentClass1: hallo");
        new HotDeploymentClass2().call();
    }
}
```

In der Methode call wird nicht nur etwas ausgegeben, wobei die Ausgabe im laufenden Betrieb geändert werden soll, sondern es wird eine weitere Klasse namens HotDeployment-Class2 benutzt, die auch eine Call-Methode hat mit einer Bildschirmausgabe. Der Sinn dieser zweiten Klasse wird später hoffentlich noch deutlich werden.

Listing 5.4 Klasse HotDeploymentClass2

```
package javacomp.basics;

public class HotDeploymentClass2 implements HotDeploymentInterface
{
    public void call()
    {
        System.out.println("HotDeploymentClass2: hallo");
```

```
    }
}
```

Der interessanteste und wichtigste Teil befindet sich in der Main-Methode der Klasse HotDeployment (s. Listing 5.5). Dort wird in einer Endlosschleife in jedem Schleifendurchlauf ein neues ClassLoader-Objekt des Typs URLClassLoader erzeugt. Dieses ClassLoader-Objekt wird benutzt, um die Klasse HotDeploymentClass1 zu laden. Dann wird ein Objekt dieser Klasse erzeugt (wir wissen ja, dass die Klasse HotDeploymentClass1 einen parameterlosen Konstruktor besitzt) und auf dieses Objekt die Methode call angewendet. Zum Erzeugen eines URLClassLoader-Objekts wird ein URL-Feld benötigt. In Listing 5.5 wird ein Feld der Länge 1 benutzt, welches ein URL-Objekt enthält, welches das Verzeichnis „hotdeployment“ repräsentiert.

Listing 5.5 Klasse HotDeployment zur Realisierung des Hot Deployment

```
package javacomp.basics;

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

public class HotDeployment
{
    public static void main(String[] args)
    {
        while(true)
        {
            try
            {
                URL url =
                         new File("hotdeployment/").toURI().toURL();
                URLClassLoader cl =
                                 new URLClassLoader(new URL[]{url});
                Class<?> newClassVersion = cl.loadClass(
                             "javacomp.basics.HotDeploymentClass1");
                HotDeploymentInterface obj;
                obj = (HotDeploymentInterface)
                                        newClassVersion.newInstance();
                obj.call();
                Thread.sleep(3000);
            }
            catch(Exception e)
            {
                System.err.println(e);
            }
        }
    }
}
```

Wenn wir von einer Eclipse-Umgebung ausgehen, dann befinden sich in dem Verzeichnis eines Java-Projekts ein Verzeichnis src für den Quellcode und ein Verzeichnis bin für die übersetzten Class-Dateien. Das zuletzt genannte Verzeichnis gehört zum CLASSPATH. Um den Effekt des Hot Deployment erfolgreich zu demonstrieren, dürfen sich die Class-Dateien für die Klassen HotDeploymentClass1 und HotDeploymentClass2 nicht innerhalb von bin befinden. Jedoch muss im Projektverzeichnis neben src und bin ein neues Verzeichnis mit

dem Namen hotdeployment angelegt werden. Dahin müssen die Class-Dateien für die Klassen HotDeploymentClass1 und HotDeploymentClass2 verschoben werden. Aber bitte beachten Sie, dass sich in diesem Verzeichnis – wie auch in bin – eine Verzeichnisstruktur befinden muss, die der Package-Struktur entspricht. Für den vorliegenden Fall heißt das, dass in hotdeployment ein Verzeichnis namens javacomp, darin ein Verzeichnis basics und darin die Class-Dateien sein müssen.

Wenn diese Struktur hergestellt wurde, kann das obige Programm ausgeführt werden. Nun kann im laufenden Betrieb die Ausgabe in der Methode call der Klasse HotDeploymentClass1 geändert werden. Anschließend muss logischerweise die aus der Übersetzung resultierende Class-Datei wieder unter hotdeployment abgespeichert werden. Dann wird beim nächsten Schleifendurchlauf diese neue Klassenversion geladen und man sieht die veränderte Ausgabe, ohne dass die Programmausführung zwischendurch gestoppt wurde.

Interessanterweise funktioniert dies auch für die Klasse HotDeploymentClass2. Das heißt: Wenn man die Ausgabe in der Methode call der Klasse HotDeploymentClass2 ändert, dann sieht man diese Änderung ebenfalls im laufenden Betrieb, obwohl diese Klasse nicht explizit durch den neuen ClassLoader geladen wird. Das wird sie aber implizit doch wegen des am Ende des vorigen Abschnitts geschilderten Sachverhalt: Da HotDeploymentClass2 in HotDeploymentClass1 benutzt wird, wird HotDeploymentClass2 von demselben ClassLoader geladen, der HotDeploymentClass1 geladen hat. Egal, ob man nur eine der beiden oder beide Klassen ändert; man sieht den Effekt beim nächsten Schleifendurchlauf. Wenn die Klassen allerdings zwischendurch in den CLASSPATH geraten und von dort geladen werden, dann ist ab diesem Zeitpunkt kein Laden einer neuen Version mehr möglich, selbst wenn man die Class-Dateien wieder aus dem CLASSPATH nimmt und im Verzeichnis hotdeployment eine neue Version ablegt. Denn bei jedem Laden über einen neuen ClassLoader werden zuerst die übergeordneten ClassLoader aufgefordert, die Klasse zu laden. Da einer von diesen die Klasse zuvor schon erfolgreich geladen hat, meldet dieser ClassLoader Erfolg und das Laden ist beendet; auf die neue Klassenversion erfolgt kein Zugriff mehr.

5.3.2 Gleichzeitige Nutzung mehrerer Versionen einer Klasse

In dem vorhergehenden Beispiel wurde immer nur die jeweils aktuellste Version der beiden Klassen verwendet. Das Objekt, das im vorhergehenden Schleifendurchlauf zu einer eventuell anderen Klassenversion erzeugt wurde, wird vergessen und ist im nächsten Schleifendurchlauf nicht mehr verfügbar. In einem weiteren Beispielprogramm soll deshalb demonstriert werden, dass es gleichzeitig mehrere Objekte unterschiedlicher Klassenversionen derselben Klasse geben kann. Dieses Beispiel geht über die Demonstration für Hot Deployment hinaus. Es soll ergänzend der Vorstellung entgegenwirken, dass es zu einem Zeitpunkt immer nur eine Version einer Klasse in einem ausgeführten Programm geben könnte. Es ist vielmehr so, dass eine Klasse eindeutig durch ihren vollständigen Klassennamen und das ClassLoader-Objekt, das sie geladen hat, identifiziert wird. Wenn also eine Klasse mit einem bestimmten Namen von unterschiedlichen ClassLoader-Objekten geladen wird, dann sind das für die JVM (Java Virtual Machine) unterschiedliche Klassen. Dabei kommt es nicht darauf an, ob diese unterschiedlichen Klassen, die alle denselben Klassennamen haben, tatsächlich unterschiedlich sind bezüglich des Programmcodes bzw. der Class-Datei oder

identisch. Dieser Sachverhalt soll mit Hilfe statischer Attribute durch das folgende Programm demonstriert werden.

Bild 5.1 zeigt das Fenster des Demo-Programms unmittelbar nach dem Start. Im oberen Teil sieht man ein Label für Meldungen, das zu Beginn leer ist, und im unteren Teil einen Button mit der Beschriftung „Neue Aktion!“, den man klicken kann.

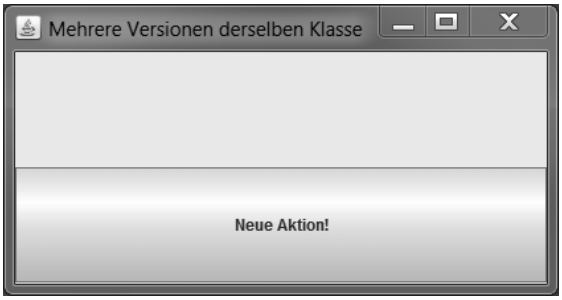

Bild 5.1 Demo-Programm zur gleichzeitigen Nutzung mehrerer Versionen einer Klasse (Zeitpunkt 1)

Bei jedem Klick auf den Button „Neue Aktion!“ wird dem Fenster ein neuer Button mit der Beschriftung „Aktion 1!“, „Aktion 2!“, „Aktion 3!“ usw. hinzugefügt. An jeden dieser Buttons wird ein Listener angemeldet, in dessen Konstruktor ein neues Objekt des Typs MultiVersionClass erzeugt wird. Beim Drücken auf den Button wird die Methode newMessage aufgerufen und der zurückgegebene String in das Label des Fensters geschrieben. In Listing 5.6 sehen wir die Version der Klasse MultiVersionClass zu dem Zeitpunkt, als der Button „Neue Aktion!“ zum ersten Mal geklickt wurde.

Listing 5.6 Klasse MultiVersionClass

```
package javacomp.basics;

public class MultiVersionClass
{
    private static int staticCounter;
    private static String version = "Version 1";

    private int counter;

    public String newMessage()
    {
        staticCounter++;
        counter++;
        return "counter=" + counter +
               ", staticCounter=" + staticCounter +
               ", version=" + version;
    }
}
```

In Bild 5.2 sehen wir, wie das Fenster aussieht, nachdem der Button „Neue Aktion!“ einmal und der dadurch erscheinende Button „Aktion 1!“ vier Mal geklickt wurde. Die Ausgabe, die im Label zu sehen ist, sollte keine Verwunderung auslösen. Durch viermaliges Klicken auf „Aktion 1!“ wurde die Methode newMessage vier Mal aufgerufen. Entsprechend wurde sowohl das Attribut counter als auch das statische Attribut staticCounter vier Mal erhöht

und beide haben somit den Wert 4. Das statische Attribut version hat den nicht veränderlichen Wert „Version 1“.

Bild 5.2 Demo-Programm zur gleichzeitigen Nutzung mehrerer Versionen einer Klasse (Zeitpunkt 2)

Die Klasse, von welcher die an den Buttons angemeldeten Listener stammen, wird über einen eigenen ClassLoader wie in Listing 5.5 geladen. Da im Konstruktor dieser Listener-Klasse ein Objekt der Klasse MultiVersionClass (s. Listing 5.6) erzeugt wird, wird auch die Klasse MultiVersionClass über diesen Listener geladen, falls sie nicht im CLASSPATH ist (das ist sehr ähnlich wie bei den Klassen HotDeploymentClass1 und HotDeploymentClass2 im vorhergehenden Beispiel). Bild 5.3 zeigt die Situation, nachdem noch weitere drei Mal auf „Neue Aktion!“ geklickt wurde. Vor jedem Klicken auf „Neue Aktion!“ wurde die Klasse MultiVersionClass so geändert, dass das statische Attribut version auf „Version 2“, „Version 3“ und „Version 4“ gesetzt wurde. Das heißt, zu jedem der Buttons „Aktion 1!“, „Aktion 2!“ usw. gibt es ein Objekt einer neuen Version der Listener-Klasse, das jeweils eine Referenz auf ein Objekt einer neuen Version der Klasse MultiVersionClass besitzt.

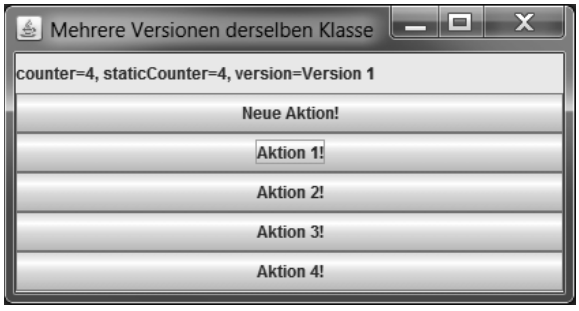

Bild 5.3 Demo-Programm zur gleichzeitigen Nutzung mehrerer Versionen einer Klasse (Zeitpunkt 3)

In Bild 5.4 sieht man die Situation nach achtmaligem Klicken auf „Aktion 2!“.

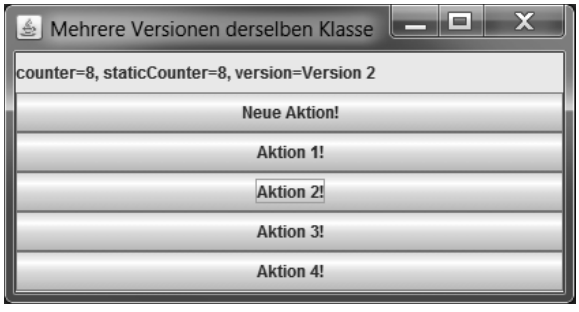

Bild 5.4 Demo-Programm zur gleichzeitigen Nutzung mehrerer Versionen einer Klasse (Zeitpunkt 4)

Und Bild 5.5 zeigt, wie das Fenster aussieht, wenn man zwei Mal auf „Aktion 4!“ geklickt hat.

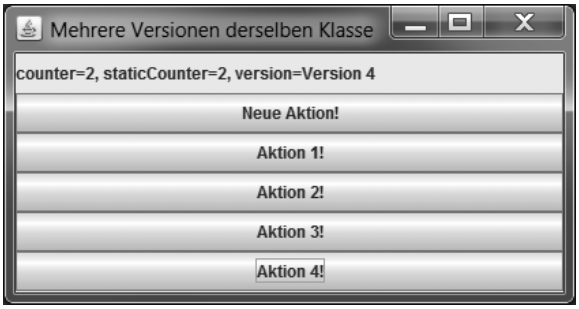

Bild 5.5 Demo-Programm zur gleichzeitigen Nutzung mehrerer Versionen einer Klasse (Zeitpunkt 5)

Wenn man anschließend wieder auf „Aktion 1“ klickt, sieht man die in Bild 5.6 gezeigte Situation.

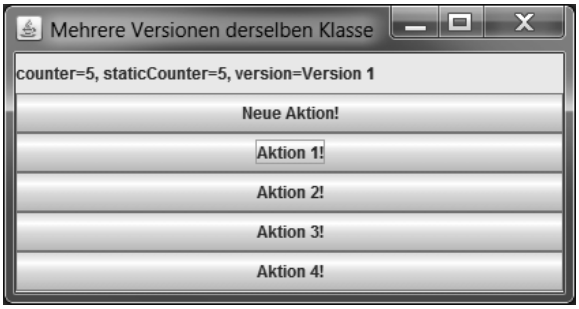

Bild 5.6 Demo-Programm zur gleichzeitigen Nutzung mehrerer Versionen einer Klasse (Zeitpunkt 6)

Zu jeder Klassenversion gibt es ein einziges Objekt. Zu jedem Objekt gibt es eine Instanz des Attributs counter. Wie in den Ausgaben zu sehen ist, hat das statische Attribut staticCounter immer denselben Wert wie das Attribut counter. Das heißt, dass es so viele Instanzen des statischen Attributs staticCounter gibt wie Objekte. Man sieht dies zusätzlich an dem anderen statischen Attribut version. Am Anfang der Java-Ausbildung lernt man, dass es zu einem statischen Attribut einer Klasse genau eine Instanz gibt. Wir haben in diesem Beispiel gesehen, dass dies nicht die volle Wahrheit ist. In dem ausgeführten Programm existieren gleichzeitig mehrere Objekte unterschiedlicher Versionen derselben Klasse. Für jede Klassenversion gibt es die statischen Attribute je ein Mal. Alle Klassenversionen existieren gleichzeitig im ausgeführten Programm. Das ist genau der Sachverhalt, der durch dieses Programm demonstriert werden sollte.

Der Programmcode für dieses Beispiel birgt keine weiteren Besonderheiten, so dass der Platz für den Abdruck des Codes eingespart werden soll. Das vollständige Programm finden Sie bei Interesse wieder auf der Web-Seite zum Buch.

II Teil 2: Java-Komponenten

Basierend auf den notwendigen Java-Grundlagen, die im ersten Teil dieses Buches behandelt wurden, befasst sich der zweite Teil mit dem Kernthema dieses Buches, nämlich Java-Komponenten. Zum Einstieg wird in Kapitel 6 ein einfaches Komponentensystem prototypisch entwickelt, das ein erste Vorstellung vermitteln soll, wie Komponenten und ein Komponenten-Framework aussehen könnten. Auf dieser Grundlage wird dann im folgenden Kapitel 7 in allgemeiner Form ausgelotet, was unter den Begriffen Komponenten und Komponentensystem in diesem Buch verstanden werden soll.

6 Prototypische Implementierung eines Komponentensystems

In diesem Kapitel wird ein Komponentensystem (auch Komponenten-Framework) mit einigen Beispielkomponenten entwickelt. Das Framework bildet eine Art von Infrastruktur oder Laufzeitumgebung für die Komponenten. Es wird deutlich werden, dass die Komponenten und das Framework miteinander interagieren: Sowohl das Framework ruft Methoden einer Komponente als auch eine Komponente Methoden des Frameworks auf. Zudem können die Komponenten untereinander zusammenarbeiten. Diese Sachverhalte sind in Bild 6.1 zusammengefasst.

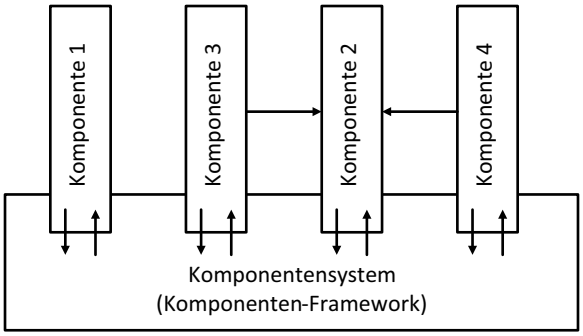

Bild 6.1 Komponentensystem (Framework) mit Komponenten

Für den Prototyp spielen die im ersten Teil des Buchs behandelten Grundlagen zu Reflection, Annotationen, ClassLoading und Hot Deployment eine bedeutsame Rolle.

Das nun im Folgenden vorgestellte Komponentensystem ist an OSGi angelehnt. Einige Sachverhalte werden hier relativ ausführlich erklärt. Dafür können einige Erläuterungen im dritten Teil des Buchs, in dem mehrere „real existierende“ Frameworks vorgestellt werden, entfallen.

6.1 Beispielkomponenten

Eine Komponente für unser prototypisches Framework muss in eine Jar- oder Zip-Datei gepackt werden. Die gepackte Datei ist einer Jar-Datei, mit der durch einen Doppelklick eine eigenständige Java-Anwendung gestartet werden kann, sehr ähnlich. Zum einen besteht die Komponentendatei aus einer Reihe von Klassen. Für unser Komponentensystem müssen sich diese alle in einem Verzeichnis namens classes innerhalb der Jar- bzw. Zip-Datei befinden. Das heißt, dass im Verzeichnis classes der Pfad, der den Package-Namen entspricht, beginnt. Ähnlich wie bei einer ausführbaren Jar-Datei muss auch für unsere Komponente festgelegt werden, welche Klasse die Einstiegsklasse ist. Zu diesem Zweck befindet sich auch in unserer Komponentendatei eine Datei MANIFEST.MF in einem Verzeichnis namens META-INF. In dieser Datei wird die Einstiegsklasse durch eine Zeile, die mit „Main:“ beginnt, spezifiziert. Das Framework erzeugt ein Objekt dieser Einstiegsklasse und ruft unter Umständen mehrere Start-Methoden auf diesem Objekt auf, wie im Folgenden noch genauer beschrieben werden wird.

6.1.1 Komponente Nr. 1

Im Package javacomp.prototype.application1 befindet sich unsere erste Beispielkomponente. Diese besteht aus einer einzigen Klasse namens MainClass, die auch die Einstiegsklasse ist. Folglich sieht die Datei MANIFEST.MF so aus:

```
Main:  javacomp.prototype.application1.MainClass
```

Die Komponentendatei, die eine Zip-Datei sein und deren Dateiname mit „.jar“ oder „.zip“ enden muss, hat die in Bild 6.2 gezeigte Struktur.

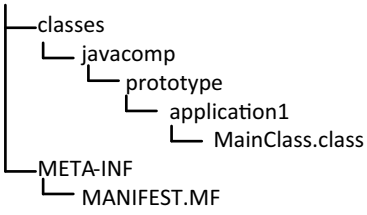

Bild 6.2 Struktur der Jar- bzw. Zip-Datei für die erste Beispielkomponente

Die Einstiegsklasse für unser prototypisches Komponentensystem muss weder eine vom Framework vorgegebene Schnittstelle implementieren noch aus einer vorgegebenen Klasse abgeleitet werden. Sie ist somit eine POJO-Klasse (POJO: Plain Old Java Object). Es ist lediglich notwendig, dass sie einen öffentlichen, parameterlosen Konstruktor besitzt. Das ist alles, was das Framework von seinen Komponenten verlangt. Optional können in der Einstiegsklasse eine oder mehrere Methoden mit der vom Framework definierten Annotation @Start gekennzeichnet werden. Falls diese Methoden öffentlich sind, keine Parameter und den Rückgabetyp void haben, werden sie nach der Erzeugung eines Objekts der Einstiegsklasse aufgerufen. Zum Installieren (Deployment) einer Komponente in das Framework muss die entsprechende Jar- oder Zip-Datei einfach in ein spezielles Verzeichnis kopiert

werden. Analog wird beim Löschen dieser Datei die Komponente wieder deinstalliert (Undeployment). Dies bedeutet, dass auf dem zu Beginn erzeugten Objekt der Einstiegsklasse nun alle öffentlichen, parameterlosen Methoden mit Rückgabetyp void aufgerufen werden, die durch @Stop annotiert sind. Wird die Jar- oder Zip-Datei verändert, so ist der Effekt so, als wäre sie gelöscht und neu angelegt werden (d. h. es erfolgt eine Deinstallation mit einer anschließenden Neuinstallation).

Zur Demonstration des beschriebenen Verhaltens betrachten wir die Klasse MainClass aus Listing 6.1.

Listing 6.1 Klasse MainClass der ersten Beispielkomponente

```
package javacomp.prototype.application1;

import javacomp.prototype.framework.Start;
import javacomp.prototype.framework.Stop;

public class MainClass
{
    public MainClass()
    {
        System.out.println("hello from application1 (version 1)");
    }

    @Start
    public void start1()
    {
        System.out.println("app1.start1()");
    }

    public void start2()
    {
        System.out.println("app1.start2()");
    }

    @Start
    public int start3()
    {
        System.out.println("app1.start3()");
        return 0;
    }

    @Start
    public void start4(int i)
    {
        System.out.println("app1.start4(int)");
    }

    @Start
    protected void start5()
    {
        System.out.println("app1.start5()");
    }

    @Start
    public void start6()
```

```
    {
        System.out.println("app1.start6()");
    }

    @Stop
    public void stop1()
    {
        System.out.println("app1.stop1()");
    }

    @Stop
    public int stop2()
    {
        System.out.println("app1.stop2()");
        return 0;
    }

    @Stop
    public void stop3()
    {
        System.out.println("app1.stop3()");
    }
}
```

Bei der Installation der Komponente werden folgende Zeilen ausgegeben:

```
hello from application1 (version 1)
app1.start1()
app1.start6()
```

Wie der Ausgabe zu entnehmen ist, werden nach der Erzeugung eines Objekts der Klasse MainClass die Methoden start1 und start6 aufgerufen. Die Methode start2 wird nicht aufgerufen, weil sie nicht annotiert ist, start3 hat nicht den Rückgabetyp void, start4 ist nicht parameterlos und start5 ist nicht öffentlich.

Entsprechend sollte klar sein, dass nach der Entfernung der Jar- bzw. Zip-Datei Folgendes zu sehen ist:

```
app1.stop3()
app1.stop1()
```

Das Beispiel soll u. a. auch zeigen, dass sowohl beim Starten als auch beim Stoppen mehrere Methoden aufgerufen werden können. Eine Komponentenentwicklerin sollte dabei allerdings nicht von einer festen Reihenfolge ausgehen (z. B. von der Reihenfolge der Methoden im Quelltext).

Unser prototypisches Komponenten-Framework unterstützt übrigens Hot Deployment. Wenn man also die Klasse MainClass verändert (z. B. die Ausgabe im Konstruktor von „Version1“ in „Version 2“ ändert), dann ist beim Neuinstallieren (Redeployment) unserer Beispielkomponente die neue Meldung zu sehen, ohne dass das Framework zwischenzeitlich neu gestartet wurde. Bitte erinnern Sie sich in diesem Zusammenhang aber an die Ausführungen im Kapitel über Hot Deployment: Das Laden der neuen Klassenversion funktioniert nur dann, wenn sich die Klasse MainClass nicht auch noch im Klassenpfad des Komponenten-Frameworks befindet, sondern ausschließlich in der Jar- bzw. Zip-Datei.

Noch ein Hinweis: Das Framework besitzt nur einen einzigen Thread, der die Aufrufe aller Start- und Stop-Methoden aller Komponenten durchführt. Deshalb sollten die Start- und Stop-Methoden keine allzu lange Ausführungszeit haben, da ansonsten die Funktionsfähigkeit unseres Komponentensystems gebremst wird. Im Extremfall würde eine Start- oder Stop-Methode mit einer Endlosschleife unser ganzes Komponenten-Framework lahmlegen (d. h. es würde nicht mehr auf Änderungen im Installationsverzeichnis reagieren und somit weder neue Komponenten in das Framework aufnehmen noch die Stop-Methoden zu deinstallierender Komponenten aufrufen). Wir gehen also davon aus, dass alle Komponentenentwickler so diszipliniert sind, dass sie ihre Start- und Stop-Methoden entsprechend gestalten. Diese Situation ist für Java-Entwickler übrigens nichts Neues: Auch bei der Programmierung grafischer Benutzeroberflächen kann man eine Anwendung außer Gefecht setzen, wenn man in eine Listener-Methode (die z. B. auf das Klicken eines Buttons reagiert) eine Endlosschleife einbaut. Wie bei den grafischen Benutzeroberflächen müsste man auch in unserem Komponentensystem in dem Fall, dass ein etwas länger dauernde Aktion durchgeführt werden muss, diese Aktion in einen separaten Thread auslagern. Wir werden ein Beispiel dafür in einer der noch folgenden Komponenten sehen.

6.1.2 Komponente Nr. 2

Ähnlich wie die erste Beispielkomponente könnten nun weitere Komponenten entwickelt werden, die unabhängig voneinander im Framework installiert, neuinstalliert und deinstalliert werden. Dies bringt nichts wesentlich Neues. Eine neue Herausforderung stellt sich jedoch, wenn man Komponenten zusammenarbeiten lassen möchte, wenn also z. B. eine Komponente ein Objekt benutzen möchte, das von einer anderen Komponente bereitgestellt wird. Um dies zu unterstützen, bietet unser Framework eine Registratur an, bei der Objekte unter einem Namen an- und abgemeldet werden können. Außerdem kann man durch Angabe des passenden Namens eine Referenz auf ein angemeldetes Objekt erhalten. Die Methoden dazu befinden sich in der Schnittstelle ComponentContext (s. Listing 6.2), welche das Framework zur Verfügung stellt. Ihre Bedeutung sollte ohne weitere Erläuterung verständlich sein.

Listing 6.2 Schnittstelle ComponentContext

```
package javacomp.prototype.framework;

public interface ComponentContext
{
    public void bind(String name, Object obj);
    public Object lookup(String name);
    public void unbind(String name);
}
```

Damit die Komponenten diesen vom Framework bereitgestellten Registraturdienst benutzen können, werden nicht nur die mit @Start und @Stop annotierten, öffentlichen Void-Methoden vom Framework aufgerufen, die parameterlos sind, sondern auch solche, die einen einzigen Parameter des Typs ComponentContext haben. Eine Service-Komponente, welche einen Dienst bereitstellt, kann in ihrer Start-Methode über den ComponentContext-Parameter ein Objekt an- und ihrer Stop-Methode wieder abmelden. Eine Client-Komponente, welche den

Dienst der Service-Komponente nutzen möchte, kann sich in ihrer Start-Methode ebenfalls über den ComponentContext-Parameter den Zugriff auf das entsprechende Objekt beschaffen. Dazu muss sie allerdings den Namen kennen, unter dem das Objekt registriert wurde.

Unsere zweite Beispielkomponente ist eine solche Service-Komponente. Sie besteht aus den Klassen Counter und CounterService. Die Klasse Counter (s. Listing 6.3) ist sehr einfach und braucht keine weiteren Erläuterungen (bis auf synchronized, worauf später eingegangen wird).

Listing 6.3 Klasse Counter

```
package javacomp.prototype.application2;

public class Counter
{
    private int counter;

    public Counter()
    {
    }

    public Counter(int initialValue)
    {
        counter = initialValue;
    }

    public synchronized int increment()
    {
        counter++;
        return counter;
    }
}
```

Die Klasse CounterService (s. Listing 6.4) ist die Einstiegsklasse (sie muss in der Manifest-Datei MANIFEST.MF in der „Main:“-Zeile angegeben werden). Von der Klasse Counter werden in der Start-Methode der Klasse CounterService zwei Objekte erzeugt und bei der Registratur des Frameworks angemeldet. In der Stop-Methode werden diese Objekte wieder abgemeldet.

Listing 6.4 Klasse CounterService

```
package javacomp.prototype.application2;

import javacomp.prototype.framework.ComponentContext;
import javacomp.prototype.framework.Start;
import javacomp.prototype.framework.Stop;

public class CounterService
{
    @Start
    public void start(ComponentContext ctx)
    {
        System.out.println("CounterService.start");
        Counter c1 = new Counter();
        ctx.bind("Counter1", c1);
        Counter c2 = new Counter(100);
```

```
        ctx.bind("Counter2", c2);
    }

    @Stop
    public void stop(ComponentContext ctx)
    {
        System.out.println("CounterService.stop");
        ctx.unbind("Counter1");
        ctx.unbind("Counter2");
    }
}
```

Auf den ersten Blick sieht es so aus, als könnte auf die Registratur nur in den Start- und Stop-Methoden zugegriffen werden. Das ist aber nicht richtig. Es könnte nämlich in der Start-Methode auch ein Thread gestartet werden, dem die Referenz auf den ComponentContext mitgegeben wird. Somit könnte eine Komponente auch nach der Rückkehr von einer Start-Methode noch auf den ComponentContext zugreifen. Wir werden bei den hier vorgestellten Beispielkomponenten davon keinen Gebrauch machen. Die Methoden, welche die Schnittstelle ComponentContext implementieren, sind aus diesem Grund dennoch vorsorglich alle mit synchronized gekennzeichnet (s. später in Listing 6.11).

6.1.3 Komponente Nr. 3

In der dritten Komponente, welche die zweite nutzen möchte, machen wir nun von der Möglichkeit Gebrauch, Threads zu starten, die aber nicht auf die Registratur des Komponentensystems zugreifen. In der Start-Methode beschafft sich die Komponente Nr. 3 Zugriff auf die zwei in der Registratur angemeldeten Counter-Objekte. Anschließend wird für jedes dieser Counter-Objekte ein Thread gestartet, in dem wiederholt die Methode increment auf das entsprechende Counter-Objekt angewendet und der zurückgegebene Wert ausgegeben wird. Damit der Bildschirm nicht mit Ausgaben überflutet wird, wird zwischen je zwei Ausgaben eine bestimmte Zeit gewartet. Die laufenden Threads werden in der Stop-Methode der Komponente angehalten. In Listing 6.5 ist der Programmcode unserer dritten Komponente zu finden. Da davon ausgegangen wird, dass die Leserinnen und Leser mit dem Thema Threads in Java vertraut sind, benötigt der Programmcode keine weiteren Erklärungen.

Listing 6.5 Klassen CounterClient und CounterThread

```
package javacomp.prototype.application3;

import javacomp.prototype.application2.Counter;
import javacomp.prototype.framework.ComponentContext;
import javacomp.prototype.framework.Start;
import javacomp.prototype.framework.Stop;

public class CounterClient
{
    private CounterThread thread1;
    private CounterThread thread2;

    @Start
    public void start(ComponentContext ctx)
```

```
    {
        System.out.println("CounterClient.start");
        Counter c1 = (Counter)ctx.lookup("Counter1");
        if(c1 != null && thread1 == null)
        {
            thread1 = new CounterThread("client1", c1, 7000);
            thread1.start();
        }
        Counter c2 = (Counter)ctx.lookup("Counter2");
        if(c2 != null && thread2 == null)
        {
            thread2 = new CounterThread("client2", c2, 5000);
            thread2.start();
        }
    }

    @Stop
    public void stop()
    {
        System.out.println("CounterClient.stop");
        if(thread1 != null)
        {
            thread1.interrupt();
            try
            {
                thread1.join();
            }
            catch(InterruptedException e)
            {
            }
            thread1 = null;
        }
        if(thread2 != null)
        {
            thread2.interrupt();
            try
            {
                thread2.join();
            }
            catch(InterruptedException e)
            {
            }
            thread2 = null;
        }
    }
}

class CounterThread extends Thread
{
    private Counter c;
    private long sleepTime;

    public CounterThread(String name, Counter c, long sleepTime)
    {
        super(name);
        this.c = c;
        this.sleepTime = sleepTime;
    }
```

```
    public void run()
    {
        try
        {
            while(!isInterrupted())
            {
                System.out.println(getName() + ": " +
                                   c.increment());
                sleep(sleepTime);
            }
        }
        catch(InterruptedException e)
        {
        }
    }
}
```

Wenn wir unsere Komponente installieren wollen, bauen wir eine Jar- oder Zip-Datei, in der sich die Manifest-Datei sowie die Class-Dateien für CounterClient und CounterThread befinden, und legen sie in das Installationsverzeichnis unseres Frameworks. Wir erleben aber eine Enttäuschung, denn es wird die folgende Ausnahme geworfen:

```
ClassNotFoundException: javacomp.prototype.application2.Counter
```

Die neue Komponente kann also offensichtlich nicht auf die Counter-Klasse zugreifen. Beim Übersetzen musste dies allerdings möglich gewesen sein, sonst wäre die Übersetzung gescheitert. Wenn Sie z. B. Eclipse verwenden und Sie haben beide Anwendungen innerhalb eines Java-Projekts, dann haben Sie keinerlei Probleme beim Übersetzen. Wenn Sie zwei unterschiedliche Java-Projekte für die beiden Komponenten in Eclipse angelegt haben, dann müssen Sie im Projekt der CounterClient-Komponente das CounterService-Projekt in den Build-Path aufnehmen.

Man könnte nun versuchen, die Class-Datei für die Klasse Counter in die Jar- bzw. Zip-Datei für die dritte Komponente mit aufzunehmen. Dieser Versuch scheitert aber ebenfalls, denn nun wird folgende Ausnahme geworfen:

```
ClassCastException: javacomp.prototype.application2.Counter cannot
be cast to javacomp.prototype.application2.Counter
```

Was für eine kuriose Ausnahmemeldung! Dass es nicht möglich sein soll, einen Typ X auf X zu casten, ist eine für den unbedarften Java-Programmierer völlig verwirrende Meldung. Verständlich wird dies jedoch mit den Erläuterungen zum Klassenladen aus dem ersten Teil des Buchs: Eine Klasse ist eindeutig durch ihren Namen und das Klassenlader-Objekt, das sie geladen hat, spezifiziert. Die Nachricht in der geworfenen Ausnahme will also ausdrücken, dass die Klasse, die mit einem bestimmten Klassenlader-Objekt geladen wurde, nicht kompatibel ist mit einer Klasse desselben Namens, die aber mit einem anderen Klassenlader-Objekt geladen wurde. Unser Komponenten-Framework benutzt für das Laden jeder Komponente ein neues Klassenlader-Objekt, damit – wie oben für die erste Beispielkomponente beschrieben wurde – Hot Deployment möglich ist. Somit wird also die Klasse Counter in unserer CounterService-Komponente von einem anderen Klassenlader-Objekt geladen als die Klasse Counter in unserer CounterClient-Komponente.

Als Lösung für dieses Problem bietet unser prototypisches Komponenten-Framework an, in der Manifest-Datei durch eine mit „Uses:“ beginnende Zeile anzugeben, dass eine Komponente K1 eine andere Komponente K2 benutzt. Eine Komponente wird dabei durch den Namen ihrer Jar- bzw. Zip-Datei identifiziert. Wir ergänzen also die Manifest-Datei der CounterClient-Komponente durch eine entsprechende Zeile, wobei davon ausgegangen wird, dass die CounterService-Komponente in einer Datei mit dem Namen application2.zip dem Framework zur Verfügung gestellt wurde. Die Manifest-Datei der Komponente Nr. 3 hat damit folgenden Inhalt:

```
Main:  javacomp.prototype.application3.CounterClient
Uses: application2.zip
```

Nachdem wir diese Erweiterung durchgeführt haben, funktioniert unsere Komponente wie gewünscht. Nach der Installation erscheint folgende Ausgabe auf dem Bildschirm:

```
CounterClient.start
client1: 1
client2: 101
client1: 2
client2: 102
...
```

Das Laufen der beiden Threads wird durch die sich ständig wiederholenden Ausgaben belegt.

Für die erfolgreiche Installation der dritten Beispielkomponente ist es übrigens unerheblich, ob sich die Class-Datei Counter.class immer noch in der Installationsdatei der Komponente befindet (wie im zweiten Versuch) oder wieder gelöscht wurde (wie im ersten Versuch). Es wird in jedem Fall die Counter-Klasse der CounterService-Komponente benutzt. Warum dies so ist und was die „Uses:“-Zeile bewirkt, wird im Abschnitt 6.2 deutlich werden, in dem wir die Implementierung des Frameworks beschreiben.

Nachdem die Installationsdatei dieser Komponente gelöscht wurde, verstummen die Meldungen, da die Threads in der Stop-Methode angehalten werden. Wird dagegen nur die Installationsdatei der zweiten Komponente gelöscht, so hat dies keine Auswirkungen auf die dritte Komponente. Zwar wird die Stop-Methode der zweiten Komponente aufgerufen, was die Entfernung der beiden Einträge aus der Registratur bewirkt, aber die Referenzen auf die beiden von der zweiten Komponente erzeugten Objekte werden der dritten Komponente nicht entzogen. Auch wird die Klasse Counter dadurch nicht „entladen“. Selbst wenn die zweite Komponente anschließend mit einer neuen Version der Klasse Counter erneut installiert werden würde, könnte die dritte Komponente auf diese neue Klassenversion nicht zugreifen. Sollte sich Ihnen die Begründung für diesen Sachverhalt nicht erschließen, so vertröste ich Sie abermals auf den Abschnitt 6.2; im Zusammenhang mit der Implementierung des Frameworks sollte es klar werden.

6.1.4 Rückblick

Nachdem wir drei Beispielkomponenten für unser prototypisches Komponentensystem kennengelernt haben, blicken wir nochmals auf Bild 6.1 zurück, das jetzt noch besser verstanden werden sollte. Die vertikalen Pfeile von unten nach oben symbolisieren die Start-

und Stop-Methoden, die vom Framework in den einzelnen Komponenten aufgerufen werden. Die Pfeile von oben nach unten stellen die Nutzung des Frameworks durch die Komponenten über die Schnittstelle ComponentContext (bind, unbind, lookup) dar. Die horizontalen Pfeile schließlich zeigen die Nutzung einer Komponente durch eine andere (im konkreten Fall geht es um die Anwendung der Methode increment auf die Counter-Objekte der zweiten Komponente durch die dritte Komponente).

In unserem einfachen Framework kann eine Komponente nur höchstens eine Komponente nutzen. In der „Uses:“-Zeile können also nicht mehrere Komponenten angegeben werden, die benutzt werden sollen, sondern nur genau eine. Werden mehrere „Uses:“-Zeilen in der Manifest-Datei angegeben, wird nur die letzte davon berücksichtigt (dasselbe gilt, falls mehrere „Main:“-Zeilen vorkommen). Umgekehrt ist es aber möglich, dass eine Komponente von mehreren Komponenten benutzt wird. Wir könnten also problemlos eine weitere Komponente Nr. 4 entwickeln, die ebenfalls die von der zweiten Komponente bereitgestellten Objekte verwendet. Dies ist in Bild 6.1 bereits angedeutet: Die Komponenten Nr. 3 und 4 benutzen beide die Komponente Nr. 2.

Damit es bei der parallelen Nutzung der Counter-Objekte durch mehrere Threads (wie in der dritten Komponente) zu keinen Synchronisationsproblemen kommt, wurde die Methode increment in Listing 6.3 bereits vorsorglich mit synchronized gekennzeichnet.

6.1.5 Variation der Komponentenbeispiele

Bei den vorgestellten Beispielkomponenten bedeutete die Nutzung einer Komponente durch eine andere, dass eine Komponente (in unserem Fall die CounterService-Komponente) Objekte einer Klasse (Counter), die zu dieser Komponente gehörte, erzeugte und dass eine andere Komponente (in unserem Fall die CounterClient-Komponente) diese Objekte und damit auch die dazugehörige Klasse nutzte. Um Missverständnissen vorzubeugen sei an dieser Stelle erwähnt, dass die Nutzung einer Komponente durch eine andere Komponente in dieser Form zwar nicht untypisch ist, dass es aber nicht in jedem Fall so sein muss. Es ist durchaus auch möglich, dass eine Komponente nur Klassen, aber nicht unbedingt auch Objekte dieser Klassen für andere Komponenten zur Verfügung stellt. Als Beispiel vereinfachen wir unsere zweite und dritte Komponente entsprechend.

Die zweite Komponente besteht jetzt aus der Klasse Counter (s. Listing 6.3) wie bisher und einer leeren Klasse CounterService (s. Listing 6.6).

Listing 6.6 Klasse CounterService (stark vereinfacht)

```
package javacomp.prototype.application5;

public class CounterService
{
}
```

Die nutzende CounterClient-Komponente beschafft sich jetzt keine von der CounterService-Komponente bereitgestellten Objekte, sondern verwendet nur die Klasse Counter, wobei sie die benötigten Objekte dieser Klasse selbst erzeugt. Da der ComponentContext nicht benutzt wird, genügt deshalb eine parameterlose Start-Methode. Wie in Listing 6.5 könnte die Nut-

zung von Counter-Objekten wieder in Threads erfolgen. In Listing 6.7 wird aber lediglich der entscheidende Punkt für dieses Beispiel gezeigt, nämlich die Erzeugung und Nutzung eines Objekts der Klasse Counter durch die CounterClient-Komponente.

Listing 6.7 CounterClient-Klasse mit Erzeugung eines Counter-Objekts

```
package javacomp.prototype.application6;

import javacomp.prototype.application5.Counter;
import javacomp.prototype.framework.Start;

public class CounterClient
{
    @Start
    public void start()
    {
        System.out.println("CounterClient.start");
        Counter c = new Counter(1000);
        System.out.println("counter = " + c.increment());
    }
}
```

Bei der Installation der CounterService-Komponente wird ein Objekt der Klasse CounterService erzeugt. Da in der CounterService-Klasse die Klasse Counter nicht benutzt wird, muss sie aufgrund des Lazy-Loading-Prinzips auch noch gar nicht geladen werden. Erst wenn die CounterClient-Komponente geladen und die Start-Methode auf ein erzeugtes CounterClient-Objekt angewendet wird, wird Counter benutzt. Erst dann wird die Klasse Counter, die zu einer anderen Komponente (CounterService) gehört, geladen. Dass dies tatsächlich so ist, kann man beobachten, wenn das Framework (s. Abschnitt 6.2) mit der Option „-verbose“ gestartet und somit für jede Klasse, die geladen wird, eine Meldung auf der Konsole ausgegeben wird:

```
[Loaded javacomp.prototype.application6.CounterClient from file:...]
...
CounterClient.start
[Loaded javacomp.prototype.application5.Counter from file:...]
counter = 1001
```

An den Ausgaben kann man deutlich ablesen, dass die Klasse Counter erst geladen wird, wenn die Start-Methode von CounterClient ausgeführt wird.

Selbstverständlich könnte man dieses Beispiel noch weiter vereinfachen, indem die Klasse CounterService ganz weggelassen und die Klasse Counter als Einstiegsklasse der CounterService-Komponente bestimmt wird. Dann würde die Klasse Counter logischerweise schon bei der Installation der CounterSevice-Komponente geladen. Dies würde aber nichts daran ändern, dass immer noch eine Klasse einer Komponente von einer anderen Komponente genutzt wird, ohne dass Objekte dieser Klasse in der Registratur abgelegt werden.

6.2 Framework

Durch die Beschreibung der Beispielkomponenten im vorhergehenden Abschnitt sollte deutlich geworden sein, was das Framework zu leisten hat. In vereinfachter Form kann es so zusammengefasst werden:

- Das Framework muss eine spezielles Verzeichnis auf Änderungen beobachten.
- Für jede neue Jar- oder Zip-Datei muss in der Manifest-Datei nachgesehen werden, wie die Einstiegsklasse heißt. Dann muss für diese Klasse ein Objekt erzeugt werden (das „Hauptobjekt“ für diese Komponente). Anschließend werden auf dem „Hauptobjekt“ alle mit @Start annotierten Methoden aufgerufen, die die passende Signatur haben.
- Falls eine Jar- oder Zip-Datei gelöscht wurde, müssen auf dem „Hauptobjekt“ alle mit @Stop versehenen Methoden aufgerufen werden, die die passende Signatur haben.
- Falls eine Jar- oder Zip-Datei aktualisiert wurde, kann das Framework sich so verhalten, als wäre die Datei gelöscht und anschließend neu erzeugt worden.

6.2.1 Struktur des Komponenten-Frameworks

Die Struktur des Frameworks mit den vier wichtigsten Objekten ist in Bild 6.3 dargestellt. Das Framework ist bewusst sehr einfach gehalten; seine Implementierung hat weniger als einen Tag in Anspruch genommen. Dennoch stellt es einige typische Funktionen eines Komponentensystems bereit und basiert auf einigen der im ersten Teil des Buchs vermittelten Java-Grundlagen.

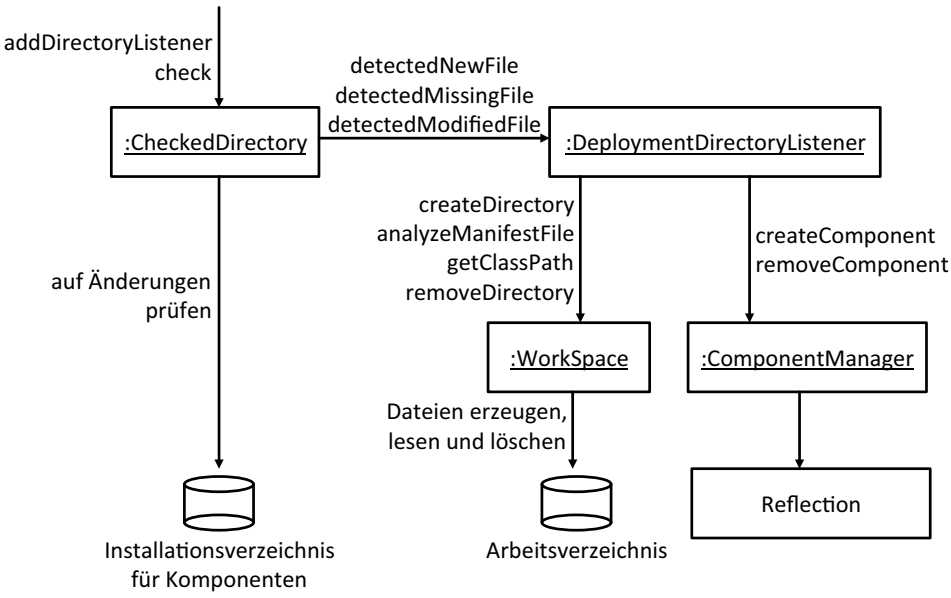

Bild 6.3 Zusammenwirken der wichtigsten Objekte des Frameworks

Die Überwachung des Installationsverzeichnisses wird von einem Objekt der Klasse CheckedDirectory vorgenommen. Im Konstruktor dieser Klasse muss der Name des Verzeichnisses übergeben werden, das überwacht werden soll. Optional kann ein Filter angegeben werden, für welche Dateien die Überwachung durchgeführt werden soll. In unserem Fall wird ein Filter verwendet, der auf alle Dateien passt, die die Endung „.jar“ oder „.zip“ haben. Mit der Methode addDirectoryListener können Listener angemeldet werden, die benachrichtigt werden, falls sich im Verzeichnis für die auf den Filter passenden Dateien Änderungen ergeben (Erzeugung, Löschung oder Aktualisierung). Objekte der Klasse CheckedDirectory sind nicht selbst aktiv, indem z. B. im Konstruktor ein Thread für die Überwachung erzeugt wird. Die Überwachung muss durch wiederholtes Aufrufen der Methode check „von außen“ am Laufen gehalten werden. In der Methode check werden die Änderungen gegenüber dem letzten Aufruf dieser Methode festgestellt. Für jede neue Datei wird auf allen angemeldeten Listenern die Methode detectedNewFile aufgerufen, für jede gelöschte Datei die Methode detectedMissingFile und für jede aktualisierte Datei die Methode detectedModifiedFile.

Wie das Installationsverzeichnis durch ein Objekt der Klasse CheckedDirectory im Framework repräsentiert wird, so erfolgt der Zugriff auf das Arbeitsverzeichnis durch ein Objekt der Klasse WorkSpace. Für jede Komponente wird im Arbeitsverzeichnis ein eigenes Verzeichnis angelegt. Das Arbeitsverzeichnis bekommt denselben Namen wie die Jar- oder Zip-Datei der Komponente inklusive der Endung. Wenn also die Datei mit dem Namen „application1.zip“ in das Installationsverzeichnis gelegt wird, dann wird ein Verzeichnis mit dem Namen „application1.zip“ im Arbeitsverzeichnis erzeugt, obwohl es sich nicht um eine Zip-Datei handelt. In dieses neu erzeugte Verzeichnis werden die Inhalte der Jar- bzw. Zip-Datei entpackt. Dies alles erfolgt durch die Methode createDirectory (die Installationsdatei der neuen Komponente ist der Parameter dieser Methode). Der Name der Installationsdatei bzw. der Name des erzeugten Verzeichnisses dient intern als Name für die Komponente. Mit analyzeManifestFile (String-Parameter für den Namen der Komponente) wird die Manifest-Datei analysiert. Das Ergebnis wird in Form einer HashMap<String, String> zurückgegeben. Für jede Zeile der Form „x: y“ in der Manifest-Datei gibt es einen Eintrag in der HashMap mit „x“ als Schlüssel und „y“ als dazugehörigem Wert. Von der Methode getClassPath (wieder mit dem Namen der Komponente als Parameter) wird der vollständige Pfadname für das Verzeichnis „classes“ innerhalb des für eine Komponente angelegten Verzeichnisses im Arbeitsverzeichnis zurückgegeben, und dies in Form eines URL-Feldes der Länge 1, damit es direkt für den Konstruktor von URLClassLoader verwendet werden kann. Mit removeDirectory (Parameter ist wieder der Komponentenname) wird – wie der Name sagt – das durch createDirectory angelegte Verzeichnis einer Komponente wieder gelöscht.

Da die beiden Klassen CheckedDirectory und WorkSpace im Wesentlichen den Zugriff auf das Dateisystem bewerkstelligen und wenig mit dem Komponentensystem zu tun haben, werden wir in diesem Buch nicht näher auf den Code eingehen. Selbstverständlich können Sie das gesamte Framework von der Web-Seite zum Buch beziehen und sich bei Interesse diese Klassen genauer anschauen. Auf die beiden anderen Klassen ComponentManager und DeploymentDirectoryListener gehen wir dagegen ausführlicher ein.

6.2.2 Die Klasse ComponentManager

Die entscheidende Funktionalität des Komponenten-Frameworks befindet sich in der Klasse ComponentManager. Den vollständigen Code dieser zentralen Klasse finden Sie in Listing 6.8.

Listing 6.8 Klasse ComponentManager

```
package javacomp.prototype.framework;

import java.lang.annotation.Annotation;
import java.lang.reflect.Method;
import java.net.*;
import java.util.*;

class ComponentManager
{
    private HashMap<String, Object> components;
    private ComponentContext context;

    public ComponentManager()
    {
        components = new HashMap<String, Object>();
        context = new ComponentContextImpl();
    }

    public void createComponent(String componentName,
                                HashMap<String,String> configInfo,
                                URL[] classpath)
           throws Exception
    {
        String mainClassName = configInfo.get("Main");
        if(mainClassName == null)
        {
            return;
        }

        ClassLoader parent = null;
        String usedComp = configInfo.get("Uses");
        if(usedComp != null)
        {
            Object obj = components.get(usedComp);
            if(obj != null)
            {
                parent = obj.getClass().getClassLoader();
            }
        }

        ClassLoader cl;
        if(parent != null)
        {
            cl = new URLClassLoader(classpath, parent);
        }
        else
        {
            cl = new URLClassLoader(classpath);
        }
```

```
        Object mainObject = null;
        try
        {
            Class<?> newClass = cl.loadClass(mainClassName);
            mainObject = newClass.newInstance();
            callAnnotatedMethods(mainObject, Start.class);
        }
        catch(Exception e)
        {
            System.out.println("---exception in addComponent");
            e.printStackTrace();
        }
        components.put(componentName, mainObject);
    }

    public void removeComponent(String componentName)
    {
        Object mainObject = components.get(componentName);
        if(mainObject == null)
        {
            return;
        }
        callAnnotatedMethods(mainObject, Stop.class);
        components.remove(componentName);
    }

    public void removeAllComponents()
    {
        String[] componentNames =
                        components.keySet().toArray(new String[0]);
        for(String componentName: componentNames)
        {
            removeComponent(componentName);
        }
    }

    private void callAnnotatedMethods(Object mainObject,
                              Class<? extends Annotation> annoClass)
    {
        Class<?> classDesc = mainObject.getClass();
        Method[] methods = classDesc.getMethods();
        for(Method m : methods)
        {
            Class<?>[] params = m.getParameterTypes();
            Class<?> returnType = m.getReturnType();
            if(m.isAnnotationPresent(annoClass) &&
               returnType == void.class)
            {
                if(params.length == 0)
                {
                    try
                    {
                        m.invoke(mainObject);
                    }
                    catch(Exception e)
                    {
                        e.printStackTrace();
                    }
```

```
                }
                else if(params.length == 1 &&
                        params[0] == ComponentContext.class)
                {
                    try
                    {
                        m.invoke(mainObject, context);
                    }
                    catch(Exception e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        }
    }
}
```

Die Klasse ComponentManager besitzt zwei Attribute: Das erste ist eine HashMap, in der zu jedem Komponentennamen das „Hauptobjekt“ der Einstiegsklasse (in der Manifest-Datei unter Main angegeben), das vom Komponenten-Framework bei der Installation einer Komponente erzeugt wird, gespeichert wird. Das zweite Attribut ist die Registratur, mit der Komponenten erzeugte Objekte anmelden und anderen zur Verfügung stellen können. Dieses Attribut wird den Start- und Stop-Methoden mit ComponentContext-Parameter übergeben.

Die interessanteste Methode ist createComponent. Ihre Parameter sind der Name der neuen Komponente, die HashMap, welche den Inhalt der Manifest-Datei repräsentiert (von der Methode analyzeManifestFile der Klasse WorkSpace zurückgegeben), sowie der Classpath für die neu zu erzeugende Komponente in Form eines URL-Feldes (von der Methode getClassPath der Klasse WorkSpace zurückgeliefert). Beim Erzeugen des ClassLoader-Objekts, das die Einstiegsklasse laden soll, ist entscheidend, ob die Manifest-Datei eine „Uses:“-Zeile hat. Wenn nicht, wird einfach ein URLClassLoader-Objekt mit dem Classpath-Parameter (drittes Argument der Methode createComponent) erzeugt. Wie im ersten Teil des Buches beschrieben, wird dieses Objekt dann in den Baum der System-ClassLoader an einer Standardposition eingebaut. Andernfalls, falls eine „Uses:“-Zeile vorhanden ist, muss das neue ClassLoader-Objekt an das ClassLoader-Objekt der benutzten Komponente angehängt werden. Die benutzte Komponente muss in der Manifest-Datei in Form ihrer Installationsdatei angegeben werden. Wie beschrieben ist dieser Name auch der intern verwendete Name der Komponente. Über den Namen der benutzten Komponente findet man das dafür erzeugte „Hauptobjekt“ und damit die dazugehörige Klasse und den ClassLoader, der diese Klasse geladen hat. Wenn man diesen ClassLoader finden konnte, wird er als der ClassLoader verwendet, an den der neu erzeugte ClassLoader angehängt wird. Nachdem das ClassLoader-Objekt erzeugt wurde, kann man die in der Manifest-Datei angegebene Einstiegsklasse laden, ein Objekt dieser Klasse erzeugen (ein öffentlicher, parameterloser Konstruktor muss dazu vorhanden sein) sowie alle passenden, mit @Start annotierten Methoden aufrufen. Schließlich wird dann noch das erzeugte Objekt der Einstiegsklasse mit dem Namen der Komponente als Schlüssel in der HashMap gespeichert.

Nachdem wir gesehen haben, was eine „Uses:“-Zeile in einer Manifest-Datei bewirkt, blicken wir nochmals auf unsere Beispielkomponenten aus Abschnitt 6.1 zurück. Ohne einen „Uses:“-Eintrag in der CounterClient-Komponente sind die ClassLoader so angeordnet, wie in Bild 6.4 gezeigt wird.

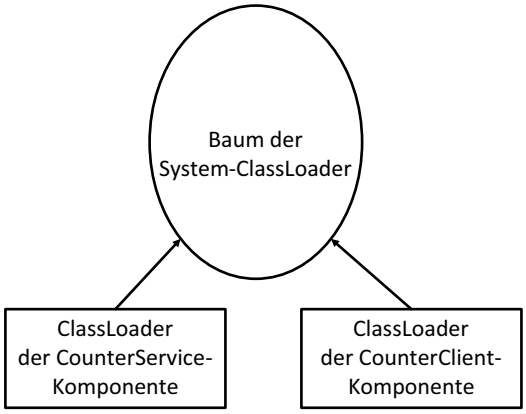

Bild 6.4 Anordnung der ClassLoader-Objekte ohne „Uses:“-Angabe

Wenn also in der CounterClient-Komponente auf die Klasse Counter der CounterService-Komponente zugegriffen werden soll, dann muss diese Klasse von den entsprechenden ClassLoadern bereits geladen sein oder geladen werden. Als Erstes wird der ClassLoader der Komponente CounterClient kontaktiert. Dieser delegiert den Auftrag an seinen Vorgänger im Baum, womit der Auftrag dann bis zur Wurzel läuft. Keiner der ClassLoader in dieser Kette kann aber die Klasse Counter laden, da sie in keinem der Klassenpfade dieser ClassLoader vorkommt. Folglich scheitert der Zugriff auf die Klasse Counter und es wird die bekannte Ausnahme ClassNotFoundException geworfen. Wie schon beschrieben bringt es auch nichts, wenn die CounterClient-Komponente die Counter-Klasse selbst mitbringt (ClassCastException).

Wenn aber andererseits der „Uses:“-Eintrag in der Manifest-Datei der CounterClient-Komponente vorhanden ist und auf die CounterService-Komponente verweist, ergibt sich die Anordnung von ClassLoader-Objekten wie in Bild 6.5.

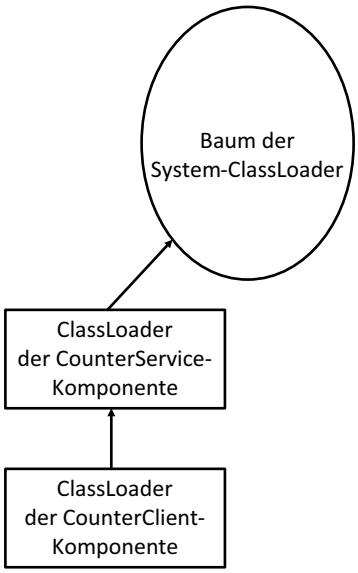

Bild 6.5 Anordnung der ClassLoader-Objekte mit „Uses:“-Angabe

Wenn jetzt die CounterClient-Komponente auf die Klasse Counter zugreifen möchte, wird der ClassLoader dieser Komponente den Auftrag an seinen Vorgänger weiterleiten. Dies ist nun aber der ClassLoader der CounterService-Komponente, der die Counter-Klasse entweder schon geladen hat oder ansonsten laden kann. Der Zugriff auf die Counter-Klasse gelingt also. Es spielt somit auch keine Rolle, ob die CounterClient-Komponente eine eigene Version der Counter-Klasse mitbringt oder nicht. Da zuerst der Vorgänger-ClassLoader befragt wird, wird in jedem Fall die Version der Counter-Klasse, die in der CounterService-Komponente enthalten ist, von der CounterClient-Komponente verwendet.

Ein weiterer in Abschnitt 6.1.3 erwähnter Effekt wird damit auch verständlich: Auch wenn die CounterService-Komponente mit einer neuen Version der Klasse Counter aktualisiert wird, verwendet die CounterClient-Komponente weiterhin die alte Counter-Version. Wenn man sich den Baum der Klassenlader-Objekte ansieht (s. Bild 6.6), leuchtet dies unmittelbar ein.

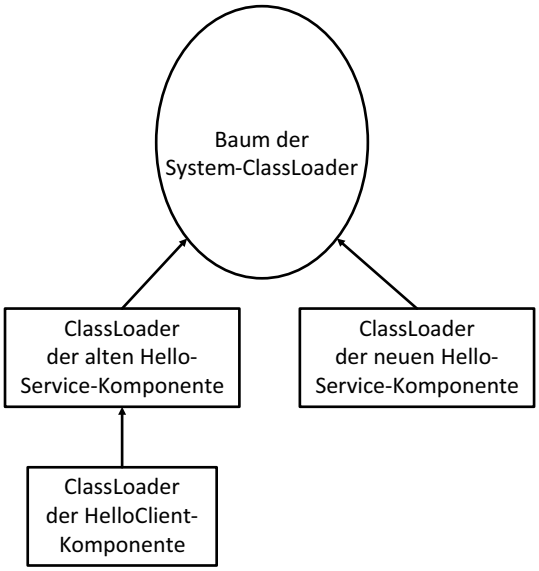

Bild 6.6 Anordnung der ClassLoader-Objekte nach einer Neuinstallation der CounterService-Komponente

Die weiteren Methoden der Klasse ComponentManager sind einfacher verständlich. In removeComponent werden alle passenden, mit @Stop annotierten Methoden aufgerufen und der Eintrag für die Komponente wird aus der HashMap gelöscht. Durch removeAllComponents wird removeComponent für jede vorhandene Komponente aufgerufen. Diese Methode wird aufgerufen, wenn das Framework angehalten wird. Die private Methode callAnnotatedMethods wird von createComponent und removeComponent verwendet, um alle passenden, mit @Start bzw. @Stop gekennzeichneten Methoden aufzurufen. Passende Methoden sind solche, deren Rückgabetyp void ist und die parameterlos sind oder einen einzigen Parameter des Typs ComponentContext haben. Letztere werden mit einer Referenz auf das zweite Attribut dieser Klasse aufgerufen, welches die Schnittstelle ComponentContext implementiert.

6.2.3 Die Klasse DeploymentDirectoryListener

Die Klasse DeploymentDirectoryListener implementiert die Schnittstelle DirectoryListener (s. Listing 6.9).

Listing 6.9 Schnittstelle DirectoryListener

```
package javacomp.prototype.framework;

import java.io.File;

interface DirectoryListener
{
    public void detectedNewFile(File newFile);
    public void detectedMissingFile(String missingFileName);
    public void detectedModifiedFile(File modifiedFile);
}
```

Ein Objekt der Klasse DeploymentDirectoryListener kann deshalb mit addDirectoryListener an dem CheckedDirectory-Objekt angemeldet werden, welches das Installationsverzeichnis für Komponenten überwacht. Wenn man auf die Funktionalität der benutzten Klassen WorkSpace und ComponentManager zurückgreift, ergibt sich die Realisierung der Klasse DeploymentDirectoryListener praktisch von selbst (s. Listing 6.10).

Listing 6.10 Klasse DeploymentDirectoryListener

```
package javacomp.prototype.framework;

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

class DeploymentDirectoryListener implements DirectoryListener
{
    private WorkSpace workSpace;
    private ComponentManager compManager;

    public DeploymentDirectoryListener(WorkSpace workSpace,
                                       ComponentManager compManager)
    {
        this.workSpace = workSpace;
        this.compManager = compManager;
    }

    public void detectedNewFile(File zipFile)
    {
        try
        {
            workSpace.createDirectory(zipFile);
            String compName = zipFile.getName();
            HashMap<String,String> configInfo =
                            workSpace.analyzeManifestFile(compName);
            URL[] classpath = workSpace.getClassPath(compName);
            compManager.createComponent(compName, configInfo,
                                        classpath);
        }
```

```
        catch(Exception e)
        {
            System.err.println("---" + e.getMessage());
            e.printStackTrace();
        }
    }

    public void detectedMissingFile(String missingFileName)
    {
        compManager.removeComponent(missingFileName);
        workSpace.deleteDirectory(missingFileName);
    }

    public void detectedModifiedFile(File modifiedFile)
    {
        detectedMissingFile(modifiedFile.getName());
        detectedNewFile(modifiedFile);
    }
}
```

6.2.4 Restliche Klassen

Mit den vier in Bild 6.3 vorkommenden Klassen haben wir den wesentlichen Teil des Frameworks kennengelernt. Eine weitere Klasse des Frameworks ist die Klasse ComponentContextImpl, welche die Schnittstelle ComponentContext implementiert und die Registratur des Komponentensystems realisiert. Sie basiert auf einer HashMap. Wie schon zuvor erwähnt wurde, ist es möglich, dass Komponenten Threads erzeugen, und dass diese Threads auf die Registratur zugreifen. Aus diesem Grund sind alle Methoden synchronized.

Listing 6.11 Klasse ComponentContextImpl

```
package javacomp.prototype.framework;

import java.util.HashMap;

class ComponentContextImpl implements ComponentContext
{
    private HashMap<String,Object> registry;

    public ComponentContextImpl()
    {
        registry = new HashMap<String,Object>();
    }

    public synchronized void bind(String name, Object obj)
    {
        registry.put(name, obj);
    }

    public synchronized Object lookup(String name)
    {
        return registry.get(name);
    }
```

```
    public synchronized void unbind(String name)
    {
        registry.remove(name);
    }
}
```

Übrigens könnte man bei der Methode bind noch prüfen, ob schon ein Eintrag desselben Namens in der Registratur existiert. Falls dies so ist, so könnte man eine Ausnahme (z. B. eine IllegalArgumentException) werfen. In unserer Implementierung wird dies nicht geprüft. Somit würde ein Eintrag, den es für diesen Namen schon gibt, durch den neuen Eintrag überschrieben. Eine weitere Variante wäre, dass man sich für jeden Eintrag merkt, von welcher Komponente er stammt. Beim Entfernen einer Komponente könnte man dann automatisch alle die Einträge, welche die gerade zu entfernende Komponente hinzugefügt und noch nicht selbst wieder herausgenommen hat, löschen. Um die Implementierung möglichst einfach zu halten, wurden diese Varianten nicht realisiert, obwohl es keinen allzu großen Aufwand erfordern würde. OSGi bietet diesen Komfort im Gegensatz dazu.

Eine weitere Klasse ist die Klasse Main, welche die Methode main enthält, die das Framework ausführt. Die Methode main enthält die Initialisierung der in Bild 6.3 gezeigten Objekte sowie eine Schleife, in der check auf das erzeugte CheckedDirectory-Objekt und anschließend sleep zum Pausieren wiederholt aufgerufen wird. Durch das Drücken der Return-Taste wird das Framework wieder beendet. Dabei werden alle noch vorhandenen Komponenten entfernt. Für die Namen des Installations- und des Arbeitsverzeichnisses sind Konstanten in der Klasse Main definiert, die als Parameter den Konstruktoren von CheckedDirectory bzw. WorkSpace übergeben werden. Sollten Sie das Framework selbst auf Ihrem Rechner ausführen wollen, so ist es ratsam, die Namen der Verzeichnisse zuerst Ihren Verhältnissen anzupassen. Die Verzeichnisse müssen übrigens nicht existieren; das Framework legt die Verzeichnisse selber an, falls sie nicht vorhanden sind. Das Arbeitsverzeichnis wird am Ende auch wieder gelöscht.

Der komplette Code des Frameworks befindet sich im Package javacomp.prototype.framework. Die in diesem Abschnitt bisher diskutierten Klassen und Schnittstellen zur Realisierung des Frameworks sind (mit Ausnahme von Main) nicht-öffentlich. Daneben gibt es noch die öffentlichen Elemente, die von den Komponenten dieses Frameworks benutzt werden. Dies sind die Schnittstelle ComponentContext, die schon in Listing 6.2 gezeigt wurde, sowie die Annotationen @Start (s. Listing 6.12) und @Stop (sehr ähnlich wie @Start).

Listing 6.12 Annotation Start

```
package javacomp.prototype.framework;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Start
{
}
```

7 Komponenten und Komponentensysteme

Nachdem Sie anhand der prototypischen Implementierung aus dem vorigen Kapitel ein konkretes Beispiel eines Komponentensystems vor Augen haben, wollen wir uns nun der Frage nähern, was Komponenten und Komponentensysteme ausmachen. Um es gleich vorwegzunehmen: Eine klare Definition wird es nicht geben. Es werden lediglich einige Merkmale aufgelistet, die Komponenten und Komponentensysteme aufweisen können oder sollen. In der Praxis ist es dann so, dass einige Systeme alle oder viele dieser Merkmale besitzen und dem Ideal eines Komponentensystems sehr nahe kommen. Andere Systeme dagegen können nur einige oder wenige dieser Merkmale vorweisen. Entsprechend handelt es sich bei diesen Systemen dann nur mehr oder weniger um Komponentensysteme, wobei die Einschätzung durchaus auch subjektiv gefärbt sein mag. Ein hartes Kriterium „Komponentensystem ja oder nein“ gibt es meiner Meinung nach nicht. Die Diskussion und die Beispiele in diesem Buch sollen die Leserinnen und Leser aber in die Lage versetzen selbst beurteilen zu können, wie sehr es sich bei einem konkreten System um ein Komponentensystem handelt, wobei unterschiedliche individuelle Einschätzungen durchaus erwünscht sind.

7.1 Modularität als grundlegendes Prinzip von Komponentensystemen

Ein wesentliches Prinzip, das sich im Bereich der Software-Entwicklung, der Informatik im Allgemeinen und auch außerhalb der Informatik wiederfindet, ist das Prinzip „Teile und herrsche“ (lateinisch „divide et impera“, englisch „divide and conquer“). Damit ist gemeint, dass Probleme oder ursprünglich gegnerische Truppen in Teile aufgeteilt werden, die leichter beherrscht werden können. Das Prinzip kann dabei rekursiv angewendet werden: Ein dadurch entstehendes Teil kann für die einfache Beherrschung wieder zu groß sein, so dass es weiter aufgeteilt wird.

Eng damit verwandt ist der Begriff der Modularität, der die zentrale Idee von Komponenten und Komponentensystemen darstellt. Ähnlich wie beim Prinzip „Teile und herrsche“ geht es auch bei dieser Idee, die auch als Baukastenprinzip bezeichnet wird, um das Aufteilen

eines Ganzen in einzelne beherrschbare Teile, die in diesem Fall Module genannt werden. Allerdings werden mit Modulen noch weitere Eigenschaften verbunden:

- Austauschbarkeit (eventuell auch im laufenden Betrieb): Es sollte möglich sein, einzelne Module auszutauschen, ohne dass dadurch andere Module oder das Gesamtsystem beeinträchtigt werden. Der Austausch eines Moduls kann nötig werden, wenn ein Modul fehlerhaft ist. In diesem Fall ersetzt man das Modul durch ein fehlerfreies Exemplar. Ein anderer Grund kann sein, dass man mit der Funktionalität oder Leistungsfähigkeit eines Moduls nicht mehr zufrieden ist und deshalb das Modul durch ein Modul mit erweiterter Funktionalität oder erhöhter Leistungsfähigkeit ersetzt. Dadurch kann das Gesamtsystem an sich ändernde Anforderungen angepasst werden. Im optimalen Fall kann der Austausch einzelner Module im laufenden Betrieb erfolgen.
- Erweiterbarkeit und Reduzierbarkeit (eventuell auch im laufenden Betrieb): Auch durch das Einbringen zusätzlicher Module soll ein Gesamtsystem an neue Anforderungen angepasst werden können. Die neuen Module können für eine zusätzliche Funktionalität, eine Leistungssteigerung durch Lastverteilung oder für Fehlertoleranz sorgen. Werden die neuen Module zum Zweck der Fehlertoleranz „dazugeschaltet“, kann es sein, dass sie erst aktiv werden, nachdem andere Module ausgefallen sind. Auch das Umgekehrte ist durch Module möglich: Gehen die Anforderungen zurück, so können bestimmte Module „abgeschaltet“ und die Funktionalität oder Leistungsfähigkeit des Gesamtsystems dadurch reduziert werden. Im optimalen Fall kann auch hier die Erweiterung und Reduzierung im laufenden Betrieb vorgenommen werden.
- Wiederverwendbarkeit: Weiterhin soll es möglich sein, dass einzelne Module, die in einem System eingesetzt werden, auch in anderen Systemen und damit in einem anderen Kontext eingesetzt werden können. Bei physikalisch vorhandenen Modulen bedeutet dies in der Regel, dass das Modul aus dem einen System aus- und in das andere System eingebaut wird. Durch die leichte Kopierbarkeit von Software muss man im Software-Bereich ein Modul nicht unbedingt aus einem System entfernen, um es in einem anderen System wiederverwenden zu können. Damit ein Modul in einem anderen System wiederverwendet werden kann, ist es notwendig, dass es in das andere System passt.
- Zusammenbau eines Systems aus Standardmodulen: Eine weitere Vorstellung, die mit dem Begriff Modul oder vielleicht eher mit dem Begriff Komponente verbunden ist, ist die Entwicklung eines komplett neuen Systems aus vorhandenen Standardmodulen oder Standardkomponenten, die bereits vorgefertigt in einem Regal bereit liegen und von dort entnommen werden können („off the shelf“). Eine gute Metapher für diese Vorstellung sind Lego-Bausteine. Aus ihnen können Kinder so unterschiedliche Dinge wie Häuser, Flugzeuge, Schiffe usw. bauen. Die Vorstellung, neue Software lediglich durch Kombination bereits vorhandener Bausteine ohne oder nur mit wenig Eigenentwicklung zu generieren, ist zum jetzigen Zeitpunkt allerdings eher eine Vision als gängige Praxis.

Die Vorstellung von der leichten Kombinierbarkeit einzelner Teile eines Gesamtsystems, die durch den Begriff Modularität beschrieben wird, findet sich in unterschiedlichen Disziplinen innerhalb und außerhalb der Informatik. Diese Idee stellt auch den Kerngedanken von Komponenten und Komponentensystemen im Software-Bereich dar. Der Gegensatz zu modularen Systemen sind monolithische Systeme „aus einem Guss“. Sie haben keine klar identifizierbaren Einzelteile bzw. die Einzelteile hängen sehr eng und in komplexer Weise zusammen, so dass ein Austausch eines einzelnen Teils einen unverhältnismäßig hohen Aufwand erfordern würde.

7.2 Definitionen für Software-Komponenten

Im Folgenden werden einige Aussagen, Charakterisierungen oder Definitionen aus der Literatur zum Begriff Komponente im Software-Umfeld aufgeführt. Die Liste ist chronologisch geordnet. Den ersten Text der Liste von McIlroy gebe ich hier wieder, um zu zeigen, dass die Idee von Software-Komponenten bereits in den 60er Jahren des 20. Jahrhunderts in den Köpfen der Leute vorhanden war, wobei damals die Vorstellung einer Industrie für die Produktion von Komponenten und deren Wiederverwendbarkeit im Vordergrund stand:

- McIlroy, 1969: My thesis is that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components subindustry. … I suspect that the very name "software components" has probably already conjured up for you an idea of how the industry could operate.
- Booch, 1987: A reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction.
- Nierstrasz and Dami, 1995: A component is a static abstraction with plugs. By "static", we mean that a software component is a long-lived entity that can be stored in a software base. By "abstraction", we mean that a component puts a more or less opaque boundary around the software it encapsulates. "With plugs" means that there are well-defined ways to interact and communicate with the component (parameters, ports, messages, etc.). Software composition, then, is the process of constructing applications by interconnecting software components through their plugs. The nature of the plugs, the binding mechanisms and the compatibility rules for connecting components can vary quite a bit.
- European Conference on Object-Oriented Programming, 1996: A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.
- Nierstrasz und Lumpe, 1997: Eine Softwarekomponente ist Teil eines Komponenten-Frameworks, das (i) eine Bibliothek von Black-Box-Komponenten zu Verfügung stellt, (ii) eine wiederverwendbare Software-Architektur definiert, in der die Komponenten geeignet integriert sind, und (iii) eine bestimmte Art von Glue, die es uns erlaubt, Komponenten miteinander zu verbinden.
- Szyperski, 1997: Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.
- D'Souza und Wills, 1998: A software component is a coherent package of software implementations that (a) has explicit and well-specified interfaces for services it provides; (b) has explicit and well-specfied interfaces for services it expects; and (c) can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves.
- Councill/Heineman, 2001: A software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

- Arbeitskreis „Komponentenorientierte betriebliche Anwendungssysteme“ der Gesellschaft für Informatik, 2002: Eine Komponente besteht aus verschiedenartigen (Software-) Artefakten. Sie ist wiederverwendbar, abgeschlossen und vermarktbar, stellt Dienste über wohldefinierte Schnittstellen zur Verfügung, verbirgt ihre Realisierung und kann in Kombination mit anderen Komponenten eingesetzt werden, die zur Zeit der Entwicklung nicht unbedingt vorhersehbar ist.
- Object Management Group, 2005: The component is a modular unit with well defined interfaces that is replaceable within its environment. … It has one or more provided and required interfaces, and its internals are hidden and inaccessible other than as provided by its interfaces.
- Handbuch der Software-Architektur, 2009: Komponenten sind modulare Teile eines Systems, die ihren Inhalt und somit komplexes Verhalten transparent kapseln und in ihrer Umgebung als austauschbare Einheiten mit klar definierten Schnittstellen auftauchen.

Die Liste dieser Zitate soll Ihnen zeigen, wie unterschiedlich der Komponentenbegriff definiert wurde, wenngleich es auch einige Charakterisierungen gibt, die häufiger vorkommen. Ich habe keine Definition gefunden, weder unter den oben aufgeführten noch unter den zahlreichen anderen nicht wiedergegebenen, die ich unverändert für dieses Buch übernehmen will, da in keiner Definition die für dieses Buch passende Menge an charakteristischen Merkmalen ausgewählt wurde. Ich halte zudem einige der angegebenen Eigenschaften nicht für wesentlich für ein Komponentensystem. So ist in den Definitionen die Rede, dass Komponenten „contractually specified interfaces“ bzw. „wohldefinierte Schnittstellen“ oder „klar definierte Schnittstellen“ besitzen müssen. Abgesehen davon, dass für mich nicht deutlich wird, was darunter zu verstehen sein soll, wird im dritten Teil des Buches deutlich werden, dass Schnittstellen keine wesentliche Eigenschaft von Komponenten sind. Auch wenn die Wichtigkeit von Schnittstellen immer wieder betont wird, gibt es eine ganze Reihe von Java-Komponentensystemen, deren Komponenten keine Schnittstellen implementieren müssen (die Beispielkomponenten des prototypischen Komponenten-Frameworks aus dem vorigen Kapitel, aber z. B. auch Java Beans und Servlets). Mit Komponenten wird oft auch die Möglichkeit der leichten Wiederverwendbarkeit verbunden, was wohl bedeuten soll, dass die Software einer Komponente in ganz unterschiedlichen Anwendungen eingesetzt werden kann. Wenn auch Wiederverwendung die Grundidee war, als der Begriff Komponente in den 60er Jahren in die Software-Welt eingeführt wurde, so steht er nach meiner Einschätzung bei den meisten Java-Komponentensystemen nicht im Vordergrund. Wenn ich z. B. einige Eclipse-Plugins betrachte, die ich sehr wohl als Komponenten einschätze, dann sind diese in einer Eclipse-Umgebung nur für den vorgesehene Einsatz brauchbar. Solche Plugins werden niemals in einem anderen Kontext verwendet. Bei Eclipse-Plugins geht es vielmehr um eine spezifische Erweiterung der Eclipse-Plattform eines individuellen Benutzers. Auch für ein Servlet oder eine EJB-Komponente ist eine Wiederverwendung in einer anderen Anwendung nicht unbedingt typisch. Weitere in den Zitaten erwähnte Merkmale von Komponenten, die ich nicht für wesentlich halte, sind, dass sie vermarktbar sein sollen oder dass eine „composition by third parties“ erfolgen soll. Ein zusätzlicher Mangel einiger der oben angeführten Aussagen über Komponenten ist ihre Unschärfe. Es werden teilweise Begriffe verwendet, die Raum für weite Interpretationen geben. So wird davon gesprochen, dass Komponenten abgeschlossen sind oder eine „more or less opaque boundary“ haben, was immer das auch bedeuten mag. Ebenso weit interpretierbar ist die Angabe, dass Komponenten „komplexes Verhalten transparent kapseln“.

Die Schwierigkeit bei der Definition von Java-Komponenten liegt darin, dass die Definition allgemein genug sein soll, damit alle diejenigen Systeme, die in diesem Buch vorgestellt werden und gemeinhin als Java-Komponentensysteme gelten, unter die Definition fallen. Wenn man ein Komponentensystem (beispielsweise das prototypische Framework aus dem vorigen Kapitel) als Vorlage für eine enge Definition heranziehen würde, dann würden andere Frameworks eventuell nicht mehr als Komponentensysteme zählen, obwohl die Allgemeinheit das anders sieht. Die Definition darf also einerseits nicht zu eng sein. Sie sollte aber andererseits auch nicht zu weit gefasst sein, damit nicht jedes beliebige Software-Systeme als Komponentensystem gelten kann.

Übrigens soll ganz bewusst nicht gefordert werden, dass sich alle Komponenten eines Komponentensystems in einem Prozess befinden müssen. In unserem Prototyp und auch in vielen „real existierenden" Komponentensystemen ist das zwar so, aber es gibt auch Komponentensysteme, die auf mehrere Prozesse und sogar auf mehrere Rechner verteilt sind. Genauso wenig soll gefordert werden, wie das manchmal zu lesen ist, dass eine Komponente mehr als eine Klasse umfassen muss. Das mag zwar in der Praxis in den meisten Fällen so sein, aber es sollte möglich sein, einfache Beispiele für Komponenten anzugeben, die auch nur aus einer Klasse bestehen und dennoch das Prinzip einer Komponente für ein bestimmtes Komponentenmodell illustrieren.

7.3 Eigenschaften von Java-Komponenten

Die Problematik der obigen Definitionen hängt u. a. damit zusammen, dass die Definition von Software-Komponenten sehr allgemeingültig sein soll. Die Aufgabe ist in unserem Fall allerdings etwas einfacher, da sich dieses Buch – wie der Titel angibt – auf Java-Komponenten beschränkt. Sie werden im folgenden Abschnitt auch keine klare Definition von Java-Komponenten finden, aber einige Eigenschaften, die Java-Komponenten typischerweise besitzen sollen oder können. Dabei wird nicht verlangt, dass eine Java-Komponente bzw. ein Java-Komponenten-Framework alle diese Eigenschaften besitzen muss. Außerdem besitzen einige Eigenschaften mehrere Varianten und sind somit in gewisser Weise auch wieder vage. Der Nutzen in der Angabe dieser Eigenschaften besteht allerdings darin, dass die Leserinnen und Leser in die Lage versetzt werden selbst einzuschätzen, ob eine gewisse Java-Plattform mehr oder wenig stark als komponentenbasiert bzw. komponentenorientiert charakterisiert werden kann.

Komponentensysteme und Komponenten können oder sollen im Rahmen dieses Buches die folgenden Eigenschaften E1 bis E4 aufweisen:

E1: Komponenten sind klar identifizierbare Einheiten, die konform sind zu einem vorgegebenen Komponentenmodell.

Für ein Komponentensystem ist klar definiert, was eine Komponente ist. Im einfachsten Fall kann eine Komponente ein Objekt bzw. eine Klasse sein. Häufiger bestehen Komponenten wie in unserem Prototyp aus Programmcode in Form mehrerer Class-Dateien, aus Konfigurationsinformationen in Form einer XML- oder Manifest-Datei sowie eventuell aus weiteren

Dateien wie z. B. serialisierten Java-Objekten, Dateien mit Meldungstexten in unterschiedlichen Landessprachen, Bildern, Audio- und Videodateien usw. Diese Dateien werden in der Regel zu einer einzigen Archiv-Datei zusammengepackt (typisch sind die Endungen .jar oder .zip, es kommen aber auch andere Endungen vor wie z. B. .war, .ear oder .apk). Wie die gepackte Datei aussehen muss, wird vom spezifischen Komponentensystem vorgegeben. Zu diesen Vorgaben, die das sogenannte Komponentenmodell darstellen, gehört z. B. auch, wie die Einstiegsklasse aussehen muss (ob sie z. B. eine bestimmte Schnittstelle implementieren oder aus einer bestimmten Klasse abgeleitet sein muss). Das Komponentenmodell kann neben strengen Vorgaben auch Möglichkeiten anbieten, die eine Komponenten nutzen kann, aber nicht muss (z. B. dass Methoden mit bestimmten Annotationen gekennzeichnet werden können, aber nicht müssen). Eine Komponente bildet eine Installationseinheit, wobei in der Regel der Einsatz nur eines Teils einer Komponente keinen Sinn macht oder technisch nicht durchführbar ist. Es ist möglich, dass mehrere Versionen einer Komponente in einem laufenden System gleichzeitig existieren.

E2: Es existiert ein Mechanismus zur Kopplung von Komponenten durch ein Komponenten-Framework. Dadurch wird das Modularitätsprinzip unterstützt, so dass Komponenten leicht hinzugefügt, entfernt und ausgetauscht werden können (nach Möglichkeit im laufenden Betrieb).

Komponenten können kombiniert werden, wobei die Kombinationsvorschrift nicht in den Komponenten explizit ausprogrammiert sein soll, sondern außerhalb des Komponenten-Codes durch ein Komponenten-Framework erbracht wird. Beispielsweise können Komponenten grafisch-interaktiv durch das „Strippenziehen“ mit der Maus kombiniert werden (z. B. Java Beans). Eine andere Form ist die Festlegung der Beziehungen zwischen Komponenten in Form einer XML-Datei oder durch Annotationen (s. Abschnitt 3.6 über Dependency Injection). Auch die Registratur, die in unserem Prototyp eingesetzt wird, soll dazu zählen, wobei ich diese Art der Komponentenkombination so einschätze, dass die Eigenschaft E2 dadurch nur schwach vorhanden ist, da zwar die Registratur vom Komponenten-Framework bereitgestellt wird, aber die Komponenten durch das An- und Abmelden sowie das Abrufen von Objekten in ihrem Programmcode selbst bei der Verschaltung der Komponenten intensiv mithelfen müssen. Durch dieses Kopplungsprinzip soll es möglich sein, dass Komponenten anders kombiniert werden können, ohne dass dazu etwas an den Komponenten verändert wird. Insbesondere soll man Komponenten in einfacher Weise hinzufügen, entfernen und austauschen können. Besonders flexibel ist ein Komponentensystem, wenn dies im laufenden Betrieb möglich ist.

E3: Das Komponenten-Framework realisiert einen Lebenszyklus für seine Komponenten und stellt weitere Dienste für sie bereit.

In der Regel werden die Objekte, zumindest die „Hauptobjekte“ einer Komponente, nicht im Programmcode der Komponente erzeugt, sondern durch das Komponenten-Framework. Weiterhin ruft das Framework beim Eintritt weiterer Ereignisse Methoden auf den Objekten der Komponenten auf. Dabei handelt es sich häufig um Initialisierungsmethoden, wobei sich die Frage stellt, warum ein Objekt einer Komponente die Initialisierung nicht in seinem Konstruktor durchführen sollte, sondern in einer separaten Initialisierungsmethode. Der Grund ist darin zu sehen, dass das Framework nach dem Erzeugen eines Objekts eventuell

noch bestimmte Manipulationen an dem erzeugten Objekt vornimmt. Typisch wäre z. B. die Zuweisung von Referenzen auf andere vom Framework erzeugte oder bereitgestellte Objekte an die Attribute des Objekts z. B. durch Dependency Injection. Erst danach soll sich das Objekt einer Komponente dann anwendungsabhängig initialisieren. Aus diesem Grund wird die Initialisierung häufig in separaten Initialisierungsmethoden durchgeführt, die vom Framework aufgerufen werden. Symmetrisch dazu ruft das Framework auch Methoden auf, wenn eine Komponente deinstalliert wird. Dies haben wir in Form der Start- und Stop-Methoden im Prototypsystem schon gesehen. Dazwischen können weitere Aufrufe vom Komponenten-Framework erfolgen, z. B. kurz bevor ein Objekt aus Performance-Gründen vom Komponenten-Framework ausgelagert und passiv gesetzt wird sowie kurz nachdem ein Objekt wieder aktiv wurde. Welche Methoden bei welchem Ereignis aufgerufen werden, kann durch die Methoden einer Schnittstelle oder einer Basisklasse festgelegt sein, die von der Komponentenklasse implementiert bzw. überschrieben werden. Eine andere Möglichkeit ist das Annotieren von Methoden. In der Regel können bestimmte Methoden nur aufgerufen werden, wenn sich ein Objekt in einem bestimmten Zustand befindet. Die Zustandswechsel werden in der Regel durch ein Zustandsdiagramm beschrieben. Darin ist festgelegt, welche Methoden in welchem Zustand vom Framework aufgerufen werden (z. B. kann die Methode, dass ein Objekt wieder aktiv geworden ist, nicht aufgerufen werden, wenn das Objekt schon aktiv ist). Man bezeichnet die möglichen Zustandsänderungen, die durch das Zustandsdiagramm dargestellt werden, auch als den Lebenszyklus eines Objekts. Die Methodenaufrufe im Rahmen des Lebenszyklus werden vom Framework auf die Objekte der Komponenten durchgeführt (in Bild 6.1 sind das Aufrufe von unten nach oben; sie werden auch als Upcalls oder Callbacks bezeichnet). Umgekehrt ist es auch möglich, dass die Komponenten Dienste, die vom Framework zur Verfügung gestellt werden, nutzen. Zu diesen Diensten kann beispielsweise ein von allen Komponenten gemeinsam nutzbarer Speicherbereich gehören.

E4: Eine Komponente definiert explizit, was sie für andere Komponenten bereitstellt und was sie von anderen Komponenten benötigt.

Eine Komponente kann vom Vorhandensein anderer Komponenten abhängig sein (d. h. eine Komponente nutzt andere Komponenten) und kann eine Plattform für andere Komponenten bieten (d. h. eine Komponente wird von anderen Komponenten genutzt). Wenn jede Komponente explizit angibt, was sie für andere Komponenten bereitstellt und was sie von anderen Komponenten erwartet, dann kann bei der Kombination von Komponenten bereits überprüft werden, ob die Kombination erfolgreich durchgeführt werden kann oder nicht. Die Angaben können in Form von Komponenten, Packages, Klassen oder eindeutigen Bezeichnern erfolgen. Im prototypischen Komponenten-Framework wurde die Abhängigkeit durch die Angabe der Komponente ausgedrückt. Die Eigenschaft E4 steht in einem gewissen Spannungsverhältnis zur Eigenschaft E2. Je spezifischer eine Abhängigkeit ausgedrückt wird, umso weniger frei sind Komponenten kombinierbar.

7.4 Beispiele und Gegenbeispiele für Komponentensysteme

7.4.1 Beispiele aus dem Java-Umfeld

Der dritte Teil dieses Buchs enthält eine ganze Reihe von Beispielen, die mehr oder weniger als Komponentensysteme gelten können. Dazu gehören z. B. Eclipse, Applets, Servlets, Enterprise Java Beans und Spring. Auf diese Beispiele soll hier nicht weiter eingegangen werden, da sie im Folgenden ausführlicher besprochen werden.

Als ein Beispiel für ein Java-Komponentensystem betrachten wir nochmals unser prototypisches System aus dem vorhergehenden Kapitel und prüfen, inwiefern dieses System die Eigenschaften E1 bis E4 tatsächlich besitzt:

- Zu E1: Es sollte gerade für das Prototypsystem deutlich geworden sein, was eine Komponente ist und wie das Komponentenmodell dafür aussieht. Dazu gehört das Verpacken der benötigten Dateien in eine Jar- oder Zip-Datei mit der vorgegebenen Verzeichnisstruktur, die Manifest-Datei mit ihren möglichen Einträgen („Main: …“ und „Uses: …“) sowie die mit @Start und @Stop annotierten Methoden, die die Einstiegsklasse haben kann, aber nicht haben muss.
- Zu E2: Das Modularitätsprinzip ist sehr gut erfüllt, da Komponenten im laufenden Betrieb hinzugefügt, entfernt und erneuert werden können. Eine Schwachstelle des prototypischen Frameworks stellt der Kopplungsmechanismus für Komponenten dar. Hier stellt das Framework zwar eine Registratur bereit. Für die Nutzung der Registratur sind allerdings die Komponenten selbst verantwortlich. Das heißt also, dass ein „Verdrahten“ der Komponenten komplett durch das Framework und von außerhalb der Komponenten nicht möglich ist.
- Zu E3: Durch die Start- und Stop-Methoden wird ein einfacher Lebenszyklus für die „Hauptobjekte“ durch das Framework realisiert. Das Framework stellt auch einen einfachen Dienst in Form einer Registratur für ihre Komponenten bereit.
- Zu E4: Eine Komponente kann und muss nicht explizit spezifizieren, was sie bereitstellt. Grundsätzlich können die öffentlichen Klassen und Schnittstellen von anderen Komponenten genutzt werden. Welche Objekte durch die Registratur zur Verfügung gestellt werden, ist im Code verborgen und wird nicht deklarativ angegeben (z. B. in einer Konfigurationsdatei oder durch eine Annotation). Wenn es also auch keine expliziten Angaben für den Export gibt, so muss eine Komponente für den Import allerdings eine entsprechende „Uses:“-Zeile in seiner Manifest-Datei haben. Das Importieren muss also explizit definiert werden.

Die Diskussion zeigt, dass unser Prototyp-Framework nicht alle Eigenschaften E1 bis E4 voll und ganz erfüllt. Die Mehrzahl der Eigenschaften wird aber erfüllt, so dass das Framework aus dem vorigen Kapitel durchaus als Beispiel eines Komponentensystems gelten kann, was nicht überraschend ist, denn andernfalls wäre es im vorigen Kapitel des Buches nicht vorgestellt worden, zumindest nicht in dieser Form.

7.4.2 Gegenbeispiele aus dem Java-Umfeld

Wenn man sich ausführbare Jar-Dateien ansieht, dann sehen diese in ihrer Struktur den Beispielkomponenten des Prototyp-Frameworks sehr ähnlich. Neben den Class-Dateien enthalten sie eine Manifest-Datei, in der durch eine „Main-Class:"-Zeile die Einstiegsklasse definiert wird, in der sich die statische Main-Methode befindet. Das Betriebssystem und die JVMs (Java Virtual Machines) könnte man als das zugrundeliegende Framework betrachten, das gewisse Dienste bereitstellt. Sogar eine Art Lebenszyklus ist vorhanden, denn so wie die Main-Methode der Einstiegsklasse zu Beginn aufgerufen wird, ist es auch möglich einzustellen, dass Methoden kurz vor dem Ende des Prozesses aktiv werden. Insofern gibt es Hinweise auf das Vorhandensein eines Komponentensystems. Was aber komplett fehlt, ist ein Kopplungsmechanismus. Auch werden bereitgestellte Komponenten nicht durch das Framework gestartet. Aus diesen Gründen würde ich ausführbare Jar-Dateien nicht als Komponenten einschätzen. Wenn man den Pipe-Mechanismus von Unix jedoch mit einbezieht, komme ich zu einer anderen Wertung (s. Abschnitt 7.4.3).

Ein weiteres Beispiel, das betrachtet werden soll, sind verteilte RMI-Anwendungen. RMI steht für Remote Method Invocation. Mit RMI wird eine Registratur namens RMI Registry bereitgestellt, die sehr ähnlich funktioniert wie die Registratur in unserem Prototyp-System. Es können Objekte an- und abgemeldet sowie abgerufen werden. Das Besondere daran ist, dass ein angemeldetes Objekt, das sich in einem Java-Prozess befindet, von einem anderen Java-Prozess aus genutzt werden kann. Die Prozesse können sich sogar auf unterschiedlichen Rechnern befinden, was das R in RMI erklärt. Man könnte nun versucht sein, die einzelnen Java-Programme, die über RMI kooperieren, als Komponenten zu sehen. In der Tat besitzt ein solches System eine gewisse Modularität, denn es können einfach einzelne Komponenten hinzugefügt, entfernt und neu gestartet werden, ohne dass andere betroffen sind, und dies sogar im laufenden Betrieb. Der Kopplungsmechanismus für diese Komponenten ist RMI. Dieser ist zwar eher schwach, wie in Abschnitt 7.4.1 für das Prototyp-System erwähnt wurde, da er keine Kopplung ohne Zutun der Komponenten erlaubt. Das Komponentenmodell ist eher unscharf (jede Anwendung, die RMI nutzt). Auch wird nicht erzwungen, dass die Komponenten über RMI kommunizieren müssen. Bereitgestellte Komponenten werden nicht durch das Framework gestartet (man könnte mit dem Hinweis auf die RMI-Aktivierung gegen diese Aussage argumentieren). Ein Lebenszyklus ist schwach vorhanden (wie oben für ausführbare Jar-Dateien diskutiert). Explizite Angaben des Im- und Exports gibt es nicht (sind im Programmcode versteckt). Ich würde auch in diesem Fall eher nicht von einem Komponentensystem sprechen, denn der Erfüllungsgrad der Eigenschaften E1 bis E4 ist zwar nicht null, aber in meinen Augen nicht stark und deutlich genug. Man könnte aber auch die gegenteilige Auffassung vertreten.

7.4.3 Beispiele aus dem Nicht-Java-Umfeld

Die Verkettung über Pipes aus dem Unix-Betriebssystem wird ab und zu ebenfalls als Komponentensystem angeführt. Betrachten wir als Beispiel die folgende Zeile, die einem Unix-Kommandointerpretierer (Unix Shell) angegeben werden kann ($ soll dabei der von der Shell ausgegebene Prompt sein):

```
$ prog1 | prog2 | prog3
```

Bei prog1, prog2 und prog3 muss es sich um ausführbare Programme handeln, die über die PATH-Umgebungsvariable im Dateisystem gefunden werden.

Die Unix-Shell erzeugt für jedes auszuführende Programm einen eigenen Prozess und für jedes Pipe-Symbol | eine Unix-Pipe. Eine Pipe ist ein Kommunikationskanal, in den Daten geschrieben und aus dem diese Daten wieder gelesen werden können. Der lesende und schreibende Zugriff hat unter Umständen synchronisierende Wirkung: So wird ein Leser blockiert, wenn er aus einer leeren Pipe lesen möchte, ein Schreiber, wenn er in eine volle Pipe schreiben möchte. Die erzeugten Prozesse und Pipes werden nun so verschaltet, dass die Standardausgabe eines Prozesses auf die Pipe, die dem nachfolgenden Pipe-Symbol entspricht, „umgebogen“ wird. Das heißt, immer wenn in der Anwendung, die von diesem Prozess ausgeführt wird, etwas auf die Standardausgabe geschrieben wird (in einem Java-Programm ist das System.out), dann erscheint die Ausgabe nicht auf dem Bildschirm, sondern wird in die Pipe geschrieben. Analog wird die Standardeingabe eins Prozesses (in Java System.in) auf die Pipe umgebogen, die dem vorausgehenden Pipe-Symbol entspricht. Das, was also das Programm normalerweise von der Tastatur einlesen würde, kommt jetzt aus der Pipe.

Wie üblich können jedem ausführbaren Programm auch Kommandozeilenargumente mitgegeben werden:

```
$ prog1 arg11 arg12 | prog2 arg21 | prog3 arg31 arg32 arg33
```

Damit kann die Kopplung auch für Java-Programme vorgenommen werden:

```
$ java a.b.Main1 | java c.d.Main2 args | java e.f.g.Main3 otherargs
```

Und ebenso können Java-Programme mit Programmen, die in anderen Sprachen geschrieben wurden, gekoppelt werden:

```
$ java a.b.Main1 | prog2 | java e.f.g.Main3 arg1 arg2
```

In meinen Augen könnte man hier in der Tat von einem kleinen Komponentensystem sprechen, welches hier durch die Unix-Shell in der Rolle des Frameworks realisiert wird. Zwar ist das Komponentenmodell auch schwach ausgeprägt (eine Komponente ist eine Anwendung, die die Standardein- und/oder -ausgabe benutzt). Auch gibt es keinen Lebenszyklus und keine expliziten Angaben, was importiert und exportiert wird. Was es aber gibt (und das halte ich hier für entscheidend), ist ein Kopplungsmechanismus, der außerhalb der Komponenten allein durch das Framework realisiert wird. Dieser Mechanismus ermöglicht eine sehr einfache Kopplung der Komponenten, die leicht veränderbar ist, wenn auch nicht im laufenden Betrieb. Ich würde also sagen, dass es sich beim Pipe-Mechanismus gerade noch um ein Komponentensystem handelt. Wenn Sie zu einer anderen Bewertung kommen, dann ist das sicher auch gut vertretbar.

Weitere Komponentensysteme im Nicht-Java-Umfeld sind COM, DCOM und die .NET-Komponenten aus der Windows-Welt, das CORBA Component Model (CCM) aus der CORBA-Welt, Web Services und die Loadable Kernel Modules des Linux-Betriebssystems. Wegen des Fokus auf Java in diesem Buch gehen wir auf diese Systeme nicht weiter ein.

III Teil 3: Beispiele für Java-Komponentensysteme

In diesem dritten Teil wird eine Reihe von Beispielen für Java-Komponentensysteme präsentiert, die alle mehr oder weniger als solche klassifiziert werden können. Es handelt sich dabei um Java Beans (Kapitel 8), OSGi (Kapitel 9), Eclipse (Kapitel 10), Applets (Kapitel 11), Servlets (Kapitel 12), Enterprise Java Beans (Kapitel 13), Spring (Kapitel 14), Ereignisbusse (Kapitel 15) und Android (Kapitel 16). Damit werden alle aus meiner Sicht interessanten und relevanten Java-Komponentensysteme abgedeckt. Denkbar wäre gewesen, CORBA und Web Services noch als Komponentensysteme mit aufzunehmen. Ich habe darauf aber aus mehreren Gründen verzichtet: CORBA und Web Services sind nicht speziell auf Java zugeschnitten, sondern in gewisser Weise sprachunabhängig, ihre praktische Bedeutung ist stark rückläufig, und sie passen nur bedingt zu der in diesem Buch vertretenen Auffassung, was Komponentensysteme ausmacht. Außerdem habe ich mir die Freiheit genommen, nur solche Themen in diesem Buch zu behandeln, die mir gefallen.

8 Java Beans

Das Konzept von Java Beans ist das älteste Java-Komponentenmodell. Fast seit Beginn der Einführung von Java Mitte der 90er Jahre waren Java Beans ein Teil der Java Standard Edition (SE). Alle für die Programmierung von Java Beans nötigen Klassen und Schnittstellen befinden sich deshalb auch in der Java-Klassenbibliothek von Java SE. Java Beans sollten nicht mit Enterprise Java Beans (EJB) verwechselt werden. Ganz im Gegensatz zu den EJBs, die in Kapitel 13 ausführlicher besprochen werden, haben Java Beans heute keine besonders große Bedeutung mehr, weshalb sie in diesem Buch nur relativ knapp abgehandelt werden.

8.1 Komponentenmodell

Die grundlegende Idee von Java Beans drückt dieser Satz der Firma Sun aus: „A Java Bean is a reusable software component that can be manipulated visually in a builder tool“. Entscheidend ist also, dass ein Java Bean eine wiederverwendbare Software-Komponente ist, die durch ein entsprechendes Software-Werkzeug (Builder Tool) in visueller Form manipuliert werden kann. Das Software-Werkzeug erzeugt zu diesem Zweck interaktiv Objekte von Bean-Klassen, wobei die Bean-Klassen dem Werkzeug in geeigneter Form zur Verfügung gestellt werden. Mit Hilfe von Reflection können die Attribute der erzeugten Bean-Objekte angezeigt und interaktiv geändert werden. Das Software-Werkzeug ermöglicht weiterhin die „Verdrahtung“ der Bean-Objekte, was im Folgenden noch genauer besprochen werden wird. Ist eine Anwendung auf diese Art interaktiv „zusammengeklickt“ worden, so kann man die konfigurierten Objekte abspeichern, entweder in dem spezifischen Binärformat der Java-Serialisierung oder als XML-Datei. Die Anwendung besteht dann aus den verwendeten Bean-Klassen, den zuvor abgespeicherten Bean-Objekten sowie weiterer Software, die aus den abgespeicherten Daten wieder die entsprechenden Objekte erzeugt. Spezielle Software in Form des Builder Tools ist nur zum Erzeugen der Anwendung nötig; zur Ausführung der Anwendung genügt in der Regel die normale Java-Laufzeitumgebung.

Es kann an dieser Stelle nicht näher auf das Abspeichern und Laden von Objekten eingegangen werden. Ganz grob sei als kleiner Einschub nur so viel erwähnt: Bei der Serialisierung eines Objekts werden der Klassenname sowie die aktuellen Werte der Attribute abgespeichert. Wenn also x ein Objekt der Klasse X ist, welche beispielsweise Attribute des Typs int

und boolean besitzt, dann werden bei der Serialisierung die Werte dieser Attribute (z. B. 1147 und false) gespeichert. Da auch der Klassenname gespeichert wird, lässt sich mit Hilfe der sogenannten Deserialisierung aus diesen Daten ein neues Objekt der Klasse X erzeugen, dessen Attributen die abgespeicherten Werte zugewiesen werden, so dass zwar ein neues Objekt entsteht, dessen Inhalte aber gleich sind wie beim Originalobjekt. Das Ganze funktioniert rekursiv: Ist der Typ eines Attributs kein primitiver Datentyp wie int und boolean, sondern eine Referenztyp (d. h. eine Klasse oder Schnittstelle), so wird beim Serialisieren der Referenz nachgegangen und das Objekt, auf das diese Referenz verweist, wird ebenfalls serialisiert. Dieser Prozess kann sich über beliebig viele Stufen erstrecken. Dabei werden Objekte, die schon bearbeitet wurden, erkannt; diese werden dann nicht nochmals serialisiert. Somit wird ein Objekt nur einmal serialisiert, auch wenn mehrere serialisierte Objekte eine Referenz darauf besitzen. Insbesondere kommt es damit bei zyklischen Referenzfolgen nicht zu einer Endlosschleife bei der Serialisierung. Bei der Deserialisierung werden Kopien der Objekte erzeugt und die „Verdrahtung“ der Objekte wird wieder genau so hergestellt, wie sie ursprünglich war. Um ein Objekt mit Hilfe der Java-Serialisierung serialisieren zu können, muss seine Klasse die Schnittstelle Serializable implementieren. Serializable ist eine leere Schnittstelle (sie besitzt also keine Methoden, die implementiert werden müssen). Serializable ist lediglich eine sogenannte Markierungsschnittstelle, mit der man explizit kennzeichnen muss, dass Objekte dieser Klasse serialisierbar sein sollen. Aus den bisherigen Ausführungen ergibt sich, dass es für eine erfolgreiche Serialisierung nicht genügt, wenn die Klasse eines Objekts die Schnittstelle Serializable implementiert, sondern dies muss in rekursiver Weise für alle Objekte gelten, auf die das zu serialisierende Objekte Referenzen hält. Will man ein Attribut von der Serialisierung explizit ausnehmen, so kann man dieses durch das Java-Schlüsselwort transient entsprechend kennzeichnen.

So viel als kurzer Einschub zum Thema Serialisierung. Kommen wir nun zurück zu den Java Beans: Aufgrund der eingangs dieses Abschnitts geschilderten Grundidee zum Einsatz von Java Beans ergeben sich folgende Eigenschaften für Bean-Klassen:

- Eine Bean-Klasse muss weder eine vorgegebene Schnittstelle implementieren noch aus einer speziellen Klasse abgeleitet sein. Bean-Objekte sind also POJOs (Plain Old Java Objects).
- Damit ein Software-Werkzeug durch interaktive Angabe einer Bean-Klasse ein Bean-Objekt erzeugen kann, sollten Bean-Klassen einen parameterlosen Konstruktor besitzen.
- Um nach dem Zusammenklicken einer Anwendung aus Bean-Komponenten die Objekte abspeichern zu können, sollten alle Bean-Klassen die leere Schnittstelle Serializable (aus dem Package java.io) implementieren. Wenn die Bean-Klassen Attribute besitzen, die auf andere Objekte (Beans oder Hilfsobjekte) verweisen, so sollten die Klassen dieser Objekte ebenfalls Serializable implementieren, oder die Attribute müssen als transient gekennzeichnet sein. Dies gilt in rekursiver Form für alle referenzierten Objekte.
- Wenn es möglich sein soll, die Attributwerte eines Bean-Objekts interaktiv über ein Software-Werkzeug anzuzeigen und zu verändern, ist es ratsam, für diese Attribute entsprechende Getter- und Setter-Methoden in der Bean-Klasse bereitzustellen.
- Das Zusammenspiel unterschiedlicher Bean-Objekte kann in jeder beliebigen Form erfolgen. Wenn aber Bean-Objekte mit den Bean-spezifischen Software-Werkzeugen „verdrahtet“ werden sollen, dann müssen die Beans gemäß des Beobachter-Entwurfsmusters interagieren. Das heißt, dass es beobachtbare Objekte (auch Ereignisquellen genannt) und

Beobachter (auch Ereignisbehandler genannt) gibt. Wenn eine Bean-Klasse eine Ereignisquelle darstellt, muss sie entsprechende Methoden besitzen, um Ereignisbehandler an- und abzumelden (ein JButton von Swing ist beispielsweise eine Ereignisquelle – die Klasse JButton besitzt die Methoden addActionListener und removeActionListener zum An- und Abmelden von ActionListenern). Wenn eine Bean-Klasse einen Ereignisbehandler darstellt, muss sie eine spezifische Schnittstelle implementieren (zum Beispiel eine Klasse, die ActionListener implementiert, um auf das Drücken eines JButtons zu reagieren). Objekte einer Ereignisbehandlerklasse (Beobachter) können an einem Quellenobjekt (beobachtbares Objekt) des passenden Ereignistyps angemeldet werden. Immer wenn im Quellobjekt ein entsprechendes Ereignis erzeugt wird, werden alle angemeldeten Behandler durch Aufruf einer Methode der implementierten Schnittstelle benachrichtigt.

- Wenn Bean-Objekte parallel genutzt werden, so muss ihr Programmcode eine entsprechende Synchronisation beinhalten. Wenn man zum Zeitpunkt der Codierung einer Bean-Klasse noch nicht weiß, ob die Beans in einem parallelen Umfeld genutzt werden, so sollte man vom schlimmsten Fall ausgehen und die Synchronisation vorsorglich vorsehen.

Die Bean-Klassen und die von ihr benötigten Hilfsklassen werden in Jar-Dateien gepackt. Damit das Software-Werkzeug zur Konfiguration von Bean-Anwendungen weiß, welche der Klassen Bean-Klassen sind und welche nicht, enthält eine Jar-Datei wie in der prototypischen Implementierung von Kapitel 6 eine Manifest-Datei, in der die Bean-Klassen entsprechend markiert sind:

```
Manifest-Version: 1.0

Name: javacomp/javabeans/BeanClass1.class
Java-Bean: True

Name: javacomp/javabeans/BeanClass2.class
Java-Bean: True
```

Wichtig ist zu wissen, dass die Information, welche Klassen Bean-Klassen sind, nicht für die spätere Ausführung von Bedeutung sind, sondern nur für das Software-Werkzeug, mit dem die Bean-Anwendung in visueller, interaktiver Weise „zusammengeklickt“ wird. Die Jar-Dateien werden dem Software-Werkzeug in der Regel in einem speziell vorgegebenen Verzeichnis oder durch entsprechende Konfiguration zur Verfügung gestellt.

8.2 Gebundene Eigenschaften und Eigenschaften mit Vetomöglichkeit

Das Zusammenspiel von Beans sollte gemäß des Beobachter-Entwurfsmusters erfolgen, damit die „Verdrahtung“ interaktiv durch ein Software-Werkzeug vorgenommen werden kann. In der Beans-Bibliothek (Package java.beans) von Java SE ist ein Ereignis definiert, mit dem andere über die Änderung eines Attributwerts eines Beans benachrichtigt werden können. Das Attribut eines Beans wird im Englischen als Property bezeichnet, was man mit

Eigenschaft übersetzen kann. Entsprechend heißt die Klasse, welche das Ereignis einer Attributwertänderung repräsentiert, PropertyChangeEvent:

```
package java.beans;

public class PropertyChangeEvent
       extends EventObject implements Serializable
{
    public String getPropertyName() {...}
    public Object getOldValue() {...}
    public Object getNewValue() {...}
    ...
}
```

Beobachter (bzw. Ereignisbehandler), welche sich über die Änderung eines Attributwerts benachrichtigen lassen wollen, müssen die Schnittstelle PropertyChangeListener implementieren:

```
package java.beans;

public interface PropertyChangeListener extends EventListener
{
    public void propertyChange(PropertyChangeEvent event);
}
```

Die Klassen der beobachtbaren Objekte (Ereignisquellen) müssen zum einen Methoden zum An- und Abmelden von PropertyChangeListenern bereitstellen und zum anderen alle angemeldeten PropertyChangeListener bei der Änderung eines Attributs benachrichtigen. Man nennt solche Attribute, für die Listener bei einer Änderung benachrichtigt werden, gebundene Eigenschaften (Bound Properties). Ein Beispiel einer Bean-Klasse, die eine solche gebundene Eigenschaft besitzt, findet sich in Listing 8.1.

Listing 8.1 Bean-Klasse mit gebundener Eigenschaft

```
package javacomp.javabeans;

import java.beans.*;
import java.util.ArrayList;

public class BeanWithBoundProperty
{
    private String ourString;
    private ArrayList<PropertyChangeListener> listeners;

    public BeanWithBoundProperty()
    {
        ourString = "hello";
        listeners = new ArrayList<PropertyChangeListener>();
    }

    public String getOurString()
    {
        return ourString;
    }

    public void setOurString(String newString)
```

```
    {
        String oldString = ourString;
        ourString = newString;
        PropertyChangeEvent event = new PropertyChangeEvent(this,
                                "ourString", oldString, newString);
        for(PropertyChangeListener l: listeners)
        {
            l.propertyChange(event);
        }
    }

    public void addPropertyChangeListener(PropertyChangeListener l)
    {
        listeners.add(l);
    }

    public void removePropertyChangeListener(
                                            PropertyChangeListener l)
    {
        listeners.remove(l);
    }
}
```

Für die Eigenschaften von Java Beans gibt es auch die Vorstellung, dass vor der Änderung einer Eigenschaft Beobachter gefragt werden, ob sie mit der Änderung einverstanden sind oder ihr Veto dagegen einlegen möchten. Die befragten Beobachter müssen zu diesem Zweck die Schnittstelle VetoableChangeListener implementieren:

```
package java.beans;

public interface VetoableChangeListener extends EventListener
{
    public void vetoableChange(PropertyChangeEvent event)
           throws PropertyVetoException;
}
```

Die Entscheidung, ob ein Befragter einverstanden ist oder nicht, wird nicht durch den Rückgabewert der Methode vetoableChange mitgeteilt. Es ist vielmehr so, dass das Einverständnis durch eine ganz normale Rückkehr aus der Methode und ein Veto durch das Auslösen einer Ausnahme des Typs PropertyVetoException angezeigt wird. Man nennt Attribute eines Beans, gegen dessen Änderung andere ein Veto einlegen können, eingeschränkte Eigenschaften (Constrained Properties). Selbstverständlich kann eine Eigenschaft gleichzeitig gebunden und eingeschränkt sein. Das Beispiel aus Listing 8.1 kann man zu diesem Zweck so erweitern, dass ein weiteres Attribut des Typs ArrayList<VetoableChangeListener> hinzugefügt wird (s. Listing 8.2).

Listing 8.2 Bean-Klasse mit gebundener und eingeschränkter Eigenschaft

```
package javacomp.javabeans;

import java.beans.*;
import java.util.ArrayList;

public class BeanWithBoundAndConstrainedProperty
{
```

```
    private String ourString;
    private ArrayList<PropertyChangeListener> changeListeners;
    private ArrayList<VetoableChangeListener> vetoListeners;

    ...

    public void setOurString(String newString)
    {
        try
        {
            String oldString = ourString;
            PropertyChangeEvent event =
                      new PropertyChangeEvent(this, "ourString",
                                              oldString, newString);
            for(VetoableChangeListener l: vetoListeners)
            {
                l.vetoableChange(event);
            }
            ourString = newString;
            for(PropertyChangeListener l: changeListeners)
            {
                l.propertyChange(event);
            }
        }
        catch (PropertyVetoException e)
        {
        }
    }

    ...

}
```

Die Initialisierung des zusätzlichen Attributs im Konstruktor, die neuen Methoden zum An- und Abmelden von VetoableChangeListenern sowie die von Listing 8.1 unverändert übernommenen Methoden sind in Listing 8.2 nicht dargestellt. Die interessanteste Methode ist die Methode setOurString. Vor der Änderung wird die Methode vetoableChange bei allen angemeldeten VetoChangeListenern aufgerufen. Wenn einer davon eine Ausnahme wirft, wird die Ausführung der Methode setOurString beendet. Andernfalls wird der Attributwert geändert und alle angemeldeten PropertyChangeListener werden über diese Änderung informiert.

Im Package java.beans gibt es die Klassen PropertyChangeSupport und VetoableChangeSupport, die das An- und Abmelden sowie das Benachrichtigen der entsprechenden Listener unterstützen.

8.3 BeanInfo

Durch das Hinzufügen einer BeanInfo-Klasse kann man zusätzliche Informationen zu einer Bean-Klasse angeben. Der Name der BeanInfo-Klasse muss sich zusammensetzen aus dem Namen der Bean-Klasse, die beschrieben wird, und dem Suffix „BeanInfo“ (zur Bean-Klasse p1.p2.Example beispielsweise gehört also p1.p2.ExampleBeanInfo). Falls eine BeanInfo-

Klasse vorhanden ist, was nicht unbedingt nötig ist, muss sie die Schnittstelle BeanInfo aus dem Package java.beans implementieren. Die Schnittstelle BeanInfo besitzt u. a. die folgenden Methoden:

- Methode getIcon: Diese Methode besitzt einen Parameter des Typs int, über den angezeigt wird, ob ein Icon der Größe 16 x 16 oder 32 x 32 in Farbe oder Schwarzweiß angefordert wird. Die Methode muss dann das entsprechende Bild in Form eines Image-Objekts zurückgeben.
- Methode getPropertyDescriptors: Diese parameterlose Methode liefert ein Feld des Typs PropertyDescriptor zurück. Beim Rückgabewert null muss das Builder-Werkzeug die Eigenschaften eines Beans mit Hilfe der Reflection selbst herausfinden. Wenn man jedoch nicht möchte, dass alle Attribute über das Builder-Werkzeug gelesen und verändert werden können, dann kann man die Methode getPropertyDescriptors so programmieren, dass in dem zurückgegebenen Feld nur für diejenigen Attribute, die vom Builder-Werkzeug beachtet werden sollen, jeweils ein PropertyDescriptor-Element vorkommt. In einem PropertyDescriptor-Element können neben dem Namen und dem Typ der Eigenschaft auch die Methoden zum Lesen und Setzen des Attributwerts angegeben werden, falls diese nicht den üblichen Namenskonventionen (für das Attribut mit dem Namen xyz ist dies getXyz und setXyz) entsprechen.
- Methode getEventSetDescriptor: Diese parameterlose Methode liefert für jedes Ereignis, das von der dazugehörigen Bean-Klasse ausgelöst werden kann, ein Element in einem EventSetDescriptor-Feld zurück. Neben dem Namen und Typ des ausgelösten Ereignisses lassen sich hier u. a. die Namen der Methoden zum An- und Abmelden der Listener spezifizieren.
- Methode getMethodDescriptors: Ähnlich wie bei getPropertyDescriptors muss das Builder-Werkzeug bei einem Rückgabewert von null die auf der Oberfläche angezeigten Methoden der dazugehörigen Bean-Klasse selbst bestimmen. Wenn aber nur bestimmte Methoden über die Oberfläche des Builder-Werkzeugs sichtbar sein sollen, kann man dies erreichen, indem in dem zurückgegebenen Feld nur MethodDescriptor-Elemente für die Methoden vorkommen, die vom Software-Werkzeug angezeigt werden sollen.
- Methode getBeanDescriptor: Diese Methode gibt ein Objekt des Typs BeanDescriptor zurück. Mit einem BeanDescriptor kann beispielsweise der Name, den das Software-Werkzeug für das Bean anzeigt, festgelegt werden. Außerdem kann der BeanDescriptor einen Customizer enthalten. Das ist eine Klasse (Angabe in Form eines Class-Objekts), die verwendet wird, um die Attributwerte eines Objekts der dazugehörigen Bean-Klasse anzuzeigen und zu verändern. Damit hat der Anwendungsprogrammierer die komplette Kontrolle, wie ein Bean-Objekt über die Oberfläche beeinflusst wird.

Statt alle Methoden der Schnittstelle BeanInfo zu implementieren, kann man seine BeanInfo-Klasse aus der vorgegebenen Klasse SimpleBeanInfo ableiten. Diese Klasse implementiert BeanInfo mit Standardrückgabewerten (null). Man muss dann nur noch die Methode oder die Methoden überschreiben, für die man ein anderes Verhalten festlegen möchte.

Das Package java.beans und auch das Package java.beans.beancontext enthalten einige weitere Schnittstellen und Klassen, auf die wir allerdings im Rahmen dieser kurzen Besprechung von Java Beans nicht weiter eingehen wollen.

8.4 Software-Werkzeuge

Für die Entwicklung einer aus Beans bestehenden Anwendung braucht man nicht notwendigerweise eine visuelles Software-Werkzeug (Builder Tool). Es ist durchaus möglich, ein Programm zu schreiben, das Bean-Objekte erzeugt, deren Eigenschaften mit Hilfe der Setter-Methoden verändert und die Bean-Objekte entsprechend miteinander verknüpft, indem zum Beispiel ein Bean-Objekt bei einem anderen Bean-Objekt als Listener angemeldet wird. Eine ganz wesentliches Merkmal von Java Beans ist aber die Vorstellung, dass mit Hilfe eines Software-Werkzeugs visuell und interaktiv eine Anwendung aus vorhandenen Bausteinen zusammengebaut werden kann, ohne dass man dafür programmieren (können) muss. Es gibt zwei Sorten von Beans, welche von diesen Werkzeugen unterstützt werden:

- Die erste Bean-Sorte sind solche Beans, die als Interaktionselemente in einer grafischen Benutzeroberfläche vorkommen. Dazu zählen zum einen die Swing-Klassen wie JButton, JSlider usw., die von den Werkzeugen direkt als Bean-Klassen verwendet werden können. Zum anderen können dies aber auch selbst entwickelte Klassen sein, wenn diese aus JPanel, JButton oder einer anderen geeigneten Klasse abgeleitet sind. Mit Hilfe des Builder-Werkzeugs lässt sich damit eine Oberfläche mit „Drag and Drop“ zusammenstellen. Das Builder Tool hat somit u. a. die Funktionalität eines sogenannten GUI Builders (GUI: Graphical User Interface).
- Die zweite Sorte von Beans sind solche, die keine Interaktionselemente darstellen. Wird ein Objekt davon erzeugt und auf die Oberfläche gezogen, so zeigt das Software-Werkzeug ein Rechteck mit entsprechender Beschriftung oder, falls für die Bean-Klasse ein Icon bereitgestellt wurde (s. Abschnitt 8.3), das Icon an.

Eine zusammengestellte Anwendung kann nur aus einer der beiden Bean-Sorten bestehen oder aus beiden. Durch Reflection lassen sich die Attributwerte anzeigen und entsprechend ändern, was sich bei Beans der ersten Sorte in der Regel unmittelbar auf ihr Aussehen auswirkt (die Änderung des Textattributs eines JLabel-Objekts sieht man unmittelbar). Ebenso kann man über Reflection erkennen, welche Listener an einem Bean angemeldet werden können. Durch das Ziehen von Verbindungslinien kann ein vorhandenes Bean-Objekt als Listener bei einem anderen Bean-Objekt angemeldet werden, falls das erste Bean-Objekt die entsprechende Schnittstelle implementiert. Falls dies nicht der Fall ist, kann das Software-Werkzeug unter Umständen den dazu fehlenden Code automatisch generieren. Das Beispiel hierzu, das in der Dokumentation der Firma Sun zu Java Beans vorkommt, ist eine Anwendung, die aus drei Bean-Objekten besteht, die alle zu der ersten Bean-Sorte gehören (die also alle in einem Fenster einer grafischen Benutzeroberfläche vorkommen können): Das erste Bean ist ein Objekt einer aus JPanel abgeleiteten Klasse, die eine Animation eines Jongleurs darstellt. Diese Klasse hat Methoden zum Starten und Anhalten der Animation. Die beiden anderen Bean-Objekte sind Objekte der Klasse JButton. Es lässt sich interaktiv die Beschriftung der JButtons verändern. Der eine JButton bekommt die Beschriftung „Start“ und der andere „Stopp“. Durch das Ziehen einer Linie von einem JButton auf das Animations-Bean können die beiden miteinander verbunden werden. Zwar implementiert das Animations-Bean nicht die Schnittstelle ActionListener, um das Animations-Bean an einem JButton als ActionListener anzumelden. Aber das Software-Werkzeug fragt, welche Methode des Ziel-Beans (in diesem Fall des Animations-Beans) beim Klicken des Buttons aufgerufen werden soll. Gibt man für den Start-Button die Methode zum Starten der Animation und für den

Stopp-Button die Methode zum Anhalten der Animation an, so wird der fehlende Programmcode (Klassen, welche ActionListener implementieren und in der Methode actionPerformed die entsprechende Methode auf das Animations-Bean anwenden) vom Software-Werkzeug automatisch generiert. Das Software-Werkzeug erlaubt in der Regel auch ein Umschalten vom Design-Modus in den Testmodus, um die zusammengestellte Anwendung auszuprobieren. Ferner kann man mit Hilfe eines Menüeintrags eine Anwendung bzw. ein Applet daraus erzeugen. Dabei werden die so zusammengestellten Bean-Objekte serialisiert. In der generierten Anwendung werden diese serialisierten Objektdaten dann eingelesen und mit Hilfe der Deserialisierung werden identische Objekte wie in der Design-Phase erzeugt, die auch entsprechend vernetzt sind.

Die wichtigsten Software-Werkzeuge zur Zusammenstellung von Bean-Anwendungen sind die BeanBox oder alternativ der BeanBuilder. Beide Werkzeuge können nicht mehr von der Java-Seite heruntergeladen werden. Sie sind im Internet nur noch schwer oder gar nicht mehr zu finden.

8.5 Bewertung

Da wie erwähnt das Java-Beans-Konzept das älteste Komponentenmodell ist und sich die Auffassung davon, was Komponenten sind, seither doch etwas geändert hat, ist es teilweise schwierig, die Vorgaben für ein Komponentensystem aus Kapitel 7 überhaupt auf Java Beans anzuwenden. Insbesondere gibt es zum einen die Schwierigkeit, dass nicht klar ist, ob als Komponente eine Bean-Klasse oder ein Bean-Objekt verstanden werden soll. Ich vermute, dass sich die Urheber der Java-Beans-Idee eher Bean-Objekte als Komponenten vorgestellt haben (obwohl ich das nirgends explizit gelesen habe). Wir werden im Folgenden jedoch abwechselnd beide Sichtweisen einnehmen, um einen möglichst großen Bezug zu den Merkmalen E1 bis E4 herstellen zu können. Zum anderen gibt es bei der Bewertung die Schwierigkeit, dass es überhaupt kein Komponenten-Framework gibt, das zur Ausführung einer Java-Beans-Anwendung gebraucht wird. Im Vordergrund stehen stattdessen (nichtstandardisierte) Software-Werkzeuge, mit denen die Bean-Objekte einer Anwendung konfiguriert und zusammengestellt werden. Im Folgenden wird zur Merkmalsüberprüfung statt eines Komponenten-Frameworks ein entsprechendes Software-Werkzeug verwendet:

- Zu E1: Wie oben erwähnt, ist nicht ganz klar, was eine Java-Beans-Komponente ist. Es gibt aber gewisse Vorgaben bezüglich der Bean-Klassen und dem Packen von Klassen in Jar-Dateien zusammen mit einer Manifest-Datei. Diese Vorgaben spielen für die Ausführung der Anwendung allerdings keine Rolle. Sie dienen aber dem Software-Werkzeug, mit dem eine aus Komponenten bestehende Anwendung zusammengestellt wird, und das hier an die Stelle eines Komponenten-Frameworks tritt. Insofern kann man schon behaupten, dass Java-Beans-Komponenten in Form von Bean-Objekten und ihrer dazugehörigen Bean-Klassen relativ klar definiert sind und durchaus einem vorgegebenen Komponentenmodell entsprechen. Dass man für die Bean-Klassen hier relativ viele Freiheiten hat, muss man übrigens nicht als Nachteil werten. Im Gegenteil: Je weniger Vorgaben eine Entwicklerin einhalten muss, umso mehr Freiraum hat sie bei der Programmierung, was durchaus positiv zu sehen ist.

- Zu E2: Die Kopplung der Java-Beans-Komponenten erfolgt zwar nicht über ein Komponenten-Framework zur Laufzeit, aber vor der Ausführung über ein interaktives Software-Werkzeug. Die Objektkonfiguration wird abgespeichert und zur Laufzeit wieder hergestellt. Somit lässt sich auch für Java Beans ein Kopplungsmechanismus identifizieren. Das Austauschen von Komponenten ist allerdings nur in der Konfigurationsphase, nicht aber zur Laufzeit möglich.
- Zu E3: Die in der Konfigurationsphase abgespeicherten Objektdaten werden zur Laufzeit eingelesen. Auf diese Art werden durch die Deserialisierung die Komponentenobjekte quasi automatisch und ohne Zutun des Anwendungsprogrammierers erzeugt. Die Realisierung eines echten Lebenszyklus ist für die Komponenten nicht zu erkennen. Statt den Diensten eines Komponenten-Frameworks gibt es die Funktionen, die von den unterschiedlichen Software-Werkzeugen bereitgestellt werden. Somit gibt es auch in diesem Punkt eine zumindest teilweise Übereinstimmung mit den Vorgaben.
- Zu E4: Mit Hilfe der BeanInfo-Klassen kann man für die Bean-Klassen explizit festlegen, welche Eigenschaften, Ereignisse und Methoden über das Software-Werkzeug angezeigt werden. Dies kann man durchaus als eine explizite Angabe dessen sehen, was eine Komponente zur Verfügung stellt. Fehlen diese Angaben, so lassen sie sich automatisch mit Hilfe von Reflection gewinnen. Was eine Komponente benötigt, wird allerdings nicht explizit angegeben.

Wie bereits erwähnt, passen die Vorgaben aus Kapitel 7 nicht so ganz zu Java Beans. Mit einigen Umdeutungen wurde es aber doch möglich, so viele Übereinstimmungen zwischen den charakteristischen Merkmalen der Java Beans und den Vorgaben zu finden, dass das Konzept von Java Beans – wenn auch mit Vorbehalt – als Komponentensystem eingestuft werden kann und die Besprechung von Java Beans in diesem Buch gerechtfertigt erscheint.

9 OSGi

OSGi stand ursprünglich für „Open Services Gateway initiative“. Heute passt diese Bedeutung nicht mehr zu OSGi. OSGi steht deshalb für sich selbst, d. h. es ist keine Abkürzung mehr, sondern lediglich ein Name. OSGi basiert auf einer 1999 gegründeten Firmenallianz mit namhaften Firmen wie z. B. Oracle/Sun, IBM, Siemens usw. Mit OSGi sollte ein Komponentenmodell für Java-Anwendungen standardisiert werden, das es ermöglichen soll, Komponenten dynamisch (d. h. zur Laufzeit) zu einer Anwendung hinzuzufügen und wieder zu entfernen. Gedacht war es als Plattform für alle Arten von Anwendungen, besonders solche, die auf spezieller Hardware laufen, also für sogenannte eingebettete Systeme. Anwendungsbereiche dafür sind beispielsweise Fahrzeuge, Mobiltelefone, die Unterhaltungselektronik, die Gebäudeautomatisierung und intelligente Assistenzsysteme (z. B. im Pflegesektor und in der Medizin). Die Entwicklungsplattform Eclipse, die wir im nächsten Kapitel behandeln, basiert mittlerweile ebenfalls auf OSGi.

Die OSGi-Spezifikation 4.1 wurde im Rahmen des JCP (Java Community Process) als dynamisches Komponentenmodell für Java unter dem Namen „Dynamic Component Support for Java SE“ angenommen. Es ist also mittlerweile das offizielle Komponentenmodell von Java. Auch für dieses Buch hat OSGi zentrale Bedeutung: Im zweiten Teil des Buches sind sowohl das prototypische Komponentensystem als auch die typischen Eigenschaften E1 bis E4 von Komponentensystemen stark von OSGi beeinflusst.

Implementierungen von OSGi sind u. a. Equinox (von Eclipse), Knopflerfish und Felix (früher Oscar). In diesem Kapitel verwenden wir Felix als Plattform für OSGi.

9.1 Komponentenmodell

In OSGi wird eine Komponente Bundle genannt. Die Dateien eines Bundles werden in eine Jar-Datei gepackt. Dies sind im Wesentlichen der Programmcode in Form von Class-Dateien sowie eine Konfigurationsdatei in Form einer Manifest-Datei. In der Jar-Datei befinden sich auf der obersten Ebene die Verzeichnisse des jeweils ersten Namensteils der vorhandenen Packages. In unserem Fall werden für die Komponenten die Package-Namen javacomp.osgi.bundle1, javacomp.osgi.bundle2 usw. verwendet. Also befindet sich in der Jar-Datei für den

Code ein Verzeichnis namens javacomp. Außerdem gibt es ein Verzeichnis META-INF, in dem sich eine Datei namens MANIFEST.MF befindet. Die Manifest-Datei enthält mehrere Zeilen der Form „Schlüsselwort: Wert“. Beispiele für Schlüsselwörter sind:

```
Bundle-Name: ...
Bundle-SymbolicName: ...
Bundle-Description: ...
Bundle-Vendor: ...
Bundle-Activator: ...
Bundle-Version: ...
Import-Package: ...
Export-Package: ...
```

Über Bundle-Name, Bundle-Symbolic-Name, Bundle-Description und Bundle-Vendor kann die betreffende Komponente beschrieben werden. Bundle-Version, Import- und Export-Package werden später bei der Besprechung der Beispiele noch erläutert. Durch Bundle-Activator wird die Einstiegsklasse bestimmt, von der das OSGi-Framework ein Objekt erzeugt. Die Einstiegsklasse muss die Schnittstelle BundleActivator (s. Listing 9.1) implementieren.

Listing 9.1 OSGi-Schnittstelle BundleActivator

```
package org.osgi.framework;
public interface BundleActivator
{
    public void start(BundleContext context) throws Exception;
    public void stop(BundleContext context) throws Exception;
}
```

Wie zu vermuten ist, ruft das Framework die Start-Methode nach der Erzeugung eines Objekts der Einstiegsklasse auf. Dies passiert immer dann, wenn die Komponente installiert oder das Framework mit bereits installierter Komponente gestartet wird. Analog dazu wird stop aufgerufen, wenn die Komponente deinstalliert oder das Framework heruntergefahren wird. BundleContext ist eine Schnittstelle, über die das Bundle-Objekt Zugriff auf die Dienste des Frameworks erhält. Zu diesen Diensten gehören eine Registratur zum An- und Abmelden von Dienstobjekten sowie zum Abfragen angemeldeter Objekte. Es gibt verschiedene Ereignisse, auf die ein Bundle reagieren kann. BundleContext bietet dazu die Möglichkeit, Listener für unterschiedliche Ereignisse anzumelden, die bei Eintreten des betreffenden Ereignisses aufgerufen werden. Über BundleContext kann sogar ein Bundle installiert werden. Weiterhin kann man über den BundleContext Referenzen auf Bundle-Objekte erhalten. Jedes Bundle-Objekt repräsentiert eine installierte Komponente. Von den Bundle-Objekten lassen sich Informationen über die installierten Bundles erfragen (z. B. über alle Informationen in der Manifest-Datei der Komponente). Außerdem kann man die Komponenten damit auch wieder deinstallieren. Nähere Informationen zu BundleContext und Bundle folgen im weiteren Verlauf des Kapitels.

Da die Start- und Stop-Methode vom Main-Thread des Frameworks aufgerufen werden, ist es sehr ratsam, dass diese Methoden keine lange Ausführungszeit haben (es wird andernfalls das Framework an der Ausführung weiterer Funktionen gehindert). Falls auch nach dem Starten der Komponente noch Code ausgeführt werden soll, so muss beim Starten ein Thread erzeugt werden, der spätestens in der Stop-Methode wieder angehalten werden sollte. Falls im Thread auch ein Zugriff auf BundleContext nötig ist, so ist dies möglich; dem

Thread muss dazu die Referenz auf das BundleContext-Objekt übergeben werden. Übrigens wird für jede Komponente ein eigenes BundleContext-Objekt erzeugt, das auch nur von dieser Komponente verwendet werden sollte. Es wird deshalb davon abgeraten, eine Referenz auf das eigene BundleContext-Objekt an Objekte anderer Bundles weiterzugeben.

9.2 Erstes Beispiel-Bundle

Als erstes Beispiel-Bundle betrachten wir eine Komponente, die über eine Schnittstelle (s. Listing 9.2) ein Objekt einer Zählerklasse (s. Listing 9.3) zur Verfügung stellt.

Listing 9.2 Schnittstelle Counter

```
package javacomp.osgi.bundle1;

public interface Counter
{
    public int reset();
    public int increment();
    public String version();
}
```

Listing 9.3 Klasse CounterImpl

```
package javacomp.osgi.bundle1;

public class CounterImpl implements Counter
{
    private static final int INCREMENT = 1;
    private static final String VERSION = "version " + INCREMENT;

    private int counter;

    public int reset()
    {
        counter = 0;
        return counter;
    }

    public int increment()
    {
        counter += INCREMENT;
        return counter;
    }

    public String version()
    {
        return VERSION;
    }
}
```

Um unterschiedliche Versionen einer Klasse unterscheiden zu können, gibt es Konstanten mit den Namen INCREMENT und VERSION, die von Version zu Version verändert werden können. Durch die Methode version kann die Konstante VERSION erfragt werden. Die Konstante INCREMENT spielt bei der Methode increment eine Rolle; der Zähler wird bei jedem Aufruf um INCREMENT erhöht.

Die erste Komponente enthält außerdem eine Activator-Klasse namens CounterServiceActivator (s. Listing 9.4), welche die Schnittstelle BundleActivator implementiert. In der Start-Methode wird ein Objekt der Klasse CounterImpl erzeugt und in der Registratur, die vom BundleContext-Parameter bereitgestellt wird, angemeldet.

Listing 9.4 Klasse CounterServiceActivator

```
package javacomp.osgi.bundle1;

import org.osgi.framework.*;

public class CounterServiceActivator implements BundleActivator
{
    public void start(BundleContext context) throws Exception
    {
        System.out.println("CounterServiceActivator.start");
        context.registerService(Counter.class,
                                new CounterImpl(),
                                null);
    }

    public void stop(BundleContext context) throws Exception
    {
        System.out.println("CounterServiceActivator.stop");
    }
}
```

Das Anmelden eines Objekts erfolgt über den BundleContext-Parameter. Als Suchschlüssel wird in der Methode registerService eine Klasse oder Schnittstelle angegeben. Die Methode registerService ist mehrfach überladen. Es gibt eine Variante, in der die Klasse bzw. Schnittstelle als String angegeben wird. Es kann aber nur ein String angegeben werden, der auch tatsächlich einer Klasse oder Schnittstelle entspricht. In Listing 9.4 wird die Variante von registerService verwendet, in der der Suchschlüssel durch ein Class-Objekt spezifiziert wird. Die Signatur dieser generischen Methode sieht so aus:

```
public <S> ServiceRegistration<S> registerService(Class<S> clazz,
                                   S service,
                                   Dictionary<String,?> properties)
```

Der zweite Parameter von registerService ist das anzumeldende Objekt. Suchschlüssel und anzumeldendes Objekt müssen kompatibel zueinander sein. Verwendet man die Methode, in der die Klasse oder Schnittstelle als String angegeben wird, dann fällt eine Inkompatibilität erst zur Laufzeit auf. Der Vorteil der verwendeten Variante ist, dass in diesem Fall die Inkompatibilität bereits zur Übersetzungszeit bemerkt wird. Wenn z. B. Listing 9.3 so verändert wird, dass die Klasse CounterImpl nicht mehr die Schnittstelle Counter implementiert, dann meldet der Compiler für den Aufruf von registerService in Listing 9.4 den Fehler: „The method registerService … is not applicable for the arguments …“. Mit dem dritten

Argument kann das anzumeldende Objekt durch zusätzliche Eigenschaften in Form einer Liste von Name-Wert-Paaren beschrieben werden, wobei die Namen Strings sein müssen. Wie in Listing 9.4 zu sehen kann durch den Parameterwert null auf die Angabe der Liste auch verzichtet werden. Der Rückgabewert vom Typ ServiceRegistration kann zur Änderung oder Löschung der Anmeldung verwendet werden. In Listing 9.4 wird der Rückgabewert von registerService nicht benutzt. Übrigens müssen die in der Start-Methode angemeldeten Objekte in der Stop-Methode nicht unbedingt wieder abgemeldet werden, denn das OSGi-Framework ist so komfortabel, dass es die entsprechenden Abmeldungen automatisch durchführt. Aus diesem Grund hat die Stop-Methode in Listing 9.4 keine Funktion zu erbringen. Es erfolgt wie in der Start-Methode lediglich eine Bildschirmausgabe, um nachverfolgen zu können, ob und wann die Methoden vom Framework aufgerufen werden.

Damit liegt der Code der ersten Komponente mit Counter, CounterImpl und CounterServiceActivator vor. Was noch fehlt, ist die Manifest-Datei. Neben rein beschreibenden Zeilen spielen vor allem die Angaben über die Activator-Klasse, den Im- und Export eine wichtige Rolle. Die folgende Manifest-Datei wird für die erste Beispielkomponente verwendet:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: CounterService
Bundle-SymbolicName: CounterService
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-Vendor: Rainer Oechsle
Bundle-Activator: javacomp.osgi.bundle1.CounterServiceActivator
Import-Package: org.osgi.framework
Export-Package: javacomp.osgi.bundle1
```

Durch Bundle-Activator wird die Activator-Klasse (also die Einstiegsklasse) festgelegt. Wird die Zeile weggelassen, so ist dies kein Fehler; eine Activator-Klasse ist nicht zwingend notwendig. In diesem Fall kann man an den Ausgaben erkennen, dass die Start- und Stop-Methoden nie aufgerufen werden. Mit Import-Package muss man die Packages angeben, die nicht Teil der Komponente sind, aber von der Komponente genutzt werden. In unserem Beispiel ist dies das Package org.osgi.framework, in dem sich die Schnittstellen BundleActivator und BundleContext befinden. Im Allgemeinen können beliebig viele Packages angegeben werden, die durch Kommas voneinander getrennt werden müssen. Wenn die Zeile Import-Package entfernt wird, gibt es einen Fehler beim Laden der Komponente („NoClassDefFoundError: org/osgi/framework/BundleActivator“). Umgekehrt werden mit Export-Package die eigenen Packages aufgelistet, die von anderen Komponenten durch Import-Package genutzt werden können. Im Allgemeinen können dies auch mehrere sein. In unserem Fall hat die Komponente nur ein einziges Package, also können auch nicht mehr exportiert werden. Wenn die Zeile Export-Package fehlen würde, würde dies zunächst nicht auffallen. Erst wenn wir in der zweiten Beispielkomponente die erste Komponente nutzen wollen, würden wir Probleme bekommen.

Die vorhandenen Klassendateien und die Manifest-Datei können wir nun zu einer Komponente (Bundle) bündeln und installieren. Als Framework nutzen wir die OSGi-Implementierung von Apache Felix, die in Form einer Jar-Datei (felix.jar) von der entsprechenden Web-Seite bezogen werden kann. Nach dem Starten von Felix (Kommando „java -jar bin\felix.jar“) können unterschiedliche Kommandos eingegeben werden (durch den Prompt „g!“ zeigt

Felix an, dass eine Eingabe entgegengenommen werden kann). Im Folgenden wird gezeigt, wie Felix gestartet wird, wie dann unsere Komponente installiert und anschließend gestartet wird. Dazwischen wird immer wieder das Kommando lb (list bundles) aufgerufen, das alle momentan installierten Komponenten u. a. mit ihrer Kennung und ihrem Status anzeigt:

```
$ java -jar bin\felix.jar
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (4.0.2)
  224|Active     |    1|Apache Felix File Install (3.2.0)
  229|Active     |    1|Apache Felix Bundle Repository (1.6.6)
  230|Active     |    1|Apache Felix Gogo Command (0.12.0)
  231|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
  232|Active     |    1|Apache Felix Gogo Shell (0.10.0)
g! install file:/Users/oechsle/Desktop/felix/bundle1.zip
Bundle ID: 280
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (4.0.2)
  224|Active     |    1|Apache Felix File Install (3.2.0)
  229|Active     |    1|Apache Felix Bundle Repository (1.6.6)
  230|Active     |    1|Apache Felix Gogo Command (0.12.0)
  231|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
  232|Active     |    1|Apache Felix Gogo Shell (0.10.0)
  280|Installed  |    1|CounterService (1.0.0)
g! start 280
CounterServiceActivator.start
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (4.0.2)
  224|Active     |    1|Apache Felix File Install (3.2.0)
  229|Active     |    1|Apache Felix Bundle Repository (1.6.6)
  230|Active     |    1|Apache Felix Gogo Command (0.12.0)
  231|Active     |    1|Apache Felix Gogo Runtime (0.10.0)
  232|Active     |    1|Apache Felix Gogo Shell (0.10.0)
  280|Active     |    1|CounterService (1.0.0)
g!
```

Wie zu sehen ist, zeigt die erstmalige Ausführung des Kommandos lb einige vorhandene Felix-Komponenten an, wobei die Komponente mit dem Namen „System Bundle“ keine echte Komponente, sondern das Framework selbst ist. Jede Komponente besitzt eine eindeutige Kennung, die fortlaufend vergeben wird. Nach dem Installieren unserer ersten Beispielkomponente gibt das Felix-Framework die Kennung für das neu installierte Bundle aus (in unserem Beispiel ist das 280). Die zweite Ausführung von lb zeigt, dass die neue Komponente existiert. Im Gegensatz zu allen anderen Komponenten ist unsere Beispielkomponente aber nicht im Zustand Active, sondern im Zustand Installed. Das heißt, dass von der Activator-Klasse noch kein Objekt erzeugt und auch die Start-Methode auf dieses Objekt noch nicht angewendet wurde. Dies erfolgt erst durch das Kommando start, wobei man dazu die Kennung des Bundles angeben muss, das gestartet werden soll. Wie man oben sehen kann, bewirkt die Eingabe des Start-Kommandos eine Ausgabe der Meldung, die von der Start-Methode unserer Komponente erzeugt wird. Offensichtlich ist nun also ein Objekt

erzeugt und die Start-Methode aufgerufen worden. Eine nochmalige Ausführung von lb zeigt, dass sich damit auch unsere Beispielkomponente im Zustand Active befindet.

Auf die Zustände einer Komponente (Installed, Active usw.) wird später noch genauer eingegangen. Dasselbe gilt für die Versionsnummern, die jeweils am Ende der von lb ausgegebenen Zeilen in Klammern stehen. Auf die Spalte Level wollen wir in diesem Buch nicht eingehen.

Ich möchte noch auf eine Problematik hinweisen, die ich aus eigener leidvoller Erfahrung kenne. Auch am Ende der letzten Zeile der Manifest-Datei muss ein Newline-Zeichen stehen. Ist dieses nicht vorhanden, was bei einer Betrachtung mit einem Editor in der Regel nicht auffällt, so ist die Wirkung so, als gäbe es die Zeile nicht. In diesem speziellen Beispiel betrifft dies die Export-Package-Zeile, was für die erste Komponente – wie oben bereits erwähnt – kein Problem darstellt. Wenn man aber in der zweiten Komponente die erste Komponente nutzen möchte und dieses auch nach vielen Versuchen immer noch nicht funktioniert, ist es unter Umständen nicht leicht, dies auf das Fehlen des Newline-Zeichens in der Manifest-Datei des ersten Bundles zurückzuführen.

9.3 Zweites Beispiel-Bundle

Die zweite Komponente beschafft sich in ihrer Start-Methode aus der Registratur das von der ersten Komponente angemeldete Objekt und startet einen Thread, der dieses Objekt wiederholt nutzt. In der Stop-Methode wird der Thread wieder angehalten (s. Listing 9.5).

Listing 9.5 Klassen CounterClientActivator und CounterThread

```
package javacomp.osgi.bundle2;

import javacomp.osgi.bundle1.*;
import org.osgi.framework.*;

public class CounterClientActivator implements BundleActivator
{
    private ServiceReference<Counter> ref;
    private CounterThread thread;

    public void start(BundleContext context) throws Exception
    {
        System.out.println("CounterClientActivator.start");
        ref = context.getServiceReference(Counter.class);
        if(ref != null)
        {
            Counter counter = context.getService(ref);
            if(counter != null && thread == null)
            {
                thread = new CounterThread(counter);
                thread.start();
            }
    }

    public void stop(BundleContext context) throws Exception
```

```
    {
        System.out.println("CounterClientActivator.stop");
        if(thread != null)
        {
            thread.interrupt();
            thread = null;
        }
        if(ref != null)
        {
            context.ungetService(ref);
            System.out.println("   with ungetService");
        }
    }
}

class CounterThread extends Thread
{
    private Counter counter;

    public CounterThread(Counter counter)
    {
        this.counter = counter;
    }

    public void run()
    {
        try
        {
            while(!isInterrupted())
            {
                String output = counter.version() + " (" +
                                counter.increment() + ")";
                System.out.println(output);
                sleep(2000);
            }
        }
        catch(InterruptedException e)
        {
        }
    }
}
```

Das Beschaffen des Objekts aus der Registratur erfolgt in zwei Schritten, wobei dazu die zwei generischen Methoden getServiceReference und getService der Schnittstelle BundleContext verwendet werden:

```
public <S> ServiceReference<S> getServiceReference(Class<S> clazz);
public <S> S getService(ServiceReference<S> reference);
```

Als Parameter von getServiceReference gibt man die Klasse oder Schnittstelle an, nach der gesucht wird. Die Angabe kann in Form eines Strings oder in Form eines Class-Objekts erfolgen (oben ist die Signatur für die zweite Variante gezeigt). Zurückgegeben wird nicht das Objekt, sondern ein typparametrisiertes ServiceReference-Objekt. Wenn man dieses ServiceReference-Objekt als Parameter beim Aufruf von getService angibt, erhält man das in der Registratur angemeldete Objekt zurück. Wie in Listing 9.5 zu sehen ist, muss man das von getService gelieferte Objekt nicht auf Counter casten, da man als Suchbegriff die

Schnittstelle Counter angegeben hat und der Typ Counter aufgrund der Generics-Regeln damit der Rückgabetyp dieses Aufrufs von getService ist. Über das ServiceReference-Objekt kann man sich auch Zugriff auf die Liste von Name-Wert-Paaren beschaffen, die als drittes Argument beim Aufruf von registerService angegeben werden kann (s. Abschnitt 9.2). Das ServiceReference-Objekt kann auch als Parameter in der Methode ungetService benutzt werden, um dem Framework mitzuteilen, dass ein mit getService beschafftes Dienstobjekt nicht länger benutzt wird. Davon wird in der Stop-Methode der Klasse CounterClientActivator Gebrauch gemacht. Damit dies möglich ist, wird das in der Start-Methode beschaffte ServiceReference-Objekt als Attribut gespeichert. Die Thread-Klasse sollte ohne weiteren Erklärungen verständlich sein.

Die folgende Manifest-Datei wird für die zweite Komponente benutzt:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: CounterClient
Bundle-SymbolicName: CounterClient
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-Vendor: Rainer Oechsle
Bundle-Activator: javacomp.osgi.bundle2.CounterClientActivator
Import-Package: org.osgi.framework, javacomp.osgi.bundle1
```

Die zweite Komponente stellt keine Packages für andere Komponenten bereit. Deshalb sucht man nach einer Export-Package-Zeile vergebens. Allerdings werden zwei Packages importiert: wieder das Package org.osgi.framework des OSGi-Frameworks sowie das von der ersten Komponente exportierte Package.

Die folgenden Zeilen geben den Ablauf wieder, in dem die zweite Komponente ausprobiert wurde:

```
g! install file:/Users/oechsle/Desktop/felix/bundle2.zip
Bundle ID: 281
g! start 281
CounterClientActivator.start
g! version 1 (1)
version 1 (2)
version 1 (3)
version 1 (4)
stversion 1 (5)
oversion 1 (6)
p version 1 (7)
2version 1 (8)
81version 1 (9)

CounterClientActivator.stop
   with ungetService
g! lb
START LEVEL 1
   ID|State      |Level|Name
    0|Active     |    0|System Bundle (4.0.2)
...
  280|Active     |    1|CounterService (1.0.0)
  281|Resolved   |    1|CounterClient (1.0.0)
g!
```

Nach der Installation und dem Starten der zweiten Komponente erzeugt der Thread fortlaufend Ausgaben. Die einzelnen Zeichen des Kommandos „stop 281“ lassen sich verstreut zwischen den Ausgaben des Threads finden. Sobald das Kommando durch Drücken der Return-Taste abgeschlossen wird, wird die Stop-Methode der Komponente aufgerufen, in der der Thread angehalten wird. Nach dem Stoppen befindet sich die zweite Komponente im Zustand Resolved, wie die Ausgabe von lb zeigt. Über die Zustände der Komponenten sprechen wir später.

Genau so, wie es als Gegenstück zum Start-Befehl den Stop-Befehl gibt, gibt es zu install auch das Kommando uninstall mit naheliegender Bedeutung. Stellen wir uns vor, wir hätten wieder die Situation, dass beide Beispielkomponenten installiert und aktiv sind (d. h. der Thread der Client-Komponente läuft momentan). Wenn nun die Server-Komponente deinstalliert wird (dazu muss das Kommando „uninstall 280“ während der laufenden Thread-Ausgaben eingetippt werden), dann würde man eventuell erwarten, dass dies irgendwelche Auswirkungen auf die Client-Komponente und ihren Thread haben muss. Dies ist aber nicht so. Denn weder kann dem Thread-Objekt die Referenz auf das CounterImpl-Objekt weggezogen werden noch wird die Klasse CounterImpl dadurch „entladen“. Somit läuft der Thread auch nach dem Deinstallieren der Server-Komponente weiter. Einen Effekt sieht man erst, wenn man die Client-Komponente wie oben anhält und dann wieder neu startet: Beim Neustart wird in der Registratur zur Klasse Counter dann kein Eintrag mehr gefunden. Unsere Client-Komponente ist so programmiert, dass in diesem Fall kein Thread erzeugt wird.

Wird dagegen die Client-Komponente während der Ausführung deinstalliert, so hält der Thread an. Dies liegt an zwei Gründen: Zum einen beinhaltet eine Deinstallation immer auch ein Stoppen der Komponente. Zum anderen wurde die Komponente so programmiert, dass in der Methode stop der Thread abgebrochen wird. Wenn dies nicht so programmiert worden wäre, dann würde der Thread auch nach der Deinstallation unserer Client-Komponente weiterlaufen.

Das Felix-Framework kann durch Drücken von Strg-C oder durch Eingabe des Kommandos „stop 0“ angehalten werden. Die Komponentenkennung 0 steht für das Framework selbst. Wird also diese „Komponente“ angehalten, endet die komplette Anwendung.

9.4 Variationen der Beispiel-Bundles

Im Folgenden wollen wir durch Variation der ersten beiden Beispiel-Bundles verhindern, dass gewisse Besonderheiten in den Beispielen fälschlicherweise verallgemeinert werden und die Leserinnen und Leser meinen, dass Bundles diese Besonderheiten notwendigerweise besitzen müssen. Als ersten Variationspunkt betrachten wir die Tatsache, dass in unseren Beispielen die Client-Komponente über eine Schnittstelle auf die Service-Komponente zugreift. Eine Schnittstelle ist nicht unbedingt notwendig. Man könnte in den beiden Beispielkomponenten auch komplett ohne die Schnittstelle Counter auskommen. In diesem Fall würde die Service-Komponente das CounterImpl-Objekt mit CounterImpl.class als erstem Parameter registrieren. Der Client würde dann auch nach CounterImpl.class suchen und würde ansonsten in seinem Programm jedes Vorkommen des Typs Counter durch

CounterImpl ersetzen. Die Beispielkomponenten würden auch in dieser Form problemlos funktionieren. Wie im folgenden Abschnitt noch deutlich werden wird, gibt es aber gute Gründe für die Verwendung einer Schnittstelle.

Eine weitere Gefahr einer falschen Verallgemeinerung besteht darin, dass man annimmt, dass das Nutzen einer Komponente immer über das Anmelden eines Objekts in der Registratur und dem Abfragen dieses Objekts durch die andere Komponente erfolgen muss. Wenn wir die Beispielkomponenten nochmals ansehen, dann stellen wir fest, dass die Client-Komponente nicht nur das von der Service-Komponente bereitgestellte Objekt, sondern auch die Schnittstelle Counter und die Klasse CounterImpl nutzt, die ebenfalls von der Service-Komponente zur Verfügung gestellt werden. Nun ist es aber auch durchaus möglich, dass die Nutzung nur auf Klassenebene erfolgt. Wir könnten unser Beispiel also z. B. wie folgt verändern: In der Manifest-Datei der Service-Komponente wird die Angabe über den Bundle-Activator gelöscht (die Klasse CounterServiceActivator kann, muss aber nicht aus dem Package entfernt werden). Dadurch wird beim Installieren und Starten der Service-Komponente kein CounterImpl-Objekt erzeugt und registriert. Die Client-Komponente kann selbst ein Objekt der Klasse CounterImpl erzeugen, wie die Klasse CounterClientActivator in Listing 9.6 zeigt.

Listing 9.6 Variante der Klasse CounterClientActivator mit Erzeugung eines CounterImpl-Objekts

```
package javacomp.osgi.bundle2;

import javacomp.osgi.bundle1.*;
import org.osgi.framework.*;

public class CounterClientActivator implements BundleActivator
{
    private CounterThread thread;

    public void start(BundleContext context) throws Exception
    {
        System.out.println("CounterClientActivator.start");
        Counter counter = new CounterImpl();
        thread = new CounterThread(counter);
        thread.start();
    }

    public void stop(BundleContext context) throws Exception
    {
        System.out.println("CounterClientActivator.stop");
        if(thread != null)
        {
            thread.interrupt();
            thread = null;
        }
    }
}

class CounterThread extends Thread
{
    //unverändert wie bisher
    ...
}
```

Wir haben also gesehen, dass eine Komponente Klassen und Objekte oder nur Klassen einer anderen Komponente nutzen kann. Der Fall, in dem nur Objekte genutzt werden, die nutzende Komponente aber die Klassen der genutzten Komponente selbst mitbringt, ist nicht möglich. Nehmen wir an, dass wir in der Manifest-Datei der zweiten Komponente die Zeile

```
Import-Package: org.osgi.framework, javacomp.osgi.bundle1
```

zu

```
Import-Package: org.osgi.framework
```

verändern und in die Jar- bzw. Zip-Datei der zweiten Komponente auch die Class-Dateien von Counter und CounterImpl aufnehmen. Der Effekt ist, dass der Aufruf

```
context.getServiceReference(Counter.class);
```

in Listing 9.5 null zurückliefert, denn wie auch in der prototypischen Implementierung eines Komponenten-Frameworks in Kapitel 6 wird jede Komponente von einem eigenen Klassenlader geladen. Folglich ist die Schnittstelle javacomp.osgi.bundle1.Counter, die mit dem Klassenlader der ersten Komponente geladen wurde und beim Anmelden im Methodenaufruf von registerService verwendet wurde, verschieden von der Schnittstelle desselben Namens, die mit dem Klassenlader der zweiten Komponente geladen wurde und als Suchschlüssel in getServiceReference verwendet wird. Dieser Unterschied erklärt, warum die Suche erfolglos ist und null zurückkommt.

9.5 Hot Deployment

Unsere beiden Beispiel-Bundles arbeiten gut zusammen, falls die Service-Komponente vor der Client-Komponente installiert und gestartet wird. Man kann sich aber ein wesentlich flexibleres Zusammenwirken der Komponenten vorstellen:

- Es sollte möglich sein, dass auch die Client-Komponente vor der Service-Komponente installiert werden kann. Die Client-Komponente soll erkennen können, wann die Service-Komponente verfügbar wird und sie dann nutzen.
- Symmetrisch dazu soll die Client-Komponente auch erkennen, dass die Service-Komponente nicht mehr verfügbar ist und sie dann (freiwillig) nicht mehr nutzen. Am Ende von Abschnitt 9.3 wurde erwähnt, dass selbst bei einer Deinstallation der Service-Komponente der Thread der Client-Komponente unverändert weiterläuft. Das soll jetzt anders werden.
- Schließlich soll auch Hot Deployment möglich sein. Dazu gehört, dass eine neue Version der Service-Komponente installiert wird und der Client diese veränderte Komponente unmittelbar nutzen kann.

Um die flexiblere Zusammenarbeit zwischen den beiden Komponenten zu realisieren, sind Änderungen sowohl auf der Klassen- als auch auf der Objektebene nötig:

- Klassenebene: In der Client-Komponente wird die Schnittstelle Counter verwendet. Wenn die Service-Komponente nicht vorhanden ist, dann ist die Class-Datei für Counter nicht verfügbar. Folglich scheitert das Starten der Client-Komponente. Aus diesem Grund (und auch zur Hot-Deployment-Unterstützung) wird die erste Komponente in zwei Komponenten aufgeteilt: Die Komponente 1a enthält das Package javacomp.osgi.bundle0 mit der Schnittstelle Counter. Die Manifest-Datei dieser Komponente besitzt keine Zeilen mit Bundle-Activator und Import-Package. Es gibt lediglich eine Export-Package-Zeile, in der das oben angesprochene Package exportiert wird. Wir gehen im Folgenden davon aus, dass sich diese Komponente selten ändert und immer installiert bleibt. Die Komponente 1b enthält das Package javacomp.osgi.bundle1 mit den Klassen CounterImpl und CounterServiceActivator, aber ohne die Schnittstelle Counter (hier wird die Schnittstelle aus dem Package javacomp.osgi.bundle0 benutzt). Entsprechend muss in der Manifest-Datei auch das Package javacomp.osgi.bundle0 importiert werden. Diese Komponente muss aber ihr eigenes Package nicht exportieren. Das heißt: Die Export-Package-Zeile in der Manifest-Datei kann komplett entfernt werden. In der Manifest-Datei der Client-Komponente muss beim Import javacomp.osgi.bundle1 durch javacomp.osgi.bundle0 ersetzt werden.
- Objektebene: Die Client-Komponente soll so programmiert werden, dass sie die Service-Komponente nicht unbedingt beim Starten, sondern eventuell erst später, sobald sie vorhanden ist, nutzt. Eine Möglichkeit zur Umsetzung besteht darin, dass in der Start-Methode der nutzenden Komponente ein Thread gestartet wird, der periodisch mit getServiceReference prüft, ob der von ihm zu nutzende Dienst vorhanden ist. Alternativ kann man sich vom OSGi-Framework benachrichtigen lassen, sobald sich das Dienstangebot ändert. Im Folgenden verwenden wir diese zweite Möglichkeit. Dazu benötigt man eine Klasse, welche die vom OSGi-Framework vorgegebene Schnittstelle ServiceListener implementiert. Ein Objekt dieser Klasse muss dann mit addServiceListener am BundleContext registriert werden. Wir verwenden in unserem einfachen Beispiel die Activator-Klasse zur Implementierung der Schnittstelle ServiceListener. Damit kann das Activator-Objekt sich selbst beim BundleContext als ServiceListener anmelden. Die einzige Methode serviceChanged der Schnittstelle ServiceListener wird mit einem ServiceEvent-Parameter aufgerufen, wenn sich im Dienstangebot etwas geändert hat. Über den ServiceEvent-Parameter kann man abfragen, ob ein Dienst hinzugefügt, gelöscht oder geändert wurde. In Listing 9.7 wird nicht nur auf das Hinzufügen, sondern auch auf das Löschen eines Dienstes reagiert. Wenn zum Beispiel die genutzte Komponente nicht mehr vorhanden ist, dann stellt die nutzende Komponente die Benutzung (freiwillig) ein.

Listing 9.7 Veränderte Klasse CounterClientActivator

```
package javacomp.osgi.bundle3;

import javacomp.osgi.bundle0.*;
import org.osgi.framework.*;

public class CounterClientActivator
       implements BundleActivator, ServiceListener
{
    private BundleContext context;
    private CounterThread thread;

    public void start(BundleContext c) throws Exception
```

```
    {
        this.context = c;
        startClient();
        c.addServiceListener(this);
    }

    public void stop(BundleContext c) throws Exception
    {
        stopClient(true);
        c.removeServiceListener(this);
    }

    public void serviceChanged(ServiceEvent event)
    {
        switch(event.getType())
        {
            case ServiceEvent.REGISTERED:
                startClient();
                break;
            case ServiceEvent.UNREGISTERING:
                stopClient(false);
                break;
            case ServiceEvent.MODIFIED:
                stopClient(false);
                startClient();
                break;
            default:
        }
    }

    private void startClient()
    {
        ServiceReference<Counter> ref =
                        context.getServiceReference(Counter.class);
        if(ref != null)
        {
            Counter hello = (Counter) context.getService(ref);
            if(hello != null)
            {
                if(thread == null)
                {
                    thread = new CounterThread(hello);
                    thread.start();
                }
            }
        }
    }

    private void stopClient(boolean uncoditionally)
    {
        ServiceReference<Counter> ref =
                        context.getServiceReference(Counter.class);
        if(uncoditionally || ref == null)
        {
            if(thread != null)
            {
                thread.interrupt();
                try
```

```
                {
                    thread.join();
                }
                catch(InterruptedException e)
                {
                }
                thread = null;
            }
        }
    }
}

class CounterThread extends Thread
{
    //unverändert wie bisher
    ...
}
```

Erklärungsbedürftig ist unter Umständen der Parameter vom Typ boolean der privaten Methode stopClient. Wenn er true ist, wird ein laufender Thread in jedem Fall angehalten. Dies ist der Fall, falls die Komponente selbst angehalten wird. Wenn der Parameter false ist, dann soll der Thread nur dann angehalten werden, falls der Dienst nicht mehr verfügbar ist. Man sollte sich in diesem Zusammenhang klar machen, dass die Methode serviceChanged auch aufgerufen wird, wenn andere Dienste als der CounterService-Dienst an- und abgemeldet werden. Bei einem Abmeldeereignis weiß man also noch nicht sicher, welcher Dienst verschwunden ist. Deshalb wird nochmals geprüft, ob der CounterService-Dienst noch in der Registratur eingetragen ist oder nicht. Als Alternative zur Implementierung der Schnittstelle ServiceListener kann die Klasse ServiceTracker eingesetzt werden. Diesem Hinweis können Sie bei Interesse selbst nachgehen.

Mit den beschriebenen Änderungen sind die eingangs genannten Nutzungsformen möglich. Wenn die Komponente, die die Schnittstelle Counter enthält, installiert ist, dann können die Client- und Service-Komponenten in beliebiger Reihenfolge installiert, gestartet, angehalten und deinstalliert werden. Auch ist Hot Deployment möglich, wie folgendes Nutzungsprotokoll zeigt:

```
g! start 328
CounterServiceActivator.start
g! version 1 (1)
version 1 (2)
version 1 (3)
version 1 (4)
version 1 (5)
update 328version 1 (6)

CounterServiceActivator.stop
CounterServiceActivator.start
g! version 2 (2)
version 2 (4)
version 2 (6)
version 2 (8)
stop version 2 (10)
328
CounterServiceActivator.stop
g!
```

Zu Beginn sind alle Komponenten installiert. Die Client-Komponente ist schon gestartet, die Service-Komponente mit der Bundle-Kennung 328 noch nicht. Wie man sieht, fängt die Client-Komponente mit der Nutzung an, sobald der Service verfügbar wird. Der Quellcode der Klasse CounterImpl (s. Listing 9.3) wird dann geändert, indem die Konstante INCREMENT von 1 auf 2 erhöht wird. Anschließend wird das Kommando „update 328“ eingegeben, was eine Neuinstallation der Service-Komponente zur Folge hat. Wie zu sehen ist, verwendet dann der Client nicht nur das neue Objekt, sondern auch die neue Version der Klasse CounterImpl, ohne dass die Client-Komponente dazu angehalten oder gar neu installiert werden muss. Dies ist ein Vorteil des Komponenten-Frameworks OSGi.

9.6 Lebenszyklus von Komponenten

Bei der Nutzung der OSGi-Implementierung Felix haben Sie bei der Ausgabe des Kommandos lb gesehen, dass ein Bundle im Lauf der Zeit unterschiedliche Zustände annehmen kann. Welche Zustände und welche Zustandsübergänge es gibt, zeigt Bild 9.1.

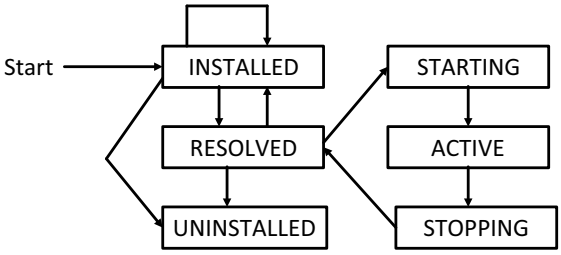

Bild 9.1 Lebenszyklus von OSGi-Komponenten

Die Zustände haben folgende Bedeutung:

- INSTALLED: Dieser Zustand wird eingenommen, nachdem ein Bundle installiert wurde.
- RESOLVED: Dieser Zustand zeigt an, dass alle Packages, die ein Bundle benötigt (d. h. importiert), verfügbar sind. Das Bundle kann gestartet werden. Wenn man in Felix das Start-Kommando eingibt, wird für den Fall, dass das Bundle noch im Zustand INSTALLED ist, zuerst der Zustand RESOLVED eingenommen. Wenn eine Komponente angehalten wird (in Felix durch Eingabe des Kommandos stop), dann kehrt die Komponente in den RESOLVED-Zustand zurück.
- STARTING und STOPPING: Dies sind Zwischenzustände, die in der Regel nur eine kurze Zeit eingenommen werden, und zwar während der Ausführung der Start- bzw. Stop-Methode.
- ACTIVE: Dieser Zustand bedeutet, dass ein Bundle aktiv ist.
- UNINSTALLED: Dieser Zustand ist in der Ausgabe des Kommandos lb nie zu sehen, da deinstallierte Komponenten gar nicht mehr in der Liste auftauchen.

Der Zustand RESOLVED weist nochmals explizit darauf hin, dass der Abgleich der Imports mit den Exports anderer Komponenten erfolgreich verlaufen ist. In OSGi kann eine Komponente in ihrem Programmcode nur die Packages verwenden, die sie explizit importiert. Und

ein solcher Import ist nur für solche Packages möglich, die von anderen Bundles explizit exportiert werden. Man erreicht dadurch eine weitere Sichtbarkeitsstufe: Normalerweise kann man durch Java-Sprachmittel die Sichtbarkeit von Attributen und Methoden einer Klasse bzw. Schnittstelle sowie die Sichtbarkeit von Klassen und Schnittstellen eines Packages festlegen. Die Packages selbst haben innerhalb einer Java-Anwendung jedoch globale Sichtbarkeit (d. h. in jedem Package kann jedes andere verwendet werden). Durch OSGi kann man auch eine Sichtbarkeit auf Package-Ebene festlegen. Zur Realisierung werden – wie schon beschrieben – unterschiedliche Klassenlader für unterschiedliche Komponenten verwendet.

9.7 BundleContext und Bundle

Über den Parameter BundleContext, der einer Komponente in den BundleActivator-Methoden übergeben wird, erhält eine Komponente Zugriff auf die Dienstleistungen, die das OSGi-Framework bietet. Viele der Methoden sind überladen. Im Folgenden wird eine Auswahl einiger Methoden (wegen des Überladens ohne Parameter) angegeben, um einen Eindruck von der gebotenen Funktionalität zu erhalten:

- getBundle und getBundles (Rückgabetyp Bundle bzw. Bundle[]): Mit diesen Methoden erhält man Zugriff auf das eigene oder andere Bundles.
- installBundle: Damit kann man per Programm ein Bundle installieren.
- addServiceListener und removeServiceListener: In Listing 9.7 wurden diese Methoden verwendet. Sie dienen zum An- und Abmelden eines ServiceListeners, der über Änderungen im Dienstangebot informiert wird.
- addBundleListener und removeBundleListener: Diese Methoden dienen zum An- und Abmelden eines BundleListeners. Ein BundleListener wird bei allen Zustandsänderungen von Bundles (s. Bild 9.1) informiert.
- addFrameworkListener und removeFrameworkListener: Ein FrameworkListener wird aufgerufen, um Zustandsänderungen des Frameworks (Fehler, Warnungen usw.) mitzuteilen.
- registerService: Diese Methoden dienen wie gesehen zum Anmelden eines Dienstes.
- getServiceReference, getServiceReferences und getAllServiceReferences: Mit diesen Methoden kann man sich auf unterschiedliche Weisen ein oder mehrere ServiceReference-Objekte geben lassen.
- getService und ungetService: Mit getService erhält man unter Angabe einer ServiceReference Zugriff auf das angemeldete Dienstobjekt. Mit ungetService kann man anzeigen, dass man das Objekt nicht mehr benötigt.

Sowohl die Methoden getBundle bzw. getBundles als auch installBundle liefern ein oder mehrere Bundle-Objekte zurück. Bundle ist eine Schnittstelle, die eine OSGi-Komponente repräsentiert. Diese Schnittstelle besitzt einige Methoden wie start, stop, update und uninstall, die aufgrund ihrer Namen selbsterklärend sind und den Felix-Kommandos gleichen Namens, die wir zuvor benutzt haben, entsprechen. Eine weitere Methode ist loadClass.

Damit kann mit dem Klassenlader des betreffenden Bundles eine Klasse geladen werden. Die zusätzlichen Methoden der Schnittstelle Bundle wie getBundleContext, getBundleId, getRegisteredServices, getState, getSymbolicName und getVersion sollen nur erwähnt werden. Die Leserinnen und Leser können sich bei näherem Interesse die Javadoc-Dokumentation von OSGi ansehen.

9.8 Erweiterungen von OSGi

Das OSGi-Framework ermöglicht nicht nur die Installation von anwendungsspezifischen Komponenten, sondern auch von solchen, die das Basis-Framework erweitern. Für den Themenschwerpunkt dieses Buches sind die Declarative Services von besonderem Interesse, so dass wir sie etwas ausführlicher betrachten wollen.

9.8.1 Declarative Services

Mit Hilfe der Declarative Services (häufig auch Service Component Runtime oder SCR genannt) kann man – wie der Name sagt – Objektbeziehungen zwischen Komponenten deklarativ in einer XML-Datei beschreiben. Mit anderen Worten: Die Bereitstellung und Nutzung von Diensten, die in unseren bisherigen Beispielen durch Aufruf der Methoden registerService, getServiceReference und getService im Programmcode realisiert wurden, können nun deklarativ mit Hilfe von XML-Dateien angegeben werden. Die „Einstiegsklasse“ muss nun auch nicht mehr die Schnittstelle BundleActivator implementieren, sondern kann eine „ganz normale“ Java-Klasse (POJO: Plain Old Java Object) sein. Durch die Beschreibung in der XML-Datei kann man zum Beispiel auch angeben, welche Methoden beim Starten und Stoppen der Komponente ausgeführt werden sollen.

Um die folgenden Beispielkomponenten erfolgreich verwenden zu können, muss die Komponente, welche Declarative Services realisiert, installiert und gestartet worden sein. Dies setzt allerdings das Vorhandensein der Komponente „Configuration Admin Service“ voraus. Wenn man also in Felix das Kommando lb ausführt, dann müssen u. a. diese Ausgabezeilen zu sehen sein:

```
g! lb
START LEVEL 1
   ID|State  |Level|Name
    0|Active |    0|System Bundle (4.0.2)
  147|Active |    1|Apache Felix Configuration Admin Service (1.2.8)
  148|Active |    1|Apache Felix Declarative Services (1.6.0)
...
g!
```

Als Beispiel für die Verwendung von Declarative Services realisieren wir die bisherigen Beispielkomponenten nun mit Hilfe von Declarative Services. Damit Hot Deployment unterstützt wird, gehen wir wieder wie in Abschnitt 9.5 davon aus, dass eine Komponente installiert ist, die das Package javacomp.osgi.bundle0 exportiert, in dem sich die Schnittstelle

Counter befindet. Unsere Service-Komponente enthält in diesem Fall nur die Klasse CounterImpl (s. Listing 9.8), die sich von der vorhergehenden Version aus Abschnitt 9.5 (das war die ursprünglichen Version aus Listing 9.3 mit der Änderung, dass die Schnittstelle Counter aus dem Package javacomp.osgi.bundle0 stammt) dadurch unterscheidet, dass – rein zu Demonstrationszwecken – noch zwei Methoden mit den Namen activate und passivate hinzugefügt wurden.

Listing 9.8 Klasse CounterImpl zur Verwendung mit Declarative Services

```
package javacomp.osgi.bundle4;

import javacomp.osgi.bundle0.Counter;
import org.osgi.service.component.ComponentContext;

public class CounterImpl implements Counter
{
    private static final int INCREMENT = 1;
    private static final String VERSION = "version " + INCREMENT;

    private int counter;

    //Methoden increment, reset und version wie bisher:
    ...

    protected void activate(ComponentContext cc)
    {
        System.out.println(getClass().getName() + ".activate");
    }

    protected void passivate(ComponentContext cc)
    {
        System.out.println(getClass().getName() + ".passivate");
    }
}
```

Die Klasse CounterImpl ist eine POJO-Klasse, implementiert also nicht die Schnittstelle BundleActivator mit den Methoden start und stop. Unsere Service-Komponente besitzt auch keine weitere Klasse, die BundleActivator implementiert. Falls Methoden beim Starten und Stoppen aufgerufen werden sollen, dann können diese in der XML-Datei angegeben werden (s. später). Diese Methoden können zwar einen beliebigen Namen tragen, müssen aber als Rückgabetyp void, die Sichtbarkeit public oder protected und ein Argument des Typs ComponentContext haben (der Typ ComponentContext stammt von der Komponente „Configuration Admin Service“). Die Manifest-Datei unserer Service-Komponente enthält jetzt keinen Eintrag mehr für Bundle-Activator. Folglich muss auch das dazugehörige Package org.osgi.framework nicht mehr importiert werden. Es kommt aber für ComponentContext ein Import des Package org.osgi.service.component hinzu. Außerdem befindet sich in der Manifest-Datei eine neue Zeile mit dem Schlüsselwort Sevice-Component, in dem die XML-Datei, die von der Declarative-Services-Komponente ausgewertet wird, benannt ist. Die Manifest-Datei für unsere neue Service-Komponente sieht damit wie folgt aus:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: CounterService
```

```
Bundle-SymbolicName: CounterService
Bundle-Version: 2.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Bundle-Vendor: Rainer Oechsle
Import-Package: org.osgi.service.component, javacomp.osgi.bundle0
Service-Component: OSGI-INF/counterservice.xml
```

Wie zu sehen ist, befindet sich in unserem Beispielfall die XML-Datei in einem eigenen Verzeichnis. Die Jar- bzw. Zip-Datei unserer Komponente besitzt auf der obersten Ebene also die Verzeichnisse javacomp, META-INF und OSGI-INF. Die Datei counterservice.xml hat den folgenden Inhalt:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
 activate="activate" deactivate="passivate" immediate="true"
 name="DeclarativeCounterService">
    <implementation
     class="javacomp.osgi.bundle4.CounterImpl"/>
    <service>
        <provide
         interface="javacomp.osgi.bundle0.Counter"/>
    </service>
</scr:component>
```

Die XML-Datei sollte zum großen Teil verständlich sein. Im Element <implementation> wird die „Einstiegsklasse“ der Komponente angegeben. Das Element <service> spezifiziert die Schnittstelle, die mit registerService als Dienst angemeldet wird. Außerdem werden als Attribute des Elements <component> die Namen der Methoden für die Aktivierung und Deaktivierung angegeben. Durch das Setzen von immediate auf true wird beim Starten der Komponente sofort ein Objekt der Einstiegsklasse erzeugt und die Aktivierungsmethode aufgerufen. Man sieht also die Ausgabe der Methode activate direkt beim Start. Setzt man immediate auf false, so wird das Erzeugen und Aktivieren des Objekts der Einstiegsklasse so lange verschoben, bis die Komponente tatsächlich von einer anderen Komponente benutzt wird. In diesem Fall sieht man die Ausgabe der Methode activate erst dann, wenn man die Client-Komponente startet, die jetzt besprochen wird.

Auch die Client-Komponente hat keinen BundleActivator mehr. Zu Demonstrationszwecken gibt es stattdessen Methoden, die beim Starten und Stoppen der Komponente aufgerufen werden. Für die Funktionalität wichtig sind die beiden Methoden startCounterThread und stopCounterThread. Durch entsprechende Konfiguration in der XML-Datei werden diese jedes Mal dann aufgerufen, wenn ein Dienst des Typs javacomp.osgi.bundle0.Counter verfügbar wird bzw. wenn dieser Dienst nicht mehr verfügbar ist. Listing 9.9 zeigt den Programmode der neuen Client-Komponente.

Listing 9.9 Klasse CounterClient zur Verwendung mit Declarative Services

```
package javacomp.osgi.bundle5;

import javacomp.osgi.bundle0.Counter;
import org.osgi.service.component.ComponentContext;

public class CounterClient
{
```

```
    private CounterThread thread;

    protected void activate(ComponentContext cc)
    {
        System.out.println(getClass().getName() + ".activate");
    }

    protected void passivate(ComponentContext cc)
    {
        System.out.println(getClass().getName() + ".passivate");
    }

    protected void startCounterThread(Counter counter)
    {
        if(thread == null)
        {
            thread = new CounterThread(counter);
            thread.start();
        }
    }

    protected void stopCounterThread(Counter counter)
    {
        if(thread != null)
        {
            thread.interrupt();
            thread = null;
        }
    }
}

class CounterThread extends Thread
{
    //unverändert wie bisher
    ...
}
```

Die Wiedergabe der Manifest-Datei für die Client-Komponente können wir uns sparen. Sie unterscheidet sich von der Manifest-Datei der Service-Komponente lediglich dadurch, dass die Komponente einen anderen Namen hat und der Name der XML-Datei anders ist. Auch in diesem Fall gibt es keinen Bundle-Activator und keine Exports. Die Imports sind ebenfalls identisch.

Die XML-Datei für die Client-Komponente ist allerdings deutlich anders:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
 activate="activate" deactivate="passivate" immediate="true"
 name="DeclarativeCounterClient">
    <implementation
     class="javacomp.osgi.bundle5.CounterClient"/>
    <reference bind="startCounterThread" cardinality="1..1"
     interface="javacomp.osgi.bundle0.Counter"
     name="Counter" policy="dynamic" unbind="stopCounterThread"/>
    />
</scr:component>
```

Neu ist jetzt das Element <reference>. Darin wird durch das Attribut interface angegeben, welchen Dienst die Komponente nutzen möchte. Durch die Attribute bind und unbind werden die Methoden spezifiziert, die automatisch aufgerufen werden, wenn der Dienst verfügbar wird bzw. nicht mehr verfügbar ist. Diese brauchen einen Parameter vom Typ des zu nutzenden Dienstes. Dadurch haben wir übrigens dieselbe Funktionalität des Clients wie in Abschnitt 9.5 durch Implementierung der ServiceListener-Schnittstelle: Es ist gleichgültig, in welcher Reihenfolge die Service- und Client-Komponenten installiert und gestartet werden. Sobald beide vorhanden sind, läuft der Thread los. Auch auf eine Aktualisierung der Service-Komponente reagiert der Client und benutzt sofort die neue Version der Klasse CounterImpl.

Statt einer XML-Datei kann man die für die Declarative Services benötigten Angaben auch durch Annotationen machen (z. B. @Component für Klassen, @Activate, @Deactivate und @Reference für Methoden). Die Annotationen haben allerdings als RetentionPolicy den Wert CLASS (vgl. Kapitel 3). Damit sind sie zur Laufzeit nicht mehr verfügbar. Die Annotationen sind dafür gedacht, dass man vor der Installation mit einem Software-Werkzeug aus den Annotationen die entsprechenden XML-Dateien erzeugt.

9.8.2 Zusätzliche Erweiterungen

Es gibt weitere Komponenten, die die Grundfunktionalität von OSGi erweitern. Dazu gehören u. a.:

- Gogo: Dies ist eine Komponente, die wir bisher schon benutzt haben. Sie liefert die Möglichkeit, mit Hilfe von Kommandos wie install und start Komponenten zu installieren und zu starten. Eine solche Shell zur Eingabe von Kommandozeilen gehört nicht zum Kern des OSGi-Frameworks. Dass man eine solche scheinbar grundlegende Funktionalität quasi als Anwendungskomponente realisieren kann, wird möglich, weil OSGi eine Programmierschnittstelle zum Installieren und Starten von Komponenten bietet. Davon macht Gogo (wie übrigens auch Declarative Services, File Install und iPOJO [s. u.]) Gebrauch. Dies ist ein schönes Beispiel dafür, wie man mit einem kleinen, aber funktional gut ausgestatteten Kern weitere zusätzliche nützliche Funktionen realisieren kann, die aber bereits die Form von Anwendungen haben und nicht zum Kern gehören.
- File Install: Mit dieser Komponente wird es möglich, Bundle-Dateien in einem Verzeichnis abzulegen. Die Komponente „File Install“ sieht von Zeit zu Zeit nach, ob es neue Bundles gibt. Wenn ja, dann werden diese installiert, ohne dass man ein entsprechendes Kommando eintippen muss. Entsprechend werden Komponenten deinstalliert oder aktualisiert, wenn die Datei verschwunden ist oder erneuert wurde. Diese Funktionalität haben wir in der prototypischen Realisierung eines Komponenten-Frameworks in Kapitel 6 schon gesehen. Dort war sie Teil des Frameworks. Hier ist die Lösung wesentlich schöner, da auch diese Komponente wie Gogo eine Anwendung eines sehr kleinen Framework-Kerns ist, somit nur bei Bedarf installiert werden muss und beliebig ausgewechselt werden kann.
- iPOJO: Einen sehr ähnlichen Ansatz wie Declarative Services verfolgt iPOJO. Mit iPOJO lässt sich ebenfalls deklarativ über eine XML-Datei spezifizieren, welche Dienste eine Komponente anbietet und nutzt. Auch für iPOJO gibt es Annotationen, die ebenfalls nicht zur Laufzeit auswertbar sind, sondern für Tools gedacht sind, die vor der Installation XML-Dateien aus den Annotationen generieren.

- Log Service: Wie der Name verrät, wird damit ein von vielen Komponenten benötigter Dienst zum Logging realisiert. Beim Starten der Komponente werden zwei Dienste angemeldet: ein LogService, mit dem man Log-Meldungen eintragen kann, und ein LogReaderService, mit dem man Log-Einträge lesen kann. An einem LogReaderService kann man sich auch als LogListener anmelden, so dass man bei jedem neuen Log-Eintrag benachrichtigt wird.
- Event Admin Service: Dieser Dienst ist ein sogenannter Pub-Sub-Dienst (Publish-Subscribe). Der Dienst stellt unterschiedliche Kanäle zur Verfügung, an die man Ereignisse melden kann (sendEvent, postEvent). Außerdem kann man sich bei diesem Dienst als EventHandler für einen bestimmten Kanal anmelden. Immer, wenn ein Ereignis für diesen Kanal produziert wird, werden alle an diesem Kanal angemeldeten EventHandler durch Aufruf der Methode handleEvent benachrichtigt.
- HttpService und WebConsole: Der Dienst HttpService stellt einen eigenen Web-Server zur Verfügung. Über den von ihm zur Verfügung gestellten Dienst kann man Servlets (s. Kapitel 12) registrieren, die dann zur Produktion einer Web-Seite aktiviert werden, wenn eine mit dem Servlet angegebene URL bei diesem Web-Server angefordert wird. WebConsole nutzt den Dienst HttpService. Er meldet ein Servlet an, so dass es möglich wird, OSGi statt über Kommandozeile (Gogo) oder über das Dateisystem (File Install) auch über einen Browser webbasiert zu steuern.

9.9 Versionen von Komponenten

OSGi bietet eine relativ ausgeprägte Unterstützung für das Umgehen mit unterschiedlichen Versionen von Komponenten und Packages. Wie bereits in den Beispiel-Manifest-Dateien gesehen kann man mit einer Bundle-Version-Zeile eine Versionsnummer für eine Komponente angeben. Von einer Komponente eines bestimmten Namens (Bundle-Symbolic-Name) und einer bestimmten Version kann zu einem Zeitpunkt immer nur höchstens eine existieren. Sehr wohl können aber zu einer Komponente mit einem bestimmten Namen gleichzeitig mehrere Versionen installiert sein. Auch einem Package kann beim Export eine Versionsnummer zugewiesen werden:

```
Export-Package: javacomp.osgi.bundle1;version="1.2.3"
```

Auf diese Art können mehrere Bundles dasselbe Package exportieren; sinnvollerweise sollte dies mit unterschiedlichen Versionsnummern erfolgen. Beim Importieren kann ebenfalls eine Versionsnummer angegeben werden:

```
Import-Package: javacomp.osgi.bundle1;version="1.2.3"
```

Statt einer konkreten Versionsnummer kann ein Versionsintervall angegeben werden. Dieses Intervall kann geschlossen sein (d. h. die als Grenzen angegebenen Versionsnummern gehören zum Bereich der akzeptierten Versionsnummern dazu):

```
Import-Package: javacomp.osgi.bundle1;version="[1.1.3,2.3.9]"
```

Das Intervall kann aber auch offen sein (in diesem Fall gehören die angegebenen Versionsnummern nicht mehr zum Bereich der akzeptierten Versionsnummern):

```
Import-Package: javacomp.osgi.bundle1;version="(1.1.3,2.3.9)"
```

Auch halboffene Intervalle sind möglich:

```
Import-Package: javacomp.osgi.bundle1;version="(1.1.3,2.3.9]",
                javacomp.osgi.bundle2;version="[2.4.8,2.5.3)"
```

Es gibt noch einige weiterer Aspekte von OSGi, die interessant wären, auf deren Darstellung ich aber verzichten möchte, da noch eine ganze Reihe weiterer Komponentensysteme auf ihre Vorstellung warten.

9.10 Bewertung

Wir prüfen nun, inwiefern OSGi gemäß den Kriterien aus Kapitel 7 ein Komponentensystem darstellt. Dabei wird nicht nur der OSGi-Kern berücksichtigt, sondern auch die Erweiterung durch die Declarative Services.

- Zu E1: Eine OSGi-Komponente, ein Bundle, muss eine klar definierte Verzeichnisstruktur aufweisen. Im Verzeichnis META-INF befindet sich die Manifest-Datei. Darin kann als Bundle-Activator eine Einstiegsklasse spezifiziert werden. Wie im Text erklärt kann es auch Bundles geben, die keine Einstiegsklasse benötigen. Das können zum Beispiel solche sein, die nur Klassen und Schnittstellen eines oder mehrerer Packages zur Verfügung stellen. Es kann auch sein, dass in der Manifest-Datei auf eine weitere Konfigurationsdatei verwiesen wird, die dann von einer Komponente wie Declarative Services ausgewertet wird. In dieser Datei kann dann ebenfalls eine (weitere) Einstiegsklasse festgelegt sein.
- Zu E2: Eine Kombination von Komponenten erfolgt einerseits deklarativ auf Package-Ebene durch Import- und Export-Package in den Manifest-Dateien und andererseits auf Objektebene. Zur Kombination auf Objektebene kann man zum einen per Programmcode Objekte beim Framework registrieren und diese dann über entsprechende Methoden finden. Zum anderen ist aber durch Declarative Services auch eine Kombination von Objekten möglich, die rein deklarativ durch XML-Dateien oder Annotationen vorgenommen werden kann. Hot Deployment wird ebenfalls problemlos unterstützt.
- Zu E3: Das Framework erzeugt Objekte der Einstiegsklassen, falls ein Bundle-Activator definiert ist. Auch mit den Declarative Services werden Objekte entsprechend konfigurierter Einstiegsklassen erzeugt. Lebenszyklen gibt es in mehrfacher Weise: Zum einen gibt es einen Lebenszyklus für die Komponente selbst, u. a. mit den Zuständen INSTALLED, RESOLVED und ACTIVE (s. Bild 9.1). Zum anderen gibt es einen Lebenszyklus für die vom Framework oder seinen Erweiterungen erzeugten Objekte, entweder durch Aufruf der Methoden start und stop eines BundleActivator-Objekts oder durch Aufruf von Methoden, die als Activate- und Passivate-Methoden gekennzeichnet sind, bei einem Component-Objekt, falls Declarative Services im Einsatz sind.

- Zu E4: Auf Package-Ebene müssen die exportierten und importierten Packages explizit genannt werden, andernfalls ist eine gemeinsame, komponentenübergreifende Nutzung von Packages nicht möglich. Auf der Objektebene kann die Nutzung eines Objekts einer Komponente durch eine andere durch Nutzung der Methoden registerService, getServiceReference und getService im Programm versteckt sein. Bei der Nutzung von Declarative Services ist auch die Objektnutzung explizit außerhalb des Programmcodes dargestellt und ablesbar.

Wie diese Ausführungen erkennen lassen, stellt OSGi ein Komponentensystem dar, das alle Anforderungen vollständig erfüllt. Dies sollte wenig überraschen, denn schließlich diente OSGi als wesentliche Vorlage bei der Aufstellung der vier Eigenschaften E1 bis E4. Nachdem wir nach den historisch interessanten Java Beans jetzt mit OSGi ein perfektes Komponentensystem kennengelernt haben, stellt sich die Frage, warum im Folgenden noch weitere Komponentensysteme betrachtet werden. Der wesentliche Grund dafür ist in der Tatsache zu sehen, dass die weiteren Komponentensysteme speziellen Einsatzzwecken dienen. So dienen Applets zum Beispiel dazu, in eine Web-Seite ein oder mehrere Java-Programme als Komponenten zu integrieren. Oder mit Servlets können einem Web-Server Komponenten hinzugefügt werden, die webbasierte Anwendungen darstellen und damit die Funktionalität des Web-Servers erweitern. Im folgenden Kapitel betrachten wir, wie die Entwicklungsumgebung Eclipse durch Komponenten, die in diesem Fall Plugins heißen, an unterschiedliche Bedürfnisse angepasst werden kann.

10 Eclipse

Eclipse ist (neben NetBeans) die bekannteste und populärste Entwicklungsumgebung (IDE: Integrated Development Environment) für Java-Software-Projekte. Eclipse ist für dieses Buch aber nicht deshalb von Interesse, sondern wegen der Tatsache, dass die Eclipse-Software aus einem kleinen Kern (Komponenten-Framework) und vielen Plugins (Komponenten) besteht. Eclipse ist dadurch perfekt auf ganz unterschiedliche Einsatzgebiete anpassbar: So gibt es spezielle Erweiterungen für die Entwicklung von Servlets, EJB- oder Android-Anwendungen. Auch zur Entwicklung von Eclipse-Plugins eignet sich Eclipse sehr gut; hierzu steht die sogenannte PDE (Plugin Development Environment) zur Verfügung. Eclipse kann aber auch für Software-Entwicklungsprojekte verwendet werden, in denen andere Programmiersprachen als Java eingesetzt werden. So gibt es zum Beispiel auch eine Entwicklungsumgebung für C++. Wäre Eclipse ein monolithischer Software-Block, der die gesamte für Eclipse zur Verfügung stehende Funktionalität beinhaltet, so wäre Eclipse sehr umfangreich, bräuchte sehr lange zum Starten und hätte einen für die Anwenderinnen nicht mehr zu überschauenden Funktionsumfang, wovon eine einzelne Anwenderin in der Regel nur einen sehr kleinen Teil benötigt (jede Anwenderin aber einen unterschiedlichen Teil).

Durch das Plugin-Konzept ist es im Gegensatz dazu möglich, dass jede Anwenderin nur die Teile zu ihrer Eclipse-Installation hinzufügt, die sie tatsächlich braucht, was sich im Laufe der Zeit auch durchaus ändern kann. Zudem ist es möglich, Eclipse durch die Entwicklung eigener Plugins für die eigenen Bedürfnisse individuell zu erweitern. In diesem Fall spielt also der Aspekt der Wiederverwendung von Komponenten kaum eine Rolle. Es geht bei den Eclipse-Komponenten primär darum, Eclipse auf seine eigenen Einsatzbereiche zuzuschneiden.

Nach einer Betrachtung grundsätzlicher Konzepte von Eclipse geht es in diesem Kapitel primär um die Entwicklung von Eclipse-Plugins.

10.1 Architektur von Eclipse

10.1.1 Eclipse-Funktionsgruppen

Die vielen Eclipse-Komponenten (Plugins) lassen sich zu unterschiedlichen Funktionsgruppen zusammenfassen, die aufeinander aufbauen. Je nach installierten Plugins sind die in einer konkreten Eclipse-Installation vorhandenen Funktionsgruppen unterschiedlich, so dass Bild 10.1 lediglich Beispielcharakter hat:

- Rich Client Platform (RCP): Eclipse basiert seit der Version 3.0 auf OSGi. Die Implementierung des OSGi-Frameworks von Eclipse heißt Equinox. Neben Equinox stellt RCP weitere grundlegende Funktionalitäten bereit. Dazu gehört die Bereitstellung einer Infrastruktur für Erweiterungspunkte (s. Abschnitt 10.1.3), das Standard Widget Toolkit (SWT) und JFace zur Programmierung grafischer Benutzeroberflächen (in der Funktionalität vergleichbar mit Swing) sowie Eclipse UI, welches eine leere grafische Anwendung darstellt, die die von Eclipse bekannten Elemente wie Views, Editoren, Perspektiven, Menüstrukturen usw. unterstützt. Diese leere Anwendung ist zum Befüllen durch Plugins bestimmt. Es ist möglich, eigene Anwendungen zu entwickeln, welche auf RCP basieren, die aber nichts mit Software-Entwicklung wie Eclipse zu tun haben.
- Integrated Development Environment (IDE): Erst durch IDE wird die allgemeine Anwendungsplattform RCP zu einer Anwendung für Software-Entwicklung, dies jedoch noch unabhängig von einer konkreten Programmiersprache.
- Java Development Tools (JDT): Durch JDT wird die IDE auf die Programmiersprache Java spezialisiert. Wie einleitend erwähnt gibt es auch Erweiterungen für zahlreiche andere Programmiersprachen.
- Plugin Development Environment (PDE): Ebenfalls in der Einleitung zu diesem Kapitel wurde schon erläutert, dass es Plugins für spezielle Arten von Java-Anwendungen wie Servlets, EJB oder Android gibt. Von besonderem Interesse für dieses Kapitel ist eine Erweiterung zur Entwicklung von Eclipse-Plugins. Diese nennt sich PDE und ist in Bild 10.1 beispielhaft gezeigt. Die in diesem Kapitel später präsentierten Beispiel-Plugins wurden ebenfalls mit Hilfe von PDE entwickelt.

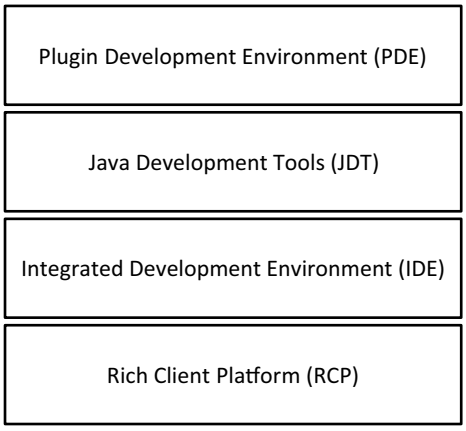

Bild 10.1 Zusammenfassung von Eclipse-Plugins zu funktionalen Gruppen

10.1.2 Workspace und Workbench

In dem Fall, dass Eclipse als Entwicklungsumgebung (für Java-Projekte) verwendet wird, spielen die Konzepte Workspace und Workbench eine zentrale Rolle:

- Workspace (Arbeitsbereich): Der Workspace liefert eine Schnittstelle zum Dateisystem und dient der Verwaltung von persistenten Ressourcen wie Projekten, Verzeichnissen und Dateien. Die von einem Dateisystem gelieferte Funktion wird durch den Workspace erweitert, indem die Ressourcen mit Markierungen versehen werden können (Aufgabenmarkierungen oder Problemmarkierungen bei Syntaxfehlern oder Warnungen in Java-Programmen, die sich „nach oben“ auf Packages und Projekte fortpflanzen). Eine weitere Zusatzfunktion des Workspace ist die Möglichkeit, dass man Listener anmelden kann, um sich über Änderungen von Ressourcen im Workspace benachrichtigen zu lassen.
- Workbench (Werkbank): Die Workbench ist die grafische Benutzeroberfläche der Eclipse-Entwicklungsumgebung. Sie ist hierarchisch aufgebaut (s. Bild 10.2) und besteht aus den folgenden Teilen:
  - Workbench: Von der Workbench gibt es pro laufender Eclipse-Anwendung nur ein einziges Objekt (Singleton). Es repräsentiert die gesamte Oberfläche.
  - WorkbenchWindow: Zu einer Eclipse-Anwendung kann es mehrere Fenster geben. Über den Menüeintrag „New Window“ im Menü Window kann man jederzeit zusätzliche Fenster öffnen.
  - WorkbenchPage: Eine WorkbenchPage ist eine Perspektive innerhalb eines Java-Fensters. Damit ist eine bestimmte Sammlung von Editoren, Views usw. in einer bestimmten Anordnung gemeint. Bei Java-Projekten gibt es zum Beispiel die Java-Perspektive zum Entwickeln und die Debug-Perspektive zur Fehlersuche.

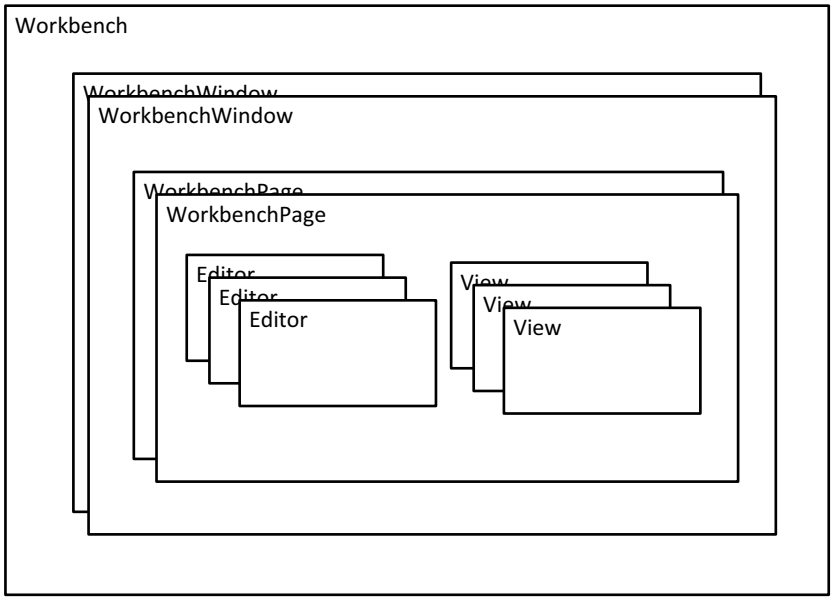

Bild 10.2 Hierarchische Struktur der Eclipse-Workbench

– Editoren und Views: Eine WorkbenchPage (Perspektive) beinhaltet u. a. Editoren und Views. Ein bekanntes Beispiel eines Editors ist ein Java-Editor, in dem nicht nur der Java-Quellcode mit Hervorhebung der syntaktischen Strukturen (Syntax-Highlighting) angezeigt wird, sondern die Eingaben des Benutzers auch ergänzt und formatiert werden. Views (Sichten) dienen der Anzeige von Informationen. Beispiele für Views sind die Console, der Package Explorer oder die Problems-View.

10.1.3 Erweiterungspunkte (Extension Points)

Wie zuvor schon erwähnt wurde, basiert Eclipse auf OSGi. Die Plugins benutzen aber nicht den BundleContext von OSGi, um sich gegenseitig zu finden, sondern sogenannte Erweiterungspunkte (Extension Points), was zum Teil historische Gründe hat, denn schließlich bildete OSGi nicht von Anfang an die Grundlage von Eclipse, sondern erst seit der Eclipse-Version 3.

In Bild 10.3 ist die Situation der Kombination von Plugins über Erweiterungspunkte dargestellt. Die Halbkreise am oberen Rand der Plugin-Kästchen symbolisieren die Erweiterungspunkte, die ein Plugin definiert. Ein Pfeil von einem Plugin zu einem Erweiterungspunkt stellt das Andocken eines Plugins an einem Erweiterungspunkt dar. Wie Bild 10.3 zu entnehmen ist, kann ein Plugin beliebig viele Erweiterungspunkte bereitstellen und beliebig viele Erweiterungspunkte desselben und unterschiedlicher Plugins nutzen. Umgekehrt kann ein Erweiterungspunkt von beliebig vielen Plugins benutzt werden.

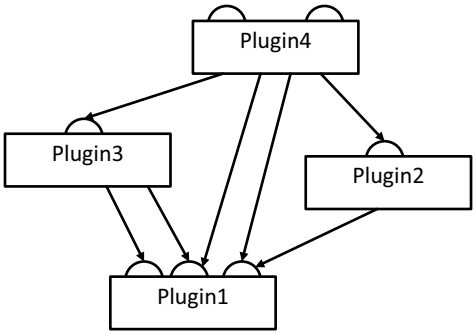

Bild 10.3 Plugins mit Erweiterungspunkten

Die Java-Entwicklungsumgebung bietet beispielsweise Erweiterungspunkte zum Hinzufügen neuer Menüs, zum Hinzufügen von Einträgen in bereits existierenden Menüs und in die Werkzeugleiste (Toolbar) von Eclipse, zum Hinzufügen eigener Views und Editoren, zum Hinzufügen von Einstellungsmöglichkeiten im Preferences-Dialog und zur Definition eigener Perspektiven. Sowohl die Definition eigener Erweiterungspunkte als auch die Nutzung von Erweiterungspunkten anderer Plugins erfolgt in einer Konfigurationsdatei namens plugin.xml, die ein Plugin neben der Manifest-Datei, die wegen OSGi nötig ist, mitbringt. Bei der Definition eines neuen Erweiterungspunkts muss neben der eindeutigen Bezeichnung des Erweiterungspunkts eine XML-Schema-Datei angegeben werden, das die XML-Struktur beschreibt, die für die Nutzung dieses Erweiterungspunkts nötig ist:

```
<extension-point id="..." name="..." schema="..."/>
```

Die Nutzung eines Erweiterungspunkts wird durch das XML-Element <extension> angezeigt:

```
<extension point="...">
...
</extension>
```

Bitte beachten Sie den wichtigen Unterschied, den der Bindestrich ausmacht! Im ersten Fall geht es um das XML-Element <extension-point> (mit Bindestrich), im zweiten Fall um das Element <extension> mit dem Attribut point (zwischen extension und point ist in diesem Fall kein Bindestrich, sondern ein Leerzeichen). Die drei Punkte zwischen <extension> und </extension> stehen stellvertretend für weitere Angaben, die abhängig vom genutzten Erweiterungspunkt sind. Die Struktur dieser Angaben muss konform zu dem XML-Schema sein, das bei der Definition des Erweiterungspunkts festgelegt wurde. Als Beispiele für solche Angaben können Sie sich Beschriftungen für die Oberflächenelemente (z. B. Menüeinträge oder Views) vorstellen. Sehr häufig, aber nicht unbedingt bei allen Erweiterungspunkten notwendig ist der vollständige Name einer Klasse, von der dann Objekte generiert werden (Plugins ohne Programmcode sind zum Beispiel solche, die Hilfetexte bereitstellen). Genauer betrachtet heißt das, dass das Plugin, das einen Erweiterungspunkt definiert, die Konfigurationsangaben aller Nutzungen dieses Erweiterungspunkts auslesen und, falls eine Einstiegsklasse spezifiziert ist, von dieser Klasse Objekte erzeugen kann. Daraus folgt, dass die Frage „wer initialisiert und erzeugt welches Objekt?“ für das Beispiel aus Bild 10.3 beantwortet werden kann, indem man sich die Pfeile in umgekehrter Richtung vorstellt. Das Plugin1 erzeugt also beispielsweise Objekte von Klassen aus den Plugins 2, 3 und 4.

Die Plugins von Eclipse nutzen in der Regel den Mechanismus der Erweiterungspunkte, um Objekte einer Einstiegsklasse zu erzeugen, so dass in den OSGi-Manifest-Dateien im Normalfall keine Bundle-Activator-Zeile zur Angabe einer Einstiegsklasse vorhanden ist. Dies ist vergleichbar mit den Declarative Services von OSGi, wo auch kein Bundle-Activator definiert werden muss und Objekte anhand der Angaben in den XML-Dateien generiert werden.

Der Programmcode von solchen Plugins, welche sich an die oben beispielhaft angegebenen Erweiterungspunkte zum Hinzufügen von Menüs, Menüeinträgen, Views, Editoren usw. andocken, basiert in der Regel immer auf SWT und JFace, also den Bibliotheken für die Programmierung grafischer Benutzeroberflächen (beim Hinzufügen eines Menüeintrags gibt man den Programmcode an, der beim Auswählen des Eintrags ausgeführt wird; beim Hinzufügen einer View gibt man den Programmcode an, der die View füllt usw.). Gute Kenntnisse von SWT und JFace sind deshalb für Entwickler „ernsthafter“ Eclipse-Plugins unverzichtbar. In unserem Demonstrationsbeispiel dieses Kapitels werden wir die Nutzung von SWT auf ein Minimum reduzieren, JFace wird sogar überhaupt nicht verwendet.

10.2 Komponentenmodell von Eclipse

Aus den Erläuterungen des vorhergehenden Abschnitts ergibt sich das Komponentenmodell von Eclipse, ohne dass dazu noch viel erklärt werden muss: Eine Eclipse-Komponente bzw. ein Eclipse-Plugin ist eine OSGi-Komponente mit einer zusätzlichen Konfigurationsdatei

namens plugin.xml. In der Datei plugin.xml wird sowohl die Nutzung fremder als auch die Definition eigener Erweiterungspunkte angegeben. Die konkreten Angaben zu der Nutzung eines Erweiterungspunkts hängen vom jeweiligen Erweiterungspunkt ab; es gibt (bis auf die Angabe, welcher Erweiterungspunkt genutzt werden soll) keine allgemein gültigen Regelungen. Dasselbe gilt für die Software eines Plugins. Falls überhaupt eine Klasse spezifiziert wird zur Nutzung eines Erweiterungspunkts, so hängt es auch in diesem Fall vom konkreten Erweiterungspunkt ab, aus welcher Klasse diese Einstiegsklasse abgeleitet oder welche Schnittstelle diese Klasse implementieren muss oder welche anderen Besonderheiten für diese Klasse gelten müssen.

Mehrere Eclipse-Plugins, die enger zusammenhängen, eine logische Einheit bilden und für die es sinnvoll ist, alle oder keines davon zu installieren, lassen sich zu sogenannten Features zusammenfassen. Darauf gehen wir im weiteren Verlauf des Buchs jedoch nicht näher ein.

Nach diesen recht abstrakten Erklärungen wird es nun höchste Zeit für einige konkrete Plugin-Beispiele.

10.3 Erstes Eclipse-Plugin

Das folgende Beispiel wurde von dem Beispiel auf der Web-Seite http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html inspiriert. Es wurde jedoch stark vereinfacht und an die Beispiele, die in den anderen Kapiteln vorkommen und immer mit einem Zähler zu tun haben, angepasst. Das erste Plugin nutzt den View-Erweiterungspunkt und liefert eine neue View für die Eclipse-Entwicklungsumgebung, die der Einfachheit halber nicht bei der Software-Entwicklung hilft und auch keinen Bezug zum Workspace hat. Es ist eine reine Demonstrationsanwendung, die für sich allein steht. Die erzeugte View enthält die Anzeige eines Zählers. In Bild 10.4 ist sie im rechten unteren Teil des Eclipse-Fensters zu sehen.

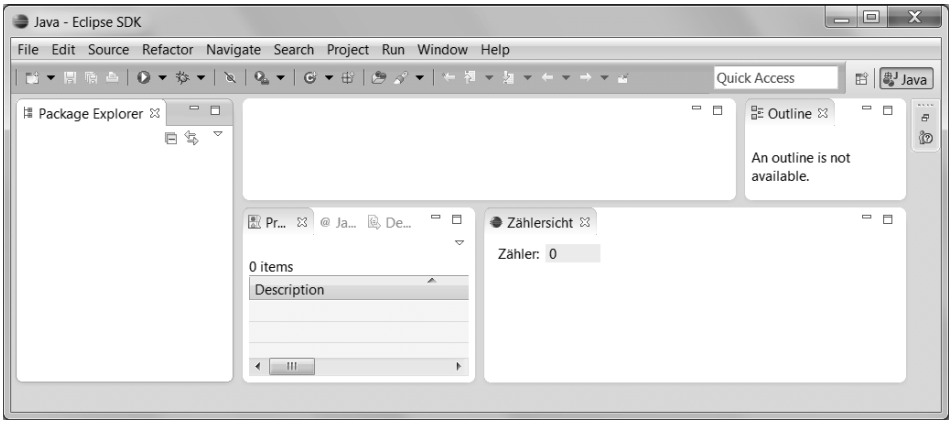

Bild 10.4 Eigene View eines Zählers (noch ohne Funktionen)

Das erste Eclipse-Plugin nutzt aber nicht nur den Erweiterungspunkt View, sondern stellt einen eigenen Erweiterungspunkt zur Verfügung, den andere Eclipse-Plugins nutzen können, um die View mit Funktionen zur Veränderung des Zählerwerts zu bestücken. In Bild 10.5 ist die Situation dargestellt, in der drei weitere Eclipse-Plugins, die im nächsten Abschnitt behandelt werden, Funktionen zum Erniedrigen, zum Erhöhen und zum Zurücksetzen des Zählers liefern.

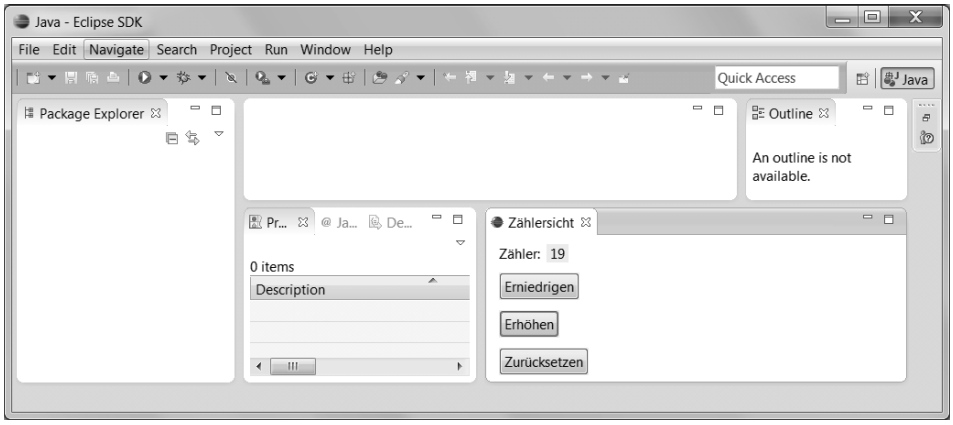

Bild 10.5 Eigene View eines Zählers mit Funktionen zum Ändern

Falls die View geschlossen wird, kann sie jederzeit wieder geöffnet werden. Wenn man im Eclipse-Menü Window den Eintrag „Show View“ auswählt, erhält man einen Dialog, in dem alle verfügbaren Views angezeigt werden, die in unterschiedliche Kategorien gruppiert sind (s. Bild 10.6). Für unsere View wurde in den Konfigurationseinstellungen festgelegt, dass die View zu der neuen Kategorie „Java-Komponenten“ gehören soll. Ein Icon kann man zu einer View ebenfalls angeben. Hier wurde einfachheitshalber das Eclipse-Logo gewählt.

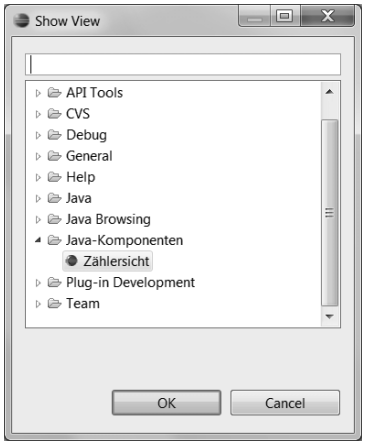

Bild 10.6 Eclipse-Dialog zum Öffnen von Views

So viel zum Aussehen unseres Eclipse-Beispiel-Plugins. Wenden wir uns nun der Realisierung zu. Unsere Eclipse-Komponente, die die Zählersicht zur Verfügung stellt, stellt eine Erweiterung des Erweiterungspunkts „org.eclipse.ui.views“ dar, der von einem anderen

Plugin der Eclipse-Entwicklungsumgebung definiert wird. Unser Plugin muss sich deshalb an die Vorgaben halten, die mit der Nutzung dieses Erweiterungspunkts verbunden sind. Dazu gehört zum einen, dass die vom Erweiterungspunkt vorgegebene XML-Schema-Definition eingehalten wird und die nötigen Angaben für diesen Erweiterungspunkt in der Datei plugin.xml unseres Plugins gemacht werden, wozu auch die Angabe des Namens einer Einstiegsklasse gehört. Diese Klasse muss aus der abstrakten Klasse ViewPart abgeleitet sein, was die Implementierung der Methode createPartControl erzwingt. Diese Methode ist dazu vorgesehen, das Befüllen der View zu programmieren. Unser Eclipse-Plugin nutzt aber nicht nur einen Erweiterungspunkt, sondern definiert auch einen eigenen Erweiterungspunkt. Dabei wird in einem XML-Schema festgelegt, welche Angaben Plugins machen müssen (und in welcher Form), die diesen Erweiterungspunkt nutzen. Auch dazu gehört wieder der Name einer Einstiegsklasse. Unser Plugin erwartet, dass jede angegebene Klasse die Schnittstelle Function, die unser CounterView-Plugin mitbringt, implementiert. Die beschriebene Situation ist in Bild 10.7 noch einmal zusammengefasst.

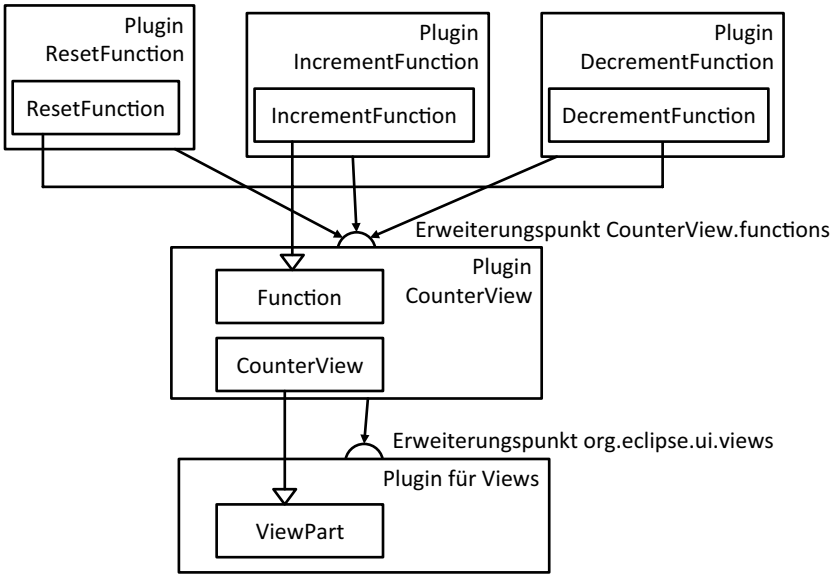

Bild 10.7 Beispiel-Plugins dieses Kapitels mit definierten und genutzten Erweiterungspunkten sowie mit definierten und genutzten Basisklassen bzw. Schnittstellen

Wie schon in Abschnitt 10.1 erläutert wurde, erfolgt die Objekterzeugung und der Aufruf initialer Methoden in der umgekehrten Richtung der Pfeile, welche die Nutzung von Erweiterungspunkten symbolisieren. Das heißt also für Bild 10.7, dass das Views-Plugin der Eclipse-Entwicklungsumgebung ein Objekt der Klasse CounterView, welche Teil unseres CounterView-Plugins ist, erzeugt und die Methode createPartControl zur Initialisierung unserer eigenen View aufruft. Dies bedeutet aber auch, dass unser CounterView-Plugin herausfinden muss, welche Plugins sich an dem selbst definierten Erweiterungspunkt angedockt haben und welche Konfigurationsinformation sie mitbringen. Weiterhin ist unsere CounterView dafür verantwortlich, auf der Oberfläche für jedes den Erweiterungspunkt nutzende Plugin einen entsprechenden Button anzulegen, aus den Klassenangaben Objekte dieser Klassen zu erzeugen und dafür zu sorgen, dass beim Klicken des Buttons

die Methode der von diesen Objekten implementierten Schnittstelle Function aufgerufen wird.

Im Folgenden schauen wir uns den Programmcode unseres ersten Eclipse-Plugins an. Wir beginnen mit der Schnittstelle Function (s. Listing 10.1). Diese hat eine einzige Methode namens compute. Als Parameter wird ihr der aktuelle Wert des Zählers übergeben. Die Methode soll einen neuen Wert für den Zähler berechnen und diesen zurückgeben.

Listing 10.1 Schnittstelle Function

```
package javacomp.eclipse.plugin1;

public interface Function
{
    public int compute(int x);
}
```

Weiterhin beinhaltet unser Plugin eine Zählerklasse Counter mit einem Attribut des Typs int sowie den Methoden get und set zum Auslesen bzw. zum Setzen des Attributs. Das Listing dazu erspare ich Ihnen.

Die interessanteste Klasse des ersten Beispiel-Plugins ist die Einstiegsklasse CounterView (s. Listing 10.2). Sie ist aus ViewPart abgeleitet und muss die Methoden createPartControl zur Initialisierung sowie die Methode setFocus implementieren. Der selbst definierte Erweiterungspunkt hat den Namen „CounterView.functions“. Das kommt daher, dass wir unserem Plugin den Namen CounterView und dem Erweiterungspunkt den Namen functions gegeben haben. Dies erfüllt nicht die Konvention, wonach die Namen von Erweiterungspunkten so gewählt werden sollen, dass sie mit großer Wahrscheinlichkeit eindeutig sind (in unserem Fall sollte also eher ein Name gewählt werden, der mit „de.hochschule_trier.javacomp“ beginnen könnte). Da wir aber nur ein Demo-Beispiel implementieren, genügt der gewählte Name. Der Name des Erweiterungspunkts ist als Konstante in der Klasse CounterView definiert. Die Klasse muss den Namen kennen, um in der ExtensionRegistry die Informationen zu allen Nutzungen dieses Erweiterungspunkts abzufragen. Ferner sollte man wissen, dass Plugins, die den Erweiterungspunkt „CounterView.functions“ nutzen, einen Namen für ihre Funktion durch „name“ und einen Klassennamen einer Function implementierenden Klasse durch „class“ angeben müssen. Um die Informationen dazu abzufragen, gibt es zwei String-Konstanten für „name“ und „class“.

Listing 10.2 Klasse CounterView

```
package javacomp.eclipse.plugin1;

import org.eclipse.core.runtime.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.ui.part.*;
import org.eclipse.ui.*;

public class CounterView extends ViewPart
{
    private static final String EXTENSION_POINT =
                                          "CounterView.functions";
    private static final String FUNCTION_NAME_ATTRIBUTE = "name";
    private static final String CLASS_ATTRIBUTE = "class";
```

```
public void createPartControl(Composite parent)
{
    FunctionsUserInterface ui =
                              new FunctionsUserInterface(parent);
    Counter counter = new Counter();
    try
    {
        process(ui, counter);
    }
    catch(WorkbenchException e)
    {
        System.err.println(e.getMessage());
    }
}

public void setFocus()
{
}

private void process(FunctionsUserInterface ui, Counter counter)
        throws WorkbenchException
{
    IExtensionRegistry registry =
                                 Platform.getExtensionRegistry();
    IExtensionPoint extensionPoint =
                    registry.getExtensionPoint(EXTENSION_POINT);
    if(extensionPoint == null)
    {
        throw new WorkbenchException("unable to resolve " +
                         "extension-point: " + EXTENSION_POINT);
    }
    IConfigurationElement[] members =
                       extensionPoint.getConfigurationElements();

    for(IConfigurationElement member : members)
    {
        String functionName =
                    member.getAttribute(FUNCTION_NAME_ATTRIBUTE);
        if(functionName == null)
        {
            functionName = "[namenlose Funktion]";
        }

        //Markierung Nr. 1
        Object callback = null;
        try
        {
            callback =
              member.createExecutableExtension(CLASS_ATTRIBUTE);
            if(!(callback instanceof Function))
            {
                System.err.println("callback function does " +
                                   "not implement Function");
                continue;
            }
        }
        catch(CoreException e)
```

```
            {
                System.err.println(e.getMessage());
                continue;
            }
            Function function = (Function)callback;
            //Markierung Nr. 2
            ui.addFunction(functionName, function, counter);
        }
    }
}
```

In der Initialisierungsmethode createPartControl wird im Konstruktor von FunctionsUserInterface die initiale Oberfläche bestehend aus der Beschriftung „Zähler“ und dem grau hinterlegten Feld mit der Anzeige des anfänglichen Werts des Zählers erzeugt (s. Bild 10.4). Die Klasse Composite entspricht in Swing einem JPanel und repräsentiert den Inhalt der View, der zum Befüllen als Parameter übergeben wird. Die entscheidende Methode ist die Methode process. Sie beschafft sich zunächst über eine statische Methode eine Referenz auf die ExtensionRegistry von Eclipse, in der alle existierenden Erweiterungspunkte gespeichert sind. Unter der Angabe des Namens des Erweiterungspunkts „CounterView.functions“ erhält man durch Aufruf der Methode getExtensionPoint den Eintrag des eigenen Erweiterungspunkts in der ExtensionRegistry. Davon kann man sich mit getConfigurationElements alle Verwendungen des Erweiterungspunkts geben lassen und insbesondere abfragen, was jeweils als „name“ und „class“ angegeben wurde. Von der Angabe unter „class“ kann man sich ein Objekt erzeugen lassen, wobei dafür nicht direkt Reflection, sondern eine Methode von Eclipse verwendet wird. Das erzeugte Objekt sollte nach Vorgabe des Erweiterungspunkts „CounterView.functions“ die Schnittstelle Function implementieren. Ist dies nicht der Fall, kann die Erweiterung nicht genutzt werden. Wenn alles in Ordnung ist, wird die Methode addFunction der Klasse FunctionsUserInterface mit dem Namen der Funktion, dem erzeugten Objekt der Function implementierenden Klasse sowie dem Zähler aufgerufen. Darin wird dann ein Button mit dem Namen der Funktion angelegt. Ferner wird ein Listener an dem Button angemeldet, der auf das Klicken so reagiert, dass der aktuelle Zählerwert ausgelesen und der Methode compute übergeben wird, die auf das Function-Objekt angewendet wird. Der von compute zurückgelieferte Wert wird in das Zählerobjekt geschrieben und die Anzeige auf der Oberfläche entsprechend aktualisiert. Die dazu nötigen Klassen geben wir zum Einsparen von Seiten im Buch nicht wieder; sie sind für das Thema Java-Komponenten auch nicht wesentlich. Sie können aber von der Web-Seite zum Buch bezogen werden. Für das Verständnis sind geringe Kenntnisse von SWT nötig. Zum Lesen des Codes sollte es aber auch ausreichen, wenn Sie Swing kennen.

Unser erstes Eclipse-Plugin benötigt eine Manifest-Datei, da jedes Eclipse-Plugin auch eine OSGi-Komponente ist. Da diese sich nicht wesentlich von den Manifest-Dateien des OSGi-Kapitels unterscheidet, wird auch in diesem Fall auf die Wiedergabe verzichtet. Die zweite Konfigurationsdatei plugin.xml ist aber von Interesse (s. Listing 10.3):

Listing 10.3 Datei plugin.xml für das Eclipse-Plugin CounterView

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.4"?>
<plugin>
   <extension point="org.eclipse.ui.views">
      <category
```

```
            name="Java-Komponenten"
            id="javacomp">
      </category>
      <view
            name="Zählersicht"
            icon="icons/sample.gif"
            category="javacomp"
            class="javacomp.eclipse.plugin1.CounterView"
            id="javacomp.eclipse.plugin1.CounterView">
      </view>
   </extension>
   <extension-point name="Functions" id="functions"
    schema="schema/functions.exsd"/>
</plugin>
```

In Listing 10.3 ist zu sehen, wie das Plugin den Erweiterungspunkt org.eclipse.ui.views benutzt. Die Angaben innerhalb des Elements <extension> sind von diesem Erweiterungspunkt vorgegeben. Mit <category> wird eine neue View-Gruppe definiert. Durch das Element <view> wird die neue View mit Name, Icon, Zugehörigkeit zu einer View-Kategorie und Klassenname definiert. Die zweite Angabe in plugin.xml ist die Definition des eigenen Erweiterungspunkts (auf die Bedeutung des Bindestrichs in <extension point=…> und <extension-point …> sei nochmals hingewiesen). Darin befindet sich ein Verweis auf die XML-Schema-Datei, in der definiert wird, welche Angaben bei der Nutzung des Erweiterungspunkts functions nötig sind. Auch diese Datei können Sie von der Web-Seite zum Buch beziehen.

10.4 Weitere Eclipse-Plugins

Die Eclipse-Plugins, welche den Erweiterungspunkt CounterView.functions nutzen, sind äußerst einfach. Wir werden stellvertretend nur das Plugin IncrementFunction betrachten, welches die Funktion zum Erhöhen des Zählers zur Verfügung stellt. Die anderen Plugins ResetFunction und DecrementFunction sind sehr ähnlich. Der Programmcode des IncrementFunction-Plugins besteht aus einer einzigen Klasse. Es ist die Klasse IncrementFunction, welche die Schnittstelle Function implementiert (s. Listing 10.4).

Listing 10.4 Klasse IncrementFunction

```
package javacomp.eclipse.plugin3;

import javacomp.eclipse.plugin1.Function;

public class IncrementFunction implements Function
{
    public int compute(int x)
    {
        return x+1;
    }
}
```

Außerdem gibt es eine Manifest-Datei, die keine Überraschungen birgt, und eine Datei plugin.xml, in der die Nutzung des Erweiterungspunkts CounterView.functions angezeigt wird (s. Listing 10.5).

Listing 10.5 Datei plugin.xml für das Eclipse-Plugin IncrementFunction

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
   <extension point="CounterView.functions">
      <function
            name="Erhöhen"
            class="javacomp.eclipse.plugin3.IncrementFunction">
      </function>
   </extension>
</plugin>
```

Wenn man die beiden Dateien plugin.xml aus Listing 10.3 und Listing 10.5 vergleicht, wird deutlich, dass der Inhalt zwischen <extension> und </extension> in den beiden Dateien vollkommen unterschiedlich ist. Es hängt eben vom benutzten Erweiterungspunkt ab, welche Angaben in welcher Form notwendig und möglich sind. Festgelegt wird dies durch die XML-Schema-Datei, die bei der Definition eines Erweiterungspunkts angegeben wird. Weiterhin wird in beiden Fällen ein Klassenname angegeben, was nicht für alle Erweiterungspunkte gelten muss, aber für die beiden hier betrachteten Erweiterungspunkte org.eclipse.ui.views und CounterView.functions der Fall ist, da dies in den jeweiligen Schemadefinitionen so festgelegt ist. Welche Bedingungen diese Einstiegsklassen erfüllen müssen, hängt ebenfalls vom Erweiterungspunkt ab: Für org.eclipse.ui.views muss die Klasse aus ViewPart abgeleitet sein, für CounterView.functions muss sie die Schnittstelle Function implementieren.

10.5 Erweiterung der Eclipse-Plugins

Die Plugins, die Sie von der Web-Seite zu diesem Buch herunterladen können, haben ein klein wenig mehr Funktionalität als bisher dargestellt. Es ist nämlich möglich, dass die bereitgestellten Funktionen zur Berechnung des neuen Zählerwerts neben dem aktuellen Zählerwert zusätzliche Parameter berücksichtigen, die von einem Benutzer in dafür vorgesehene Felder der View eingegeben werden können (s. Bild 10.8).

Wie in Bild 10.8 zu sehen ist, kann man eingeben, um wie viel der Zähler hoch- bzw. heruntergezählt werden soll. Außerdem gibt es eine Funktion zum Setzen, die den aktuellen Zählerwert nicht berücksichtigt, sondern nur den eingegebenen Parameter verwendet. Aus Demonstrationsgründen ist in der letzten Zeile eine Funktion gezeigt, die sogar zwei Parameter besitzt.

Außerdem sei noch erwähnt, dass die XML-Schemadatei für den Erweiterungspunkt CounterView.functions so definiert ist, dass bei der Nutzung des Erweiterungspunkts beliebig viele <function>-Elemente angegeben werden dürfen. Das bedeutet, dass alle in Bild 10.8 gezeigten Funktionen mit einem einzigen Plugin realisiert werden könnten.

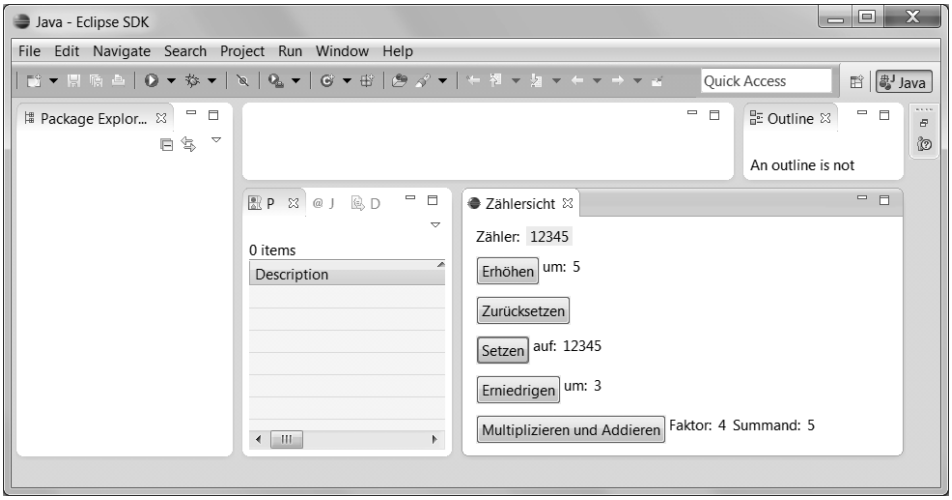

Bild 10.8 Änderungsfunktionen der Zählersicht mit zusätzlichen Parametern

10.6 Klassenladen bei Bedarf

Wenn wir der Klasse CounterView (s. Listing 10.2) einen parameterlosen Konstruktor mit einer Ausgabe hinzufügen, dann sehen wir diese Ausgabe nur dann direkt beim Starten von Eclipse, wenn die CounterView von Anfang an sichtbar ist. Wenn dagegen die View geschlossen und anschließend Eclipse neu gestartet wird, dann hat sich Eclipse gemerkt, dass die View nicht geöffnet war und öffnet sie auch beim Neustart nicht. Entsprechend sehen wir die Ausgabe des CounterView-Konstruktors nicht. Das bedeutet, dass das Plugin, welches für alle Views in Eclipse verantwortlich ist (in Bild 10.7 ganz unten zu sehen), das Laden der Klasse CounterView und die Instanziierung eines Objekts so lange verzögert, bis die View tatsächlich mit Hilfe des Dialogs aus Bild 10.6 geöffnet wird; dann erst sehen wir die Ausgabe des Konstruktors. Man nennt dieses Verhalten „Lazy Extension Processing“. Die Faulheit verhindert in diesem Fall das unnötige Laden von Klassen. Auch wenn man seiner Eclipse-Installation nur die Plugins hinzugefügt hat, die man zumindest ab und zu einmal benötigt, hat eine Eclipse-Installation in der Regel deutlich mehr Plugins als man in einer Sitzung benutzt. Wenn die Klassen aller Plugins bei jedem Starten von Eclipse geladen würden, dann wäre das ein riesiger und zum großen Teil unnötiger Aufwand. Außerdem kann man sich vorstellen, dass bei einem großen Plugin zur Initialisierung nicht nur eine Klasse geladen wird, denn in der Initialisierungsmethode werden in der Regel weitere Klassen verwendet, die dann eventuell wieder weitere Klassen benutzen usw., so dass das Laden einer Klasse, die Erzeugung eines Objekts davon und der eventuelle Aufruf von speziellen Initialisierungsmethoden dazu führen kann, dass sehr viele Klassen geladen werden müssen. Wenn diese Klassen benötigt werden, ist das Laden unvermeidbar. Wenn aber diese Klassen alle unnötig geladen werden, dann ist es ärgerlich. Die Eclipse-Plugins sind in der Regel so programmiert, dass unnötiges Klassenladen vermieden wird, wie das eingangs

geschilderte Experiment für das Views-Plugin zeigte. Dies gilt allerdings nicht für unser CounterView-Plugin aus Abschnitt 10.3. Dieses Plugin ist viel zu fleißig, denn es lädt bei der Initialisierung alle Funktionsklassen und erzeugt Objekte davon. Auch wenn es in unserem Beispiel nur um wenige kleine Plugins geht, wollen wir im Folgenden unserem Plugin das Faulenzen beibringen, um die Technik des „Lazy Extension Processing“ anhand eines Beispiels zu illustrieren. Wir schreiben dazu die Klasse FunctionProxy (s. Listing 10.6), die wie alle Function-Klassen die Schnittstelle Function (s. Listing 10.1) implementiert. Diese Klasse ist, wie der Namen ausdrückt, ein Proxy für das Objekt, das die eigentliche Funktion realisiert. Die Methode compute delegiert den Aufruf einfach an das Objekt der eigentlichen Funktionsklasse. Allerdings wird dieses Objekt erst bei der ersten Anwendung der Methode compute auf ein Proxy-Objekt erzeugt.

Listing 10.6 Klasse FunctionProxy

```
package javacomp.eclipse.plugin1;

import org.eclipse.core.runtime.*;

class FunctionProxy implements Function
{
    private IConfigurationElement element;
    private String classAttribute;
    private Function realFunction = null;

    public FunctionProxy(IConfigurationElement element,
                         String classAttribute)
    {
        this.element = element;
        this.classAttribute = classAttribute;
    }

    public int compute(int x)
    {
        try
        {
            getRealFunction();
        }
        catch(Exception e)
        {
            System.err.println(e.getMessage());
            e.printStackTrace();
            throw new IllegalArgumentException("cannot get " +
                                               "real function");
        }
        return realFunction.compute(x);
    }

    private void getRealFunction() throws Exception
    {
        if(realFunction != null)
        {
            return;
        }
        Object callback =
                  element.createExecutableExtension(classAttribute);
```

```
        if(!(callback instanceof Function))
        {
            throw new Exception(classAttribute + " does not " +
                                "implement Function");
        }
        realFunction = (Function) callback;
    }
}
```

Die private Methode getRealFunction enthält im Wesentlichen Code aus der Methode process der Klasse CounterView. Wir müssen dafür den Teil zwischen den Kommentaren „//Markierung Nr. 1“ und „//Markierung Nr. 2“ in der Methode process der Klasse CounterView (s. Listing 10.2) ersetzen durch die folgende Zeile:

```
Function function = new FunctionProxy(member, CLASS_ATTRIBUTE);
```

Statt der Erzeugung des Objekts der Funktionsklasse wird nur ein FunctionProxy-Objekt erzeugt, dem durch die im Konstruktor übergebenen Parameter ermöglicht wird, das Funktionsobjekt später erzeugen zu können. Dies passiert erst dann, wenn eine Benutzerin auf den entsprechenden Button klickt und das Funktionsobjekt dadurch tatsächlich benötigt wird.

10.7 Bewertung

Da jedes Eclipse-Plugin ein OSGi-Bundle ist und OSGi im vorhergehenden Kapitel schon als perfektes Komponentensystem bewertet wurde, kann auch Eclipse zweifellos als Komponentensystem gelten. Durch das Konzept der Erweiterungspunkte erfolgt wie bei den Declarative Services die Kopplung von Eclipse-Plugins in deklarativer Weise; durch die Erweiterungspunkte wird explizit deklariert, was ein Plugin liefert und was ein Plugin erwartet. Wie wir in den Beispielen dieses Kapitels gesehen haben, erfolgt aber die Kopplung nicht rein deklarativ, sondern muss in der Regel zusätzlich durch Programmcode unterstützt werden. Dennoch stellt der Mechanismus der Erweiterungspunkte durch die Angabe von XML-Schema-Dateien nach meiner Einschätzung ein sehr flexibles und elegantes Kopplungskonzept dar, das Eclipse gegenüber OSGi als Komponentensystem noch aufwertet. Ich habe also keinerlei Kritikpunkte an Eclipse bezüglich der Erfüllung der Eigenschaften, die ein Komponentensystem haben sollte. Allerdings erschwert die Plattform SWT und JFace als Basis für die Programmierung grafischer Benutzeroberflächen einen schnellen und problemlosen Einstieg in die Eclipse-Plugin-Entwicklung, denn die allermeisten Java-Entwickler lernen in ihrer Ausbildung zwar Swing, nicht aber SWT und JFace.

11 Applets

Applets sind Java-Programme, die in Web-Seiten eingebettet sind. Nachdem man im World Wide Web zunächst nur statische Dateien abrufen konnte, gab es unterschiedliche Anstrengungen, um Inhalte dynamisch zu erzeugen. Im Java-Umfeld waren das auf der Web-Server-Seite Servlets (s. folgendes Kapitel) und auf Client-Seite (also für den Browser) Applets. Applets spielen heute (leider) keine große Rolle mehr, da andere Techniken wie JavaScript und Flash stark an Bedeutung gewonnen haben. Wir werden deshalb das Thema Applets nur relativ knapp behandeln.

11.1 Erstes Beispiel

Für die Programmierung eines Applets sind Kenntnisse über die Programmierung grafischer Benutzeroberflächen in Java notwendig, die an dieser Stelle vorausgesetzt werden. Man kann Applets sowohl auf Basis des alten AWT (Abstract Window Toolkit) als auch auf Basis von Swing entwickeln. Wir werden im Folgenden Swing benutzen.

Wie jede Anwendung kann ein Applet aus mehreren Klassen bestehen. Die Haupt- oder Einstiegsklasse muss (für Swing) aus der Klasse JApplet abgeleitet sein. Wie zu einem JFrame können mit der Methode add Interaktionselemente wie Labels und Buttons zum Applet hinzugefügt werden. Es wird empfohlen, dies nicht im Konstruktor der Klasse vorzunehmen, sondern für diesen Zweck die Methode init der Basisklasse JApplet zu überschreiben.

Unser erstes Applet-Beispiel hat einen Button, durch den man bei jedem Klicken einen Zähler um eins erhöhen kann. Neben dem Button wird der Wert des Zählers in einem Label angezeigt. Der Zähler und das Label werden in der Klasse Counter (s. Listing 11.1) gekapselt. Für alle Leserinnen und Leser, die das Entwurfs- bzw. Architekturmuster MVC (Model-View-Controller) kennen, sei erwähnt, dass es sich bei dieser Klasse um eine Kombination aus Model und View handelt, was für dieses sehr einfache Beispiel in Ordnung sein soll. Die Klasse wird aus JPanel abgeleitet, damit Objekte dieser Klasse direkt dem Applet hinzugefügt werden können.

Listing 11.1 Klasse Counter mit Zähler und grafischer Ansicht

```
package javacomp.applets.applet1;

import javax.swing.*;

public class Counter extends JPanel
{
    private int counter;
    private JLabel label;

    public Counter()
    {
        counter = 0;
        label = new JLabel("" + counter);
        add(label);
    }

    public void increment()
    {
        counter++;
        label.setText("" + counter);
    }

    public void reset()
    {
        counter = 0;
        label.setText("" + counter);
    }
}
```

Im Konstruktor wird das Label erzeugt und zum JPanel (this) hinzugefügt. Mit den Methoden increment und reset wird der Zähler verändert und die Labelanzeige entsprechend angepasst. Die Methode reset spielt im ersten Beispiel keine Rolle; sie wird aber später noch benötigt.

Die Einstiegsklasse des ersten Applet-Beispiels heißt MainClass1 (s. Listing 11.2).

Listing 11.2 Klasse MainClass1

```
package javacomp.applets.applet1;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MainClass1 extends JApplet implements ActionListener
{
    private Counter counter;

    public void init()
    {
        setLayout(new GridLayout(0, 1));
        counter = new Counter();
        add(counter);
        JButton b = new JButton("Erhöhen");
        b.addActionListener(this);
        add(b);
```

```
    }

    public Counter getCounter()
    {
        return counter;
    }

    public void actionPerformed(ActionEvent e)
    {
        counter.increment();
    }
}
```

Wie empfohlen, wird die grafische Benutzeroberfläche nicht im Konstruktor, sondern in der überschriebenen Methode init aufgebaut. Diese besteht aus einem Objekt der Klasse Counter und einem Button mit der Beschriftung „Erhöhen“. Da die Klasse auch die Schnittstelle ActionListener implementiert (wer MVC kennt, sieht, dass es sich bei MainClass1 um eine kombinierte View-Controller-Klasse handelt), kann this als ActionListener am Button angemeldet werden. Bei jedem Klicken des Buttons wird die Methode actionPerformed aufgerufen, was den Aufruf von increment auf dem Counter-Objekt zur Folge hat. Wundern Sie sich bitte nicht über die Methode getCounter; sie wird für dieses Beispiel wie reset in der Klasse Counter ebenfalls noch nicht benötigt.

Die Konfigurationsinformationen befinden sich in Form eines Tags <applet> an der Stelle in der HTML-Datei, an der das Applet in die Seite eingefügt werden soll. Das Tag <applet> muss zwingend die Attribute class, width und height haben. Mit class wird der vollständige Name (d. h. mit Package-Namen) der Einstiegsklasse angegeben, mit width und height die Breite und Höhe in Pixeln, die das Applet auf der Seite einnehmen soll. Wenn das Applet aus mehreren Klassen besteht, bietet es sich an, diese in eine Archiv-Datei (Jar oder Zip) zu packen und diese Datei unter archive im <applet>-Tag anzugeben. Eine HTML-Seite, die das entwickelte Applet enthält, könnte dann z. B. wie folgt aussehen, wenn sich die Class-Dateien in der Archiv-Datei applet.zip befinden:

```
<html>
<head>
<title>Applet-Beispiel</title>
</head>
<body>
Hier kommt Text ...:
<p>
Und an dieser Stelle ist das Applet eingebettet:
<p>
<applet code="javacomp.applets.applet1.MainClass1.class"
 archive="applet.zip" width="300" height="60">
</applet>
<p>
Hier kann weiterer Text folgen ...
</body>
</html>
```

Das Applet funktioniert wie erwartet: Jedes Mal, wenn man den Button drückt, wird der um eins erhöhte Zählerwert angezeigt. Es gibt dafür keine eigene Abbildung. Falls Sie sich nicht vorstellen können, wie das Applet aussieht, so blättern Sie bitte weiter und betrachten den oberen Teil von Bild 11.1.

11.2 Zweites Beispiel

Im diesem Abschnitt soll nun eine zweite Applet-Komponente entwickelt werden, welche die erste Applet-Komponente benutzt, wobei sich beide Applets auf derselben HTML-Seite befinden sollen. Das zweite Applet soll lediglich einen Button mit der Beschriftung „Zurücksetzen“ enthalten, mit dem der Zähler der ersten Applet-Komponente zurückgesetzt werden kann. Die Methode reset ist für diesen Zweck in der Klasse Counter (s. Listing 11.1) schon vorhanden. Die Problematik liegt nun darin, sich im zweiten Applet eine Referenz auf das Counter-Objekt im ersten Applet zu verschaffen. Die Java-Ausführungsumgebung des Browsers, welche das Komponenten-Framework für unsere Applets darstellt, bietet eine Lösung für diese Problemstellung an: Man kann sich über die ererbte Methode getAppletContext ein AppletContext-Objekt beschaffen. Das AppletContext-Objekt hat eine Methode getApplet, mit der man eine Referenz auf ein anderes Applet beschaffen kann. Als Parameter muss man den Namen des Applets, dessen Referenz man haben möchte, angeben. Doch welchen Namen hat unser erstes Applet? Bisher noch keinen, aber durch das Attribut name im <applet>-Tag können wir diesen Namen deklarativ setzen. In der folgenden HTML-Seite wird der Name des ersten Applets auf „Applet1“ gesetzt. Hat man eine Referenz auf das MainClass1-Objekt der ersten Applet-Komponente, kann man sich davon eine Referenz auf das Counter-Objekt besorgen. Die Klasse MainClass1 enthält für diesen Zweck bereits die Methode getCounter (s. Listing 11.2). Damit sollte der Programmcode des zweiten Applets in Listing 11.3, das nur aus einer einzigen Klasse namens MainClass2 besteht, verständlich sein.

Listing 11.3 Klasse MainClass2

```
package javacomp.applets.applet2;

import java.awt.event.*;
import javacomp.applets.applet1.*;

import javax.swing.*;

public class MainClass2 extends JApplet implements ActionListener
{
    private Counter counter;

    public void init()
    {
        JButton b = new JButton("Zurücksetzen");
        b.addActionListener(this);
        add(b);
        MainClass1 otherApplet =
                (MainClass1)getAppletContext().getApplet("Applet1");
        counter = otherApplet.getCounter();
    }

    public void actionPerformed(ActionEvent e)
    {
        if(counter != null)
        {
            counter.reset();
```

```
        }
    }
}
```

Wenn wir davon ausgehen, dass sich die Class-Dateien für alle drei Klassen von Listing 11.1, Listing 11.2 und Listing 11.3 in der Zip-Datei applet.zip befinden, dann kann die HTML-Seite, die beide Applets enthält, z. B. so aussehen:

```
<html>
<head>
<title>Applet-Beispiel</title>
</head>
<body>
Hier kommt das erste Applet:
<p>
<applet name="Applet1"
 code="javacomp.applets.applet1.MainClass1.class"
 archive="applet.zip" width="300" height="60">
</applet>
<p>
Hier kommt wieder Text und dann das zweite Applet:
<p>
<applet name="Applet2"
 code="javacomp.applets.applet2.MainClass2.class"
 archive="applet.zip" width="300" height="30">
</applet>
</body>
</html>
```

Bitte beachten Sie, dass im ersten <applet>-Tag jetzt zusätzlich das Namensattribut vorhanden ist, das den Namen der Applet-Komponente auf „Applet1“ setzt. Dieser Name wird im Code von MainClass2 benutzt. Aus Symmetriegründen wurde dem zweiten Applet ebenfalls ein Name gegeben, obwohl dieser an keiner Stelle verwendet wird.

Nach dem Laden der HTML-Seite zeigt der Browser die Seite wie in Bild 11.1 an.

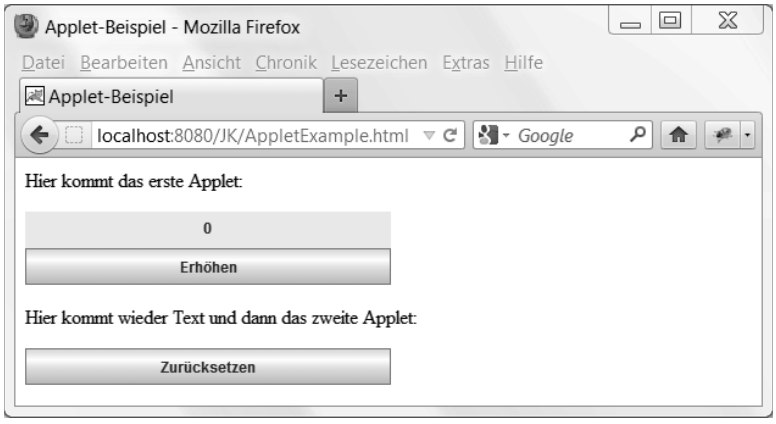

Bild 11.1 HTML-Seite mit beiden Applets nach dem Laden der Seite

Man kann nun mehrfach auf den oberen Button mit der Beschriftung „Erhöhen“ klicken, wodurch sich der Fensterinhalt wie erwartet ändert (s. Bild 11.2). Der entscheidende Punkt kommt nun: Klickt man auf den Rücksetzen-Button des zweiten Applets, wird der Zähler tatsächlich zurückgesetzt, und man sieht wieder den Zustand wie in Bild 11.1.

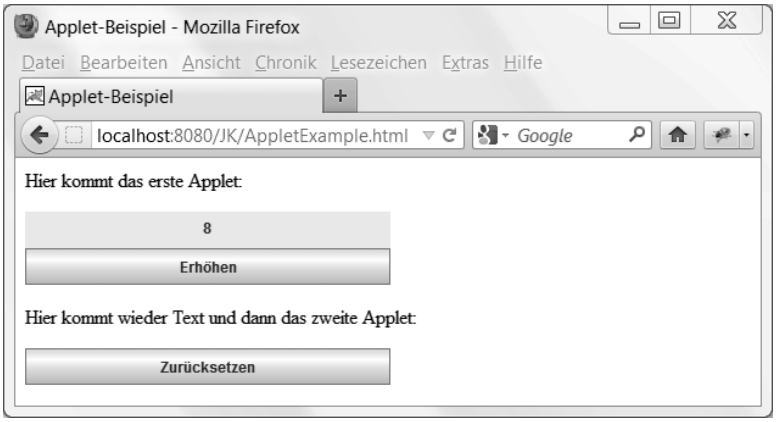

Bild 11.2 HTML-Seite nach mehrmaligem Klicken auf den Erhöhen-Button

Auf einige subtile Eigenheiten bezüglich des Klassenladens soll noch hingewiesen werden: In diesem Beispiel wurde – nicht zufällig – für beide Applets dieselbe Zip-Datei als Archiv angegeben. Dies bewirkt, dass für beide Applets dasselbe ClassLoader-Objekt benutzt wird, was Sie durch Hinzufügen der folgenden Zeile in die beiden Initialisierungsmethoden der Applet-Einstiegsklassen selber nachprüfen können:

```
System.out.println(getClass().getClassLoader());
```

Verwenden Sie dagegen unterschiedliche Archiv-Dateien, werden die Applets auch durch unterschiedliche ClassLoader-Objekte geladen. Auch das wird Ihnen durch die obige Ausgabeanweisung in der Java-Console des Browsers angezeigt. Wegen der unterschiedlichen ClassLoader-Objekte lassen sich die folgenden Effekte erklären:

- Wenn man zwei Zip-Dateien anlegt, applet1.zip für das erste Applet mit den Class-Dateien für Counter und MainClass1 und applet2.zip für das zweite Applet mit MainClass2.class, und diese jeweils als Archiv-Datei in den <applet>-Tags angibt, dann erzeugt die Initialisierung des zweiten Applets die Ausnahme ClassNotFoundException für die Klasse javacomp.applets.applet1.MainClass1. Im zweiten Applet hat man keinen Zugriff auf die Klassen MainClass1 und Counter, weil diese durch ein anderes ClassLoader-Objekt geladen wurden (diese Situation haben wir in Kapitel 6 vor der Einführung von „Uses:“ schon gesehen).
- Wenn man wieder von einer Zip-Datei mit allen drei Klassen ausgeht, diese Zip-Datei kopiert, für das erste Applet als Archiv die Original-Zip-Datei und für das zweite Applet als Archiv die Kopie der Zip-Datei angibt, dann sind die Zip-Dateien zwar gleich, es ist aber nicht dieselbe Datei. Folglich werden auch in diesem Fall wieder zwei unterschiedliche ClassLoader-Objekte für die beiden Applets benutzt. Beim Initialisieren des zweiten Applets gibt es jetzt wieder die kuriose ClassCastException, weil javacomp.applets.applet1.MainClass1 nicht zu javacomp.applets.applet1.MainClass1 gecastet werden kann (auch dieser Fall begegnete uns schon in Kapitel 6).

11.3 Bewertung

Applets mit der Java-Laufzeitumgebung des Browsers als Framework erfüllen ziemlich viele unserer Eigenschaften E1 bis E4 aus Kapitel 7:

- Zu E1: Es existiert ein klares Komponentenmodell, wie eine Komponente aussehen muss. Dazu gehören – wie gesehen – die Archiv-Datei, das <applet>-Tag, die Ableitung der Einstiegsklasse aus der Klasse JApplet (oder alternativ aus Applet für AWT) und das Überschreiben der Methoden init, start, stop und destroy in der Einstiegsklasse. Die letzten drei Methoden haben wir in unseren Beispielen nicht benutzt.
- Zu E2: Wie im Beispiel gesehen gibt es einen klaren Kopplungsmechanismus für Applets. Dieser ist „komponentenorientierter" als der Mechanismus des Prototyps aus Kapitel 6. Denn zwar muss das Beschaffen der Referenz auf ein Objekt einer anderen Komponente über das Anwenden der Methode getApplet auf den AppletContext programmiert werden (entspricht dem Aufruf von lookup auf den ComponentContext im Prototyp), aber das Anmelden einer Komponente unter einem Namen erfolgt deklarativ durch Angabe des Namens im <applet>-Tag. Wenn also geeignete Applet-Komponenten vorhanden sind, kann man sich vorstellen, dass man lediglich durch Verändern der HTML-Seite bzw. der Jar- oder Zip-Dateien Komponenten hinzufügt, entfernt oder austauscht, wenn dies auch nicht im laufenden Betrieb möglich ist.
- Zu E3: Objekte der Applet-Einstiegsklassen werden nicht von den Komponenten, sondern vom Framework erzeugt. Außerdem existiert ein Lebenszyklus für die Applet-Objekte: Neben der Methode init wird z. B. bei Beendigung des Browsers die Methode destroy aufgerufen. Diese kann man durch Überschreiben anwendungsspezifisch an seine Bedürfnisse anpassen. Weiterhin kann man die Methoden start und stop überschreiben. Früher wurden diese beim Betreten bzw. Wiederbetreten und beim Verlassen einer Seite aufgerufen. Da es heute in der Regel so ist, dass eine Komponente beim Verlassen einer Seite beendet und beim Wiederbetreten komplett neu initialisiert wird, gibt es keinen Unterschied zwischen init und start bzw. zwischen stop und destroy. Aber es bleibt festzuhalten, dass ein Lebenszyklus für Applets existiert. Umgekehrt bietet das Framework den Applet-Komponenten über den AppletContext seine Dienste an. Neben der schon benutzten Methode getApplet kann z. B. mit showStatus etwas in die Statuszeile des Browsers geschrieben oder mit showDocument der Browser veranlasst werden, eine angegebene URL zu laden.
- Zu E4: Eine explizite Angabe darüber, was eine Komponente anbietet und von anderen Komponenten benötigt, gibt es bei Applets nicht.

Da die ersten drei Eigenschaften ziemlich gut erfüllt sind, bewerte ich das Applet-Konzept ohne Vorbehalte als Komponentensystem.

12 Servlets

Servlets sind in gewissem Sinn das Gegenstück zu Applets. Während Applets Java-Programme sind, die in Web-Seiten eingebettet und damit vom Browser (also auf Client-Seite) ausgeführt werden, stellen Servlets Programme dar, die auf der Seite eines Web-Servers ausgeführt werden. Servlets reagieren auf eine Anforderung, die ein Browser in der Regel aufgrund eines Anklickens eines Links oder des Ausfüllens und Absendens eines Formulars an den Web-Server sendet, und generieren dynamisch als Antwort eine Web-Seite. Die dynamisch generierte Web-Seite hängt dabei oft von den Eingaben der Benutzerin, manchmal auch von der Adresse des Rechners, der die Anforderung gesendet hat, dem aktuellen Datum und der Uhrzeit ab. Ein typisches Beispiel für den Einsatz von Servlets sind Online-Shops. Hier könnte ein Benutzer durch einen Klick etwas bestellen. Das Servlet müsste im positiven Fall die Bestellung für die Auslieferung vermerken und eine entsprechende Seite generieren, die dem Benutzer anzeigt, dass die Bestellung erfolgreich durchgeführt wurde.

12.1 Verzeichnisstruktur eines Web-Servers

Die folgenden Ausführungen basieren auf dem Apache-Tomcat-Server in der Version 7. Der Tomcat-Server benutzt die in Bild 12.1 skizzierte Verzeichnisstruktur im Dateisystem.

Das Verzeichnis bin enthält mehrere ausführbare Dateien. In Bild 12.1 sind die beiden Kommando-Dateien startup.bat und shutdown.bat gezeigt, mit denen der Tomcat-Server für ein Windows-Betriebssystem gestartet und wieder angehalten werden kann. Entsprechende Dateien gibt es für die Unix- bzw. Linux-Welt mit den Namen startup.sh und shutdown.sh. Das Verzeichnis conf enthält mehrere Konfigurationsdateien, u. a. die Datei server.xml, in der u. a. festgelegt wird, über welche Portnummer der Web-Server erreichbar ist (die Standardeinstellung ist der Port mit der Nummer 8080). Wir werden auf weitere Dateien, die sich in diesem Verzeichnis befinden können, später noch eingehen. Das Verzeichnis lib enthält Bibliotheksdateien in Form von Jar-Dateien, darunter servlet-api.jar, in der sich die nötigen Klassen und Schnittstellen zur Implementierung von Servlets befinden. Im Verzeichnis logs befinden sich mehrere Log-Dateien, die vom Tomcat-Server geschrieben werden. Die Verzeichnisse temp und work sind Temporär- und Arbeitsverzeichnisse des Tomcat-Servers. Das für Anwendungsentwickler interessanteste Verzeichnis ist webapps. Darin

befindet sich für jede Web-Anwendung ein eigenes Unterverzeichnis. In Bild 12.1 haben diese beispielhaft die Namen application1, application2 und application3. Die Struktur dieser Verzeichnisse ist Teil des Komponentenmodells und wird im folgenden Abschnitt näher erläutert.

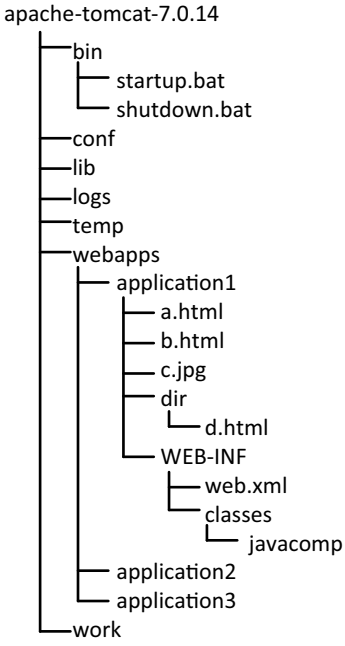

Bild 12.1 Verzeichnisstruktur des Apache-Tomcat-Servers (Version 7)

12.2 Komponentenmodell

12.2.1 Verzeichnisstruktur einer Komponente

Eine Web-Anwendung entspricht einer Komponente. In diesem Kapitel über Servlets bedeuten somit die Bezeichnungen Anwendung oder Web-Anwendung dasselbe wie Komponente oder Web-Komponente. Durch Web-Komponenten kann die Funktionalität eines Web-Servers spezifisch für unterschiedliche Anwendungen erweitert werden. Die Dateien einer Web-Komponente müssen in einer bestimmten Verzeichnisstruktur angeordnet sein. Die Struktur ist unabhängig vom verwendeten Web-Server. Deshalb sollten die folgenden Erläuterungen zum allergrößten Teil unabhängig vom verwendeten Web-Server sein und somit auch für andere Web-Server-Implementierungen gelten.

Im Verzeichnis einer Web-Anwendung sowie in beliebig tief verschachtelten Unterverzeichnissen befinden sich in der Regel HTML-Dateien sowie von diesen HTML-Dateien referenzierte Dateien wie z. B. CSS-Dateien (Cascading Style Sheets), Bild-, Audio- und Videodateien sowie JavaScript- und Jar-Dateien für eingebettete JavaScript-Programme bzw. Applets. In Bild 12.1 kommen beispielhaft drei HTML-Dateien und eine Bilddatei im JPEG-Format vor.

Zum Abrufen einer Datei muss man als URL den Web-Server, dessen Portnummer (falls es nicht die Portnummer 80 ist), den Namen der Komponente (identisch mit dem Namen des Verzeichnisses in webapps) sowie den Pfad zu dieser Datei innerhalb des Anwendungsverzeichnisses angeben. Wenn wir also davon ausgehen, dass der Tomcat-Server auf einem Rechner mit dem Namen www.abc.de gestartet wurde und standardmäßig auf dem Port 8080 lauscht, dann kann die HTML-Datei a.html aus Bild 12.1 unter folgender URL abgerufen werden:

```
http://www.abc.de:8080/application1/a.html
```

Entsprechend lautet die URL für die Datei d.html:

```
http://www.abc.de:8080/application1/dir/d.html
```

Wie in Bild 12.1 zu sehen ist, gibt es auch ein Verzeichnis WEB-INF mit einer Datei web.xml. Man könnte nun analog zu den obigen HTML-Dateien eine URL für web.xml bilden:

```
http://www.abc.de:8080/application1/WEB-INF/web.xml
```

Der Abruf der Datei web.xml mit dieser URL würde aber scheitern, da das Verzeichnis WEB-INF eine Sonderstellung einnimmt. Dieses Verzeichnis und alle seine Inhalte sind von außen über das HTTP-Protokoll nicht sichtbar. Das Verzeichnis WEB-INF dient primär zur Speicherung der Konfigurationsdatei web.xml für diese Web-Komponente sowie der Class-Dateien der Servlets und der von den Servlets benutzten Klassen. Die Class-Dateien müssen in dem Unterverzeichnis classes von WEB-INF in Form einzelner Klassen (mit Package-Pfadnamen) und/oder in dem Unterverzeichnis lib in Form von Jar-Dateien abgelegt werden. Mit anderen Worten: Zum Klassenpfad einer Web-Komponente gehört also das Verzeichnis WEB-INF/classes sowie alle Jar-Dateien in WEB-INF/lib. In Bild 12.1 existiert lediglich das Unterverzeichnis classes. Da der erste Teil der in diesem Buch verwendeten Package-Namen immer javacomp ist, muss also classes ein Verzeichnis namens javacomp besitzen. Die weitere Struktur von javacomp wird der Einfachheit halber in Bild 12.1 nicht gezeigt.

Eine Web-Entwicklerin kann nun direkt auf der vorgestellten Verzeichnisstruktur für die einzelnen Anwendungen bzw. Komponenten unter webapps arbeiten. Es gibt aber auch die beiden folgenden Alternativen:

- Es kann eine War-Datei (das ist eine Jar- oder Zip-Datei mit der Endung .war für „Web Archive“) in webapps abgelegt werden. Der Web-Server entpackt diese Datei dann unter webapps (das ist sehr ähnlich wie beim prototypischen Komponentensystem aus Kapitel 6). Damit die Struktur der Web-Anwendung korrekt ist, muss die War-Datei bereits die richtige Struktur – wie oben beschrieben – haben. Wird die War-Datei wieder gelöscht, so wird die Web-Anwendung samt der entpackten Verzeichnisstruktur ebenfalls gelöscht (auch dies erinnert an das prototypische Komponentensystem).
- Eine weitere Alternative besteht darin, dass das Verzeichnis für eine Web-Anwendung außerhalb des Verzeichnisses webapps angelegt wird, und dass der Tomcat-Server einen Hinweis erhält, wo es ein Verzeichnis für eine weitere Web-Komponente außerhalb von webapps gibt. Dies ist besonders nützlich, wenn man mit einer Entwicklungsumgebung wie Eclipse seine Web-Anwendung erstellt und diese im Workspace seiner Entwicklungsumgebung abspeichern möchte. Diese Variante ist die Alternative meiner Wahl. Ich verwende ausschließlich diese Möglichkeit für die Web-Anwendungen in diesem Buch.

Der soeben erwähnte „Hinweis“, den der Tomcat-Server zum Finden eines weiteren Verzeichnisses einer Web-Komponente braucht, hat die Gestalt einer XML-Datei, die im Verzeichnis conf\Catalina\localhost des Tomcat-Dateibaums abgelegt wird. Hier ist ein Beispiel für eine solche XML-Datei namens application1.xml:

```
<Context path="/application1" reloadable="true"
 docBase="C:\Users\oechsle\EclipseWorkspace\application1"
 workDir="C:\Users\oechsle\EclipseWorkspace\application1\work" />
```

Die XML-Datei enthält lediglich ein XML-Element namens Context. Darin wird der Name der Anwendung, der in der URL vorkommt (hier „ application1“), sowie die Referenz auf das Verzeichnis im Dateisystem, in dem sich die Dateien der Web-Komponente befinden (hier innerhalb meines EclipseWorkspace), und ein Arbeitsverzeichnis spezifiziert. Durch die Angabe reloadable=“true“ wird Hot Deployment aktiviert. In der Regel wird man Hot Deployment nur während der Entwicklungsphase verwenden, um durchgeführte Änderungen ausprobieren zu können, ohne den Tomcat-Server neu zu starten. Im Produktionsbetrieb wird empfohlen, Hot Deployment zu deaktivieren, da es relativ aufwändig für den Web-Server ist, einen weit und tief verzweigten Verzeichnisbaum mit vielen Dateien alle paar Sekunden zu durchsuchen und auf Änderungen zu überprüfen.

12.2.2 Die Konfigurationsdatei web.xml

In der Datei mussten in früheren Versionen die Servlets einer Web-Anwendung konfiguriert werden. Dies ist heute immer noch möglich, aber nicht mehr unbedingt notwendig, da Servlets heute auch mit Hilfe von Annotationen konfiguriert werden können, wovon auch in diesem Buch Gebrauch gemacht werden wird. Im einfachsten Fall besteht die Datei web.xml nur noch aus einem leeren <web-app>-Element:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app>

<web-app>
</web-app>
```

Weitere Einstellungen, die außer der Konfiguration von Servlets in der Datei web.xml vorgenommen werden können, sind z. B. der für Management-Anwendungen anzuzeigende Name und eine dazugehörige Beschreibung für die Web-Komponente sowie eine Angabe, wie eine URL, die nur den Namen der Anwendung enthält, ergänzt werden soll. Die folgende Datei web.xml enthält diese zusätzlichen Einstellungen:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app>

<web-app>
    <display-name>Beispiel einer Web-Anwendung</display-name>
    <description>In diesem Beispiel wird gezeigt ...</description>

    <welcome-file-list>
        <welcome-file>a.html</welcome-file>
    </welcome-file-list>
</web-app>
```

Wenn wir annehmen, dass die obige Datei die Konfigurationsdatei für die Web-Anwendung application1 aus Bild 12.1 ist, dann kann durch die Angabe von a.html im Element <welcome-file> die in a.html angegebene Web-Seite auch durch eine URL abgerufen werden, die nur den Namen der Anwendung enthält:

```
http://www.abc.de:8080/application1
```

12.2.3 Java-Code einer Web-Komponente

Wie für andere Komponentenmodelle auch haben wir jetzt die Verzeichnisstruktur, in der die Dateien einer Web-Komponente organisiert sein müssen, sowie das Format der Konfigurationsdatei kennengelernt. Üblicherweise wird in der Konfigurationsdatei auch die Einstiegsklasse angegeben, von der das Komponenten-Framework ein Objekt erzeugt. Davon haben wir in web.xml bisher nichts gesehen. Bei den Servlets ist es nun so, dass es pro Web-Komponente nicht nur eine Einstiegsklasse geben kann, sondern beliebig viele. Diese Einstiegsklassen benötigen tatsächlich Konfigurationsinformationen, die früher in der web.xml abgelegt werden konnten, die man heute aber auch über Annotationen angeben kann. Die wichtigste Information ist dabei der Bezeichner, der in einer URL angegeben sein muss, damit das Komponenten-Framework (also der Servlets unterstützende Web-Server) entsprechende Methoden auf einem Objekt dieser Klasse aufruft.

Die Servlet-Klassen leitet man am besten von der Klasse HttpServlet ab. Wenn man die Konfiguration durch Annotationen vornimmt, versieht man diese Klasse mit einer @WebServlet-Annotation:

```
package javacomp.servlets.application1;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet("/hello")
public class HelloServlet extends HttpServlet
{
    ...
}
```

Wenn wir annehmen, dass dieses Servlet zu der Web-Komponente application1 aus Bild 12.1 gehört, dann kann durch die Angabe von „/hello“ in der @WebServlet-Annotation ein Objekt dieser Klasse durch diese URL aktiviert werden:

```
http://www.abc.de:8080/application1/hello
```

Mit einer Aktivierung ist gemeint, dass das Framework ein Objekt der Servlet-Klasse erzeugt, falls es noch keines gibt. Von jeder Servlet-Klasse einer Web-Komponente existiert zu jedem Zeitpunkt höchstens ein Objekt. Aktivierung bedeutet auch, dass auf diesem einen Objekt der angesprochenen Servlet-Klasse die Methode doGet oder die Methode doPost aufgerufen wird, je nachdem, ob der Browser ein GET- oder ein POST-Kommando geschickt hat. Wenn man einen Link anklickt oder eine URL in die Adresszeile des Browsers eingibt und abschickt, wird ein GET-Kommando abgesetzt. Wenn man ein Formular ausfüllt und dieses

abschickt, kann ein GET- oder ein POST-Kommando gesendet werden; dies hängt von der Angabe im Formular, das durch HTML beschreiben wird, ab.

Die Protected-Methode doGet der Basisklasse HttpServlet erzeugt eine Fehlerseite mit der Überschrift „HTTP Status 405 – HTTP method GET is not supported by this URL“. Entsprechendes gilt für doPost. Wenn ein Servlet auf HTTP-GET-Kommandos reagieren soll, muss man einfach die Methode doGet in seiner abgeleiteten Servlet-Klasse überschreiben. Entsprechendes gilt für POST-Kommandos. Sowohl doGet als auch doPost werden vom Komponenten-Framework (d. h. dem Web-Server) mit einem Request- und einem Response-Parameter aufgerufen. Über den Request-Parameter kann man Informationen über das HTTP-Kommando herausfinden (u. a. die in einem Formular eingegeben Werte), über den Response-Parameter kann das Servlet die HTTP-Antwort, die vom Web-Server zurückgeschickt wird, gestalten (u. a. den HTML-Text).

In Listing 12.1 ist ein einfaches Servlet dargestellt, das nur auf GET-Kommandos reagiert.

Listing 12.1 Einfaches Hello-Servlet

```
package javacomp.servlets.application1;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet("/hello")
public class HelloServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hallo</title>");
        out.println("</head>");
        out.println("<body>");
        double random = Math.random();
        if(random < 0.25)
        {
            out.println("<h1>Hallo Welt!</h1>");
        }
        else if(random < 0.5)
        {
            out.println("<h1>Hallo Sonne!</h1>");
        }
        else if(random < 0.75)
        {
            out.println("<h1>Hallo Mond!</h1>");
        }
        else
        {
            out.println("<h1>Hallo Mars!</h1>");
        }
```

```
        out.println("</body>");
        out.println("</html>");
    }
}
```

Der Request-Parameter wird bei diesem einfachen Servlet nicht genutzt. Über den Response-Parameter wird der Typ der Antwort („text/html“) gesetzt sowie ein PrintWriter-Objekt beschafft, über das der HTML-Antworttext geschrieben werden kann. Wie zu sehen ist, wird dieser mit Hilfe des Zufalls variiert. Eine mögliche von diesem Servlet erzeugte Seite ist in Bild 12.2 zu sehen. Man kann erkennen, dass in diesem Fall der Firefox-Browser auf demselben Rechner läuft wie der Tomcat-Server, denn als Rechnername für den Tomcat-Server wurde localhost angegeben.

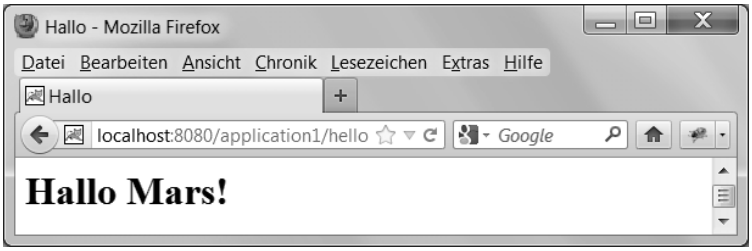

Bild 12.2 Beispiel einer vom Servlet aus Listing 12.1 erzeugten Web-Seite

Falls, wie oben beschrieben wurde, ein Servlet in der URL eines HTTP-Kommandos angesprochen wird und es dafür noch kein Objekt gibt, dann wird es erzeugt. Dies ist das Standardverhalten: Ein Objekt einer Servlet-Klasse wird erst dann erzeugt, wenn es zum ersten Mal benötigt wird. Dieses Standardverhalten kann durch eine Konfigurationseinstellung geändert werden:

```
@WebServlet(value="/hello", loadOnStartup=1)
```

Wenn man in der Annotation @WebServlet für loadOnStartup einen nicht-negativen Wert (d. h. 0 oder größer) angibt, dann wird ein Objekt der Klasse schon erzeugt, wenn die Web-Anwendung installiert bzw. der Web-Server gestartet wird. Wenn eine Web-Komponente mehrere Servlets enthält, dann kann man die Reihenfolge, in der die Objekte dieser Servlets erzeugt werden, durch unterschiedliche Attributwerte von loadOnStartup beeinflussen (Servlets mit kleiner Werten werden vor Servlets mit größeren Werten gestartet, bei gleichen Werten ist die Reihenfolge undefiniert). Falls bei loadOnStartup ein negativer Wert angegeben wird, was die Standardeinstellung ist, wird das Servlet-Objekt erst erzeugt, wenn es zum ersten Mal angesprochen wird.

Wie im obigen Code-Fragment zu sehen ist, kann in der Annotation @WebServlet der Attributname für value jetzt, wo es ein weiteres Attribut gibt, nicht mehr weggelassen werden. Der Typ des Attributs value ist String[]. D. h., es können damit auch mehrere Bezeichner angegeben werden, unter denen das Servlet erreichbar sein soll:

```
@WebServlet(value={"/hello", "/hallo"}, loadOnStartup=1)
```

Statt value kann übrigens auch urlPatterns verwendet werden:

```
@WebServlet(urlPatterns={"/hello", "/hallo"}, loadOnStartup=1)
```

Egal, zu welchem Zeitpunkt ein Servlet-Objekt erzeugt wird, nach dem Erzeugen wird eine Initialisierungsmethode auf das Servlet-Objekt angewendet. In der Basisklasse gibt es zwei überladene Methodenvarianten mit dem Namen init. Wenn das Servlet eine Methode namens init mit einem ServletConfig-Parameter überschrieben hat, wird diese Methode aufgerufen. Falls es eine solche Methode nicht gibt, wird die parameterlose Variante der Init-Methode aufgerufen, sofern vorhanden.

Wenn die Web-Komponente deinstalliert wird oder der Web-Server heruntergefahren wird, dann wird kurz davor die Methode destroy auf allen existierenden Servlet-Objekten einer Web-Anwendung aufgerufen. Die Methode destroy gibt es allerdings nur in der parameterlosen Version. Durch die Methoden init und destroy wird ein einfacher Lebenszyklus für Servlets realisiert. Zusammen mit doGet und doPost sind dies die Methoden, die vom Komponenten-Framework auf den Komponentenobjekten aufgerufen werden.

Umgekehrt können natürlich auch die Komponenten Dienste des Komponenten-Frameworks nutzen. Einige der Dienste stehen über ein ServletContext-Objekt zur Verfügung. Ähnlich wie sich ein Applet über die geerbte Methode getAppletContext Zugriff auf den AppletContext beschaffen kann, erbt ein Servlet die Methode getServletContext und erhält damit eine Referenz auf ein ServletContext-Objekt (AppletContext und ServletContext sind übrigens Schnittstellen, keine Klassen). Zu jeder Web-Anwendung existiert genau ein ServletContext-Objekt. Das heißt, allen Servlets derselben Web-Komponente wird durch getServletContext eine Referenz auf dasselbe Objekt zurückgegeben, während Servlets unterschiedlicher Anwendungen auch unterschiedliche ServletContext-Objekte erhalten. Eine Funktionalität, welche ein ServletContext bereitstellt, ist eine Hash-Tabelle. Mit der Methode setAttribute der Klasse ServletContext kann ein beliebiges Objekt unter einem anzugebenden Namen in die Tabelle eingetragen werden. Durch Angabe des Namens kann man diesen Eintrag mit getAttribute auslesen und mit removeAttribute wieder löschen. Alle in der Tabelle eingetragenen Namen kann man mit getAttributeNames erfragen. Durch diese Hash-Tabelle eines ServletContexts wird ein gemeinsamer Speicher für alle Servlets derselben Web-Anwendung realisiert.

Ein Web-Server kann in der Regel gleichzeitig mehrere Kommandos entgegennehmen. Entsprechend werden die Aufrufe von doGet und doPost auf den Servlet-Objekten durch unterschiedliche Threads realisiert. Es ist also Parallelität vorhanden, auch wenn diese im Programmcode der Web-Komponenten nicht offensichtlich ist. Das heißt also, dass die Methode doGet der Klasse HelloServlet aus Listing 12.1 mehrfach parallel ausgeführt werden kann. Dazu kann es sein, dass die Methoden doGet und doPost anderer Servlets parallel dazu aufgerufen werden. Aus diesem Grund muss der Zugriff auf gemeinsam genutzte Daten synchronisiert werden. Dies gilt insbesondere auch für die Hash-Tabelle des ServletContexts.

Weitere Aspekte von Servlets wie z. B. das wichtige Thema Sessions werden in diesem Buch nicht behandelt. Die Leserinnen und Leser seien hierzu auf spezielle Veröffentlichungen verwiesen (s. auch das Literaturverzeichnis dieses Buchs). Dasselbe gilt auch für die weiterführenden Themenbereiche Java Server Pages und Java Server Faces.

12.3 Erste Beispielkomponente

Die von unserer ersten Web-Komponente realisierte Anwendung stellt sich aus Benutzersicht wie in Bild 12.3 dar.

Bild 12.3 Erste Web-Komponente aus Benutzersicht

Es wird der Wert eines Zählers angezeigt (im Beispiel 0) sowie zwei Buttons, mit denen der Zählerwert erhöht oder auf 0 zurückgesetzt werden kann. Bitte beachten Sie: Wenn die Web-Seite einen gewissen Zählerwert (z. B. 104) anzeigt und Sie klicken auf „Erhöhen“, dann muss nicht notwendigerweise der um eins erhöhte Wert (105) zu sehen sein, sondern es kann auch ein kleinerer Wert (47) oder ein deutlich größerer Wert (543) ausgegeben werden. Dies liegt daran, dass neben Ihnen auch andere Benutzerinnen und Benutzer mit ihren Browsern auf die Anwendung zugreifen können. Wenn die in Ihrem Browser angezeigte Seite schon vor Stunden abgerufen wurde, dann ist der dargestellte Zählerwert unter Umständen seit längerer Zeit nicht mehr gültig (Ihre Web-Seite wird ja nicht automatisch aktualisiert). Das heißt, seit Ihrer letzten Aktion kann der Zähler mehrfach erhöht und zurückgesetzt worden sein.

Die gezeigte Anwendung ist so einfach, dass sie problemlos mit einer einzigen Servlet-Klasse implementiert werden könnte. Um eine etwas größere Anwendung vorzugaukeln, sollen jedoch drei Servlets verwendet werden: eines zum Anzeigen der Seite gemäß Bild 12.3, eines zum Erhöhen und eines zum Zurücksetzen des Zählers. Durch die Aufteilung auf mehrere Servlets muss ein Zugriff auf ein gemeinsames Datum, nämlich den Zähler, realisiert werden. Es bietet sich an, hierfür den vom ServletContext bereitgestellten Speicher zu benutzen. Der aktuelle Zählerwert wird in der Klasse Counter (s. Listing 12.2) gekapselt. Diese Klasse enthält die für diese Anwendung notwendigen Methoden increment, reset und get. Die zusätzliche Methode decrement wird später noch benötigt werden. Wegen eines möglichen parallelen Zugriffs sind alle Methoden synchronized (s. hierzu die Erläuterungen am Ende des letzten Abschnitts).

Listing 12.2 Klasse Counter

```
package javacomp.servlets.application1;

public class Counter
{
    private int counter;

    public synchronized void increment()
    {
        counter++;
    }

    public synchronized void decrement()
    {
        counter--;
    }

    public synchronized void reset()
    {
        counter = 0;
    }

    public synchronized int get()
    {
        return counter;
    }
}
```

Zur Initialisierung könnte man nun in irgendeinem der drei Servlets eine Methode init anlegen und darin ein Counter-Objekt erzeugen und im ServletContext verankern. Um zu garantieren, dass diese Initialisierungsmethode vor den Aufrufen von doGet bzw. doPost der Servlets ausgeführt wird, würde man das Servlet, das die Initialisierung enthält, über loadOnStartup entsprechend konfigurieren. Das wäre die naheliegende und „schönste“ Herangehensweise zur Initialisierung. Um eine weitere Synchronisationsproblematik und ihre Lösung darzustellen, soll im Folgenden jedoch eine andere Variante implementiert werden: Dasjenige Servlet, das als erstes das Counter-Objekt benötigt, erzeugt dieses und legt es im ServletContext ab. Zu diesem Zweck prüft jedes Servlet, ob es schon ein Counter-Objekt im ServletContext gibt. Wenn nicht, wird es erzeugt und gespeichert. Da diese Prüfung aber von mehreren Threads (durch Methodenaufruf auf demselben und/oder unterschiedlichen Servlet-Objekten) parallel durchgeführt werden kann, könnte es sein, dass auch versucht wird die Initialisierung gleichzeitig mehrfach durchzuführen. Als Gegenmaßnahme wird das Prüfen auf die Existenz eines Counter-Objekts und das eventuelle Initialisieren in einen synchronisierten Programmblock gesetzt. Als ein von allen Servlet-Objekten gemeinsam zugreifbares Objekt wird der ServletContext als Synchronisationsobjekt benutzt. Der Code für das Servlet, welches die Web-Seite von Bild 12.3 erzeugt, ist in Listing 12.3 zu sehen.

Listing 12.3 Klasse DisplayServlet

```
package javacomp.servlets.application1;

import java.io.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet("/display")
public class DisplayServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException
    {
        ServletContext myContext = getServletContext();
        Counter counter;
        synchronized(myContext)
        {
            counter = (Counter)myContext.getAttribute("counter");
            if(counter == null)
            {
                counter = new Counter();
                myContext.setAttribute("counter", counter);
            }
        }

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Z&auml;hler</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Realisierung einer " +
                    "Z&auml;hleranwendung</h1>");
        out.println("Der aktuelle Stand des Z&auml;hlers ist " +
                    counter.get() + ".");
        out.println("<p>");

        out.println("<form method=\"get\" action=\"increment\">");
        out.println("<input type=\"submit\"" +
                    "value=\"Erh&ouml;hen\"/>");
        out.println("</form>");
        out.println("<form method=\"get\" action=\"reset\"/>");
        out.println("<input type=\"submit\"" +
                    "value=\"Zur&uuml;cksetzen\">");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
    }
}
```

Im Prinzip würde ein einziges Formular genügen, das die beiden Submit-Buttons zum Erhöhen und Zurücksetzen enthält. Da aber unterschiedliche Servlets durch die beiden Buttons aktiviert werden sollen, benötigen wir zwei Formulare. Wie dem generierten HTML-Text entnommen werden kann, werden beim Drücken der Buttons HTTP-GET-Kommandos an den Web-Server geschickt. Es genügt daher, wenn in allen Servlets dieser Web-Komponente nur die Methode doGet überschrieben wird.

Die Servlets zum Erhöhen bzw. Zurücksetzen des Zählers beschaffen sich eine Referenz auf das Counter-Objekt aus dem ServletContext. Sollte das Counter-Objekt noch nicht existie-

ren, wird es in gleicher Weise erzeugt wie in Listing 12.3 zu sehen. Anschließend muss eine Web-Seite erzeugt werden, die im einfachsten Fall nur einen Text der Art „Änderung wurde durchgeführt“ enthalten könnte. Etwas komfortabler wäre es, wenn zusätzlich der neue Zählerwert angezeigt wird. Wenn dann aber anschließend der Zähler nochmals verändert werden soll, dann müsste man zuerst wieder auf die Seite, die die Submit-Buttons enthält, zurücknavigieren. Bequemer ist es, wenn als Reaktion auf das Erhöhen und Zurücksetzen wieder die Seite aus Bild 12.3 erscheint. Dort sieht man den aktuellen Zählerstand und kann direkt wieder eine Aktion anstoßen. Man könnte nun den Code zur Generierung des HTML-Textes aus Listing 12.3 in die beiden anderen Servlets kopieren. Ich glaube, es ist unnötig zu betonen, dass dies nicht die beste aller denkbaren Lösungen darstellt.

Da die Servlet-Objekte durch das Framework erzeugt werden, hat man in einem Servlet zwar keine Referenz auf die anderen Servlets. Das Framework bietet aber an, eine HTTP-Anfrage an ein anderes Servlet weiterzuleiten, wobei das zuerst aktivierte Servlet vor dem Weiterleiten durchaus noch eigene Aktionen ausführen kann. Das Servlet zum Erhöhen des Zählers macht von dieser Möglichkeit Gebrauch (s. Listing 12.4).

Listing 12.4 Klasse IncrementServlet

```
package javacomp.servlets.application1;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet("/increment")
public class IncrementServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException
    {
        ServletContext myContext = getServletContext();
        Counter counter;
        synchronized(myContext)
        {
            counter = (Counter)myContext.getAttribute("counter");
            if(counter == null)
            {
                counter = new Counter();
                myContext.setAttribute("counter", counter);
            }
        }
        counter.increment();
        RequestDispatcher dispatcher =
                         myContext.getRequestDispatcher("/display");
        dispatcher.forward(request, response);
    }
}
```

Wie dem Programmcode von Listing 12.4 entnommen werden kann, funktioniert das Weiterleiten so, dass man sich über das ServletContext-Objekt einen RequestDispatcher beschafft (dabei wie in @WebServlet die URL relativ zur Anwendung angibt) und auf dieses Request-

Dispatcher-Objekt dann die Methode forward aufruft (vom Request-Parameter könnte man sich den RequestDispatcher alternativ auch beschaffen).

Das Servlet zum Zurücksetzen ist sehr ähnlich und muss deshalb nicht auch in einem Listing dargestellt werden.

Um die Anwendung auszuführen, muss man die Vorgaben bezüglich der Verzeichnisstruktur aus Abschnitt 12.2.1 einhalten, wobei im Verzeichnis WEB-INF eine Datei namens web.xml existieren muss. Wie diese Datei im einfachsten Fall aussehen kann, wurde in Abschnitt 12.2.2 beschrieben. Wenn alle diese Vorgaben eingehalten werden, sollte einer Ausführung dieser Web-Komponente nichts mehr entgegenstehen.

Zum Abschluss dieses Kapitels soll noch auf einen möglichen Effekt hingewiesen werden, der sich durch Parallelität ergeben kann: Wenn ein Thread, der für die Bearbeitung eines bestimmten HTTP-Kommandos zuständig ist, den Zählerwert auf einen bestimmten Wert n setzt (durch Aufruf der Methoden increment oder reset), dann ist nicht sicher, dass der im Anschluss vom selben Thread gelesene Wert (durch Aufruf der Methode get), der auf die Web-Seite geschrieben wird, immer noch n ist. Mit anderen Worten: Man kann sich nicht sicher sein, dass der auf der Web-Seite zu sehende Zählerwert von der unmittelbar vorausgegangenen, eigenen Aktion gesetzt wurde. Dies liegt daran, dass zwischen dem Ändern (increment und reset) und dem Lesen (get) der Zähler durch andere parallel laufende Threads erneut geändert worden sein könnte. Beim Erhöhen fällt das in der Regel nicht auf, da man nicht weiß, welchen Wert der Zähler zu Beginn einer Aktion tatsächlich hat (dieser Sachverhalt wurde oben schon diskutiert). Es könnte aber auffallen, wenn nach einem Drücken auf den Erhöhen-Button der Wert 0 erscheint. Hier würde man naiverweise davon ausgehen, dass (solange das Erniedrigen noch nicht möglich ist) der Wert nach einer Erhöhung immer mindestens 1 sein muss. Ebenso könnte es beim Zurücksetzen dazu kommen, dass in der Antwortseite nicht 0, sondern ein größerer Wert erscheint. Es wäre möglich, den Programmcode so zu ändern, dass dieser Effekt verhindert und garantiert wird, dass immer der Wert zu sehen ist, der von der eigenen Aktion gesetzt wurde. Da dies aber im Kontext dieses Buches nicht relevant ist, soll darauf verzichtet werden.

12.4 Zweite Beispielkomponente

Nach meiner persönlichen Erfahrung sind die einzelnen Web-Komponenten, die sich auf einem Web-Server befinden, unabhängig voneinander. Das heißt: Sie kennen sich nicht, benutzen sich nicht und funktionieren vollkommen unabhängig voneinander. Wie in jedem Komponentensystem ist es aber auch bei Servlets möglich, dass eine Komponente eine andere benutzt. Als zugegebenermaßen etwas künstliches Beispiel soll durch eine weitere Beispielkomponente ermöglicht werden, den Zähler auch wieder zu erniedrigen (die Methode decrement wurde zu diesem Zweck in der Klasse Counter in Listing 12.2 bereits vorgesehen). In der Implementierung müssen zwei Hürden überwunden werden:

1. Es muss von der zweiten Komponente aus auf das Counter-Objekt zugegriffen werden. Dieses Counter-Objekt ist im ServletContext der ersten Anwendung gespeichert. Die zweite Anwendung hat aber ihren eigenen ServletContext.

2. Wenn man eine Referenz auf das Objekt hat, braucht man auch Zugriff auf die Klasse Counter, die Teil der ersten Web-Anwendung ist.

Zur Lösung des ersten Problems ist in der Schnittstelle ServletContext eine Methode mit dem Namen getContext vorhanden, mit der sich ein Zugriff auf ein anderes ServletContext-Objekt beschaffen lässt. Als Parameter muss man dazu den Namen der Anwendung (mit „/“ beginnend) angeben. Durch die folgenden Codezeilen bekommt man also in der zweiten Web-Anwendung Zugriff auf den ServletContext der ersten Anwendung „application1“:

```
ServletContext myContext = getServletContext();
ServletContext otherContext = myContext.getContext("/application1");
```

Dies ist zwar der Code, der am Ende zielführend ist. Ohne zusätzliche Konfigurationseinstellungen wird aber der lokalen Variablen otherContext null zugewiesen. In Abschnitt 12.2.1 wurde erwähnt, dass die von mir bevorzugte Variante der Dateiorganisation so aussieht, dass im Verzeichnis conf\Catalina\localhost für jede Web-Anwendung eine XML-Datei angelegt wird, in der auf das Verzeichnis, in dem sich die Dateien der Anwendung befinden, verwiesen wird. In der XML-Datei muss man nun seiner Anwendung das Recht einräumen, auf andere Komponenten zugreifen zu dürfen. In jeder XML-Datei befindet sich ein <Context>-Element. Dessen Attribut crossConnect, das den Standardwert false hat, muss explizit auf true gesetzt werden. Die XML-Datei für die zweite Web-Komponente sieht damit wie folgt aus:

```
<Context path="/application2" reloadable="true"
 docBase="C:\Users\oechsle\EclipseWorkspace\application2"
 workDir="C:\Users\oechsle\EclipseWorkspace\application2\work"
 crossContext="true"/>
```

Wenn man Zugriff auf einen anderen ServletContext hat, dann ist es nicht nur möglich, über setAttribute, removeAttribute und getAttribute auf dem Speicher der anderen Anwendung zu arbeiten, sondern man kann sich damit auch wieder einen RequestDispatcher von diesem anderen ServletContext geben lassen und damit die HHTP-Anfrage an ein Servlet der anderen Anwendung weiterleiten. In Listing 12.5 wird diese Möglichkeit genutzt. Nach dem Erniedrigen des Zählers wird wieder das DisplayServlet der ersten Web-Komponente aktiviert, um eine Web-Seite mit dem aktuellen Zählerstand und den beiden Buttons zu erzeugen.

Listing 12.5 Klasse DecrementServlet

```
package javacomp.servlets.application2;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;
import javacomp.servlets.application1.Counter;

@WebServlet("/decrement")
public class DecrementServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException
```

```
    {
        ServletContext myContext = getServletContext();
        ServletContext otherContext =
                              myContext.getContext("/application1");
        Counter counter;
        synchronized(otherContext)
        {
            counter = (Counter)otherContext.getAttribute("counter");
            if(counter == null)
            {
                counter = new Counter();
                otherContext.setAttribute("counter", counter);
            }
        }
        counter.decrement();

        RequestDispatcher dispatcher =
                      otherContext.getRequestDispatcher("/display");
        dispatcher.forward(request, response);
    }
}
```

Zur Übersetzung der Klasse DecrementServlet benötigt man klarerweise Zugriff auf die Klasse Counter aus dem Package javacomp.servlets.application1. Wenn man mit Eclipse arbeitet und für die beiden Web-Anwendungen zwei Projekte angelegt hat, muss man im zweiten Projekt das erste Projekt zum Java Build Path hinzufügen. Wenn die Übersetzung dann geklappt hat, kann man die zweite Web-Anwendung im Tomcat-Server installieren und ausprobieren. Für das DecrementServlet haben wir kein Formular mit einem Button vorgesehen. Um das Servlet anzusprechen, müssen wir im Browser die richtige URL angeben. Falls der Browser und der Tomcat-Server wieder auf demselben Rechner ausgeführt werden, lautet die URL wie folgt:

```
http://localhost:8080/application2/decrement
```

Das Ergebnis ist aber enttäuschend, wie Bild 12.4 zeigt.

Dies ist aber auch nicht überraschend, denn wir haben oben zwei zu überwindende Hürden angegeben und davon erst eine überwunden. Das zweite Problem, nämlich der Zugriff auf die Klasse Counter in der zweiten Anwendung, haben wir noch nicht zu lösen versucht. Deshalb gibt es bei der Ausführung des DecrementServlets eine ClassNotFoundException für die Klasse Counter. Das Problem existiert überhaupt, weil auch in diesem Fall wieder jede Web-Komponente von einem eigenen Klassenlader geladen wird. Die Counter-Klasse, die also vom Klassenlader der ersten Web-Anwendung geladen wurde, ist deshalb in der zweiten Anwendung nicht auffindbar. Der Versuch, die Klasse Counter (natürlich im richtigen Package) in die zweite Anwendung mit aufzunehmen, erzeugt einen schon zuvor gesehenen Effekt (s. Bild 12.5).

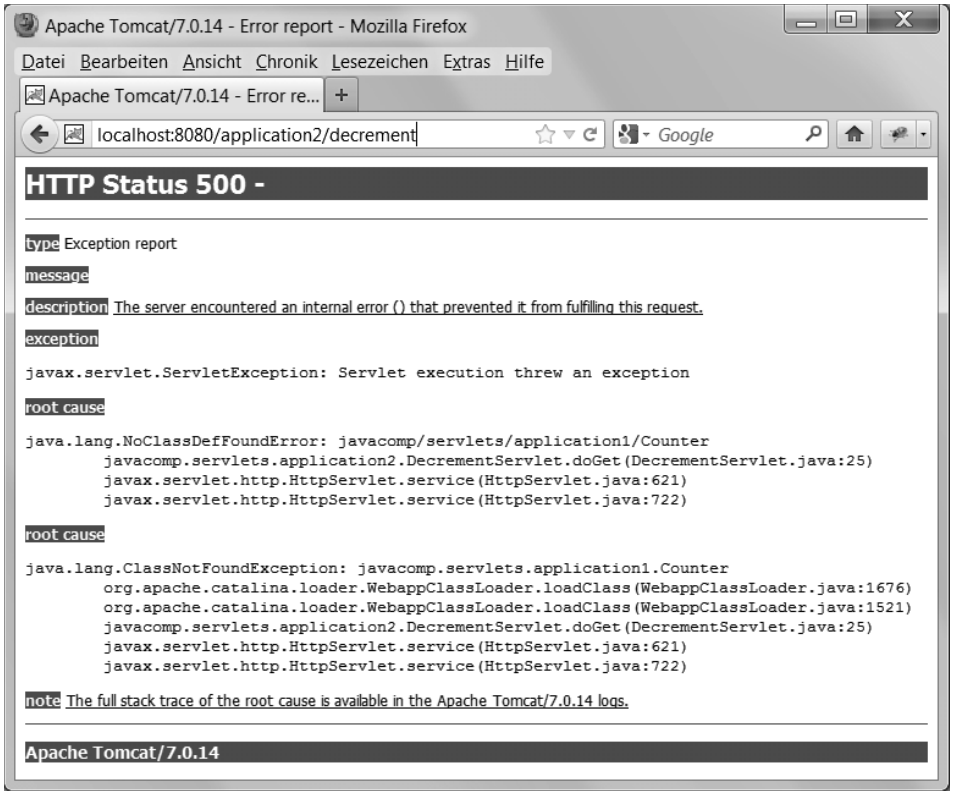

Bild 12.4 1. Versuch der Aktivierung der DecrementServlets

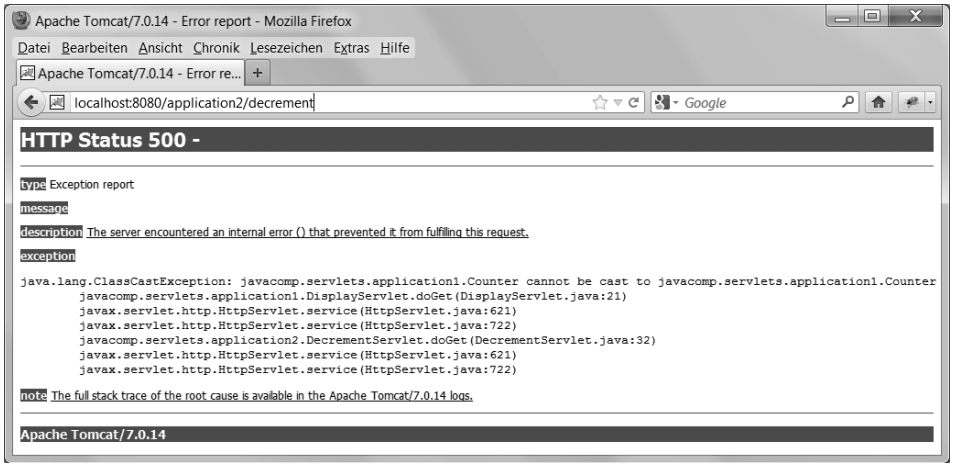

Bild 12.5 2. Versuch der Aktivierung der DecrementServlets

Wir erhalten eine ClassCastException mit der Meldung, dass Counter nicht auf Counter zu casten ist. Darüber wundern wir uns jetzt aber nicht mehr.

Als Lösung packen wir die Class-Datei für die Klasse Counter in eine eigene Jar-Datei und speichern diese im Verzeichnis lib (s. Bild 12.1) des Tomcat-Servers ab. Die ClassLoader sind nämlich in einem Baum angeordnet, der in Bild 12.6 skizziert ist.

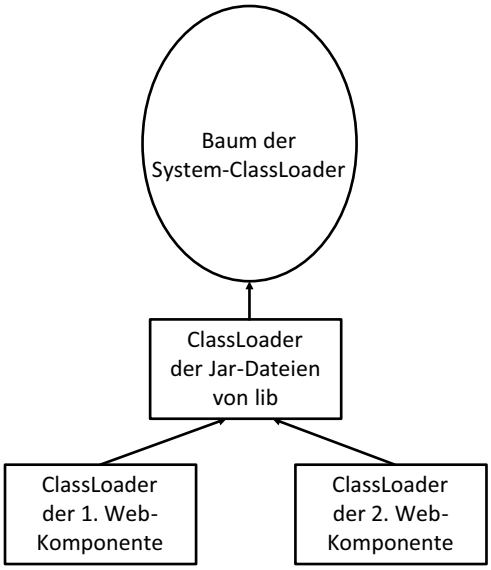

Bild 12.6 Baum der Klassenlader für Web-Anwendungen

Damit alles funktioniert, darf aber die Class-Datei für die Klasse Counter weder in der ersten noch in der zweiten Web-Komponente vorhanden sein. Sie darf also nur in einer Jar-Datei im Verzeichnis lib auftauchen. Dies liegt daran, dass die Klassenlader für die Web-Komponenten nicht genau gleich arbeiten wie bisher beschrieben. Insbesondere funktioniert das Delegieren an den im Baum übergeordneten Klassenlader anders. Wenn also z. B. die Klasse Counter sowohl in der ersten Web-Komponente als auch im Lib-Verzeichnis vorkommt, wird die Klasse Counter vom Klassenlader der ersten Web-Komponente geladen. Wenn dann die Counter-Klasse in der zweiten Web-Anwendung benutzt wird, wird sie entweder vom Klassenlader der zweiten Komponente geladen, falls sie in der zweiten Komponente enthalten ist, oder andernfalls vom Klassenlader des Verzeichnisses lib. In beiden Fällen ist dies aber ein anderer Klassenlader als derjenige der ersten Web-Komponente. Es funktioniert also nur dann richtig, wenn Counter in beiden Web-Anwendungen nicht vorkommt, denn dann wird für beide Anwendungen zum Laden von Counter derselbe Klassenlader, nämlich der für die Jar-Dateien in lib, benutzt.

Damit funktioniert unsere Anwendung, hat aber noch einen kleinen Schönheitsfehler. Um dies zu verstehen, schauen wir uns das Tag <form> für den Erhöhen-Button an:

```
<form method="get" action="increment">
```

Als Aktion ist „increment“ ohne Angabe der Anwendung eingetragen. Das heißt, dass der Browser eine vollständige URL aus der aktuellen URL und der Angabe im <form>-Tag bildet. Nehmen wir an, dass das Formular über diese URL abgerufen wurde:

```
http://localhost:8080/application1/display
```

Wenn man nun auf den Erhöhen-Button klickt, dann bedeutet dies, dass der Browser folgende URL anfordert:

```
http://localhost:8080/application1/increment
```

Dies ist vollkommen korrekt. Das DecrementServlet wird aber über diese URL aktiviert:

```
http://localhost:8080/application2/decrement
```

Wegen der Weiterleitung an das DisplayServlet befinden sich in der Antwortseite wieder dieselben Formulare. Ein Klick auf den Erhöhen-Button führt jetzt aber zur Anforderung der folgenden URL:

```
http://localhost:8080/application2/increment
```

Dafür liefert der Web-Server eine Fehlerseite, weil es diese URL nicht gibt.

Als Lösung wird im Tag <form> für das Attribut action einfach noch die Anwendung ergänzt:

```
<form method="get" action="/application1/increment">
```

Wir ersetzen deshalb die folgende Zeile im DisplayServlet (s. Listing 12.3) der ersten Web-Anwendung

```
out.println("<form method=\"get\" action=\"increment\">");
```

durch diese Zeile:

```
out.println("<form method=\"get\" action=\"" +
            request.getContextPath() + "/increment\">");
```

In gleicher Weise wird das andere Formular, das den Zurücksetzen-Button enthält, geändert.

Nachdem Sie jetzt ein Applet- und ein Servlet-Beispiel für einen Zähler gesehen haben, sollte Ihnen klar sein, dass sich beim Applet-Beispiel der Zähler auf dem Client befindet (jeder Client hat somit seinen eigenen Zähler), während der Zähler beim Servlet-Beispiel auf dem Server beheimatet ist und von allen Clients gemeinsam benutzt wird. Über Servlets, die Sessions benutzen, wäre es auch möglich, für jeden Client einen eigenen Zähler auf dem Server zu halten.

12.5 Bewertung

Auch Servlets mit einem geeigneten Web-Server als Framework erfüllen die meisten der Eigenschaften E1 bis E4 aus Kapitel 7:

- Zu E1: Wie in diesem Kapitel ausführlich erläutert wurde, besteht eine Komponente aus einer klar definierten Verzeichnisstruktur mit einer Konfigurationsdatei im WEB-INF-Verzeichnis. Für eine Web-Komponente kann es mehrere Einstiegsklassen geben, die alle aus HttpServlet abgeleitet werden und in denen die im Text erwähnten Methoden überschrieben werden.

- Zu E2: Wie gesehen existiert ein Kopplungsmechanismus für Servlets; dieser ist aber nicht besonders ausgeprägt. Über getContext des eigenen ServletContexts kann man unter Angabe des Namens einer anderen Web-Komponente Zugriff auf einen anderen ServletContext erhalten. Damit kann man auf die Servlets der anderen Anwendung über einen RequestDispatcher und über die Daten, die in dem ServletContext der anderen Anwendung gespeichert sind, zugreifen.
- Zu E3: Objekte der Servlet-Einstiegsklassen werden vom Web-Server, also vom Framework und nicht von den Anwendungen selbst erzeugt. Es existiert ein einfacher Lebenszyklus für Servlets: Zu Beginn wird init und am Ende destroy aufgerufen. Dazwischen wird abhängig von den HTTP-Kommandos, die beim Web-Server ankommen, beliebig oft doGet und doPost aufgerufen. Das Framework stellt eine Reihe von Funktionen für seine Komponenten bereit. Über den ServletContext wird ein komponentenglobaler Speicher zur Verfügung gestellt. Ferner stellt der ServletContext den Kopplungsmechanismus zu anderen Komponenten bereit. Weitere Dienstleistungen des Frameworks sind die Session-Unterstützung und die Aufrufe der Servlet-Methoden durch unterschiedliche Threads.
- Zu E4: Explizite Angaben über das Angebot und den Bedarf gibt es nicht bei Servlets. Es ist lediglich notwendig, dass eine Web-Komponente, die eine andere Web-Komponente benutzen möchte, dies im Tag <context> durch Setzen des Attributs crossConnect auf true anmeldet. Es muss also die nutzende Komponente angeben, dass sie auf andere Komponenten zugreifen will. Es muss aber nicht angegeben werden, auf welche. Für die genutzten Komponenten muss standardmäßig – und vielleicht überraschenderweise – nichts konfiguriert werden.

Auch Servlets mit einem geeigneten Web-Server stellen für mich zweifellos ein Komponentensystem dar.

13 Enterprise Java Beans (EJB)

EJB ist ein in der Praxis weit verbreitetes Komponentenmodell. Wenn auch der Name Enterprise Java Beans eine Ähnlichkeit zu Java Beans suggeriert, haben beide Komponentenmodelle so gut wie nichts miteinander zu tun. EJB ist ein Komponentenmodell für Anwendungs-Server, wie sie im Rahmen mehrschichtiger Architekturen vorkommen.

13.1 Mehrschichtige Architekturen

Um EJB positionieren zu können, hilft eine kurze Betrachtung der Entwicklung hin zu mehrschichtigen Architekturen. Ursprünglich war eine Anwendung auf einem PC Software, die eine grafische Benutzeroberfläche, die nötige Programmlogik sowie die Datenhaltung umfasste, wobei die Daten in Dateien auf dem PC selbst abgespeichert wurden. Auch wenn dieses Konzept so nie genannt wurde, könnte man es rückblickend als 1-Schicht-Architektur bezeichnen (s. Bild 13.1).

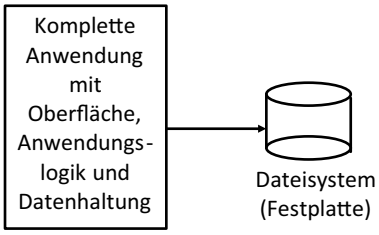

Bild 13.1 1-Schicht-Architektur

Ein Nachteil davon war, dass es nicht möglich war, die Daten von mehreren laufenden Instanzen des Anwendungsprogramms gemeinsam zu nutzen. Dies wurde jedoch möglich, als die Datenhaltung in ein Datenbanksystem ausgelagert wurde, das auf einem separaten Rechner lief. Damit waren neben der gemeinsamen Nutzung der Daten weitere Vorteile verbunden wie zum Beispiel die Synchronisation bei gleichzeitigem Zugriff sowie ein stärkerer Schutz der Konsistenz der Daten im Falle von Abstürzen eines Anwendungsprogramms oder des Datenbank-Rechners. In Bild 13.2 ist die so entstandene 2-Schicht-Architektur zu sehen, wobei nur eine Instanz des Anwendungsprogramms gezeigt wird, obwohl

aber in der Regel mehrere vorhanden sind. Es könnten auch mehrere Datenbanken eingesetzt werden.

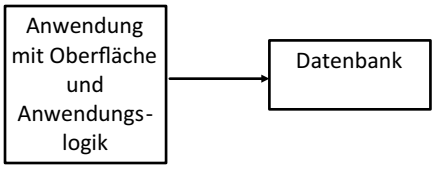

Bild 13.2 2-Schicht-Architektur

Der nächste Schritt war nun die Auslagerung der Anwendungslogik in einen Anwendungs-Server. Die Anwendungsprogramme stellten somit nur noch eine Benutzeroberfläche zur Verfügung und wurden aus diesem Grund auch als „Thin Clients" bezeichnet. Damit war es möglich, Anpassungen an der Programmlogik vorzunehmen, ohne die Software auf den PCs aktualisieren zu müssen. Man spricht in diesem Fall von einer 3-Schicht-Architektur (s. Bild 13.3). Auch in diesem Fall können in jeder Schicht wieder mehrere Instanzen vorkommen, wobei es in der Regel so ist, dass die Anzahl der Instanzen von links nach rechts deutlich abnimmt. Es ist durchaus nicht unüblich, dass es viele Clients gibt, aber nur einen oder zwei Anwendungs-Server und ein einziges Datenbanksystem.

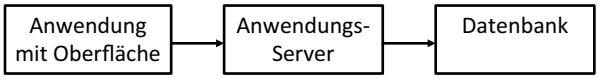

Bild 13.3 3-Schicht-Architektur

Web-Browser dienen heute für viele Anwendungen als Benutzeroberfläche. Man könnte somit einen Web-Browser als universelles Programm für Benutzeroberflächen sehen. Aus einer 3- wird damit eine 4-Schicht-Architektur (s. Bild 13.4). Dabei zeigt der Web-Browser die Oberfläche an; die Reaktion auf Benutzereingaben sowie die Erzeugung der folgenden Web-Seite erfolgt auf dem Web-Server (im Falle von Java durch Servlets).

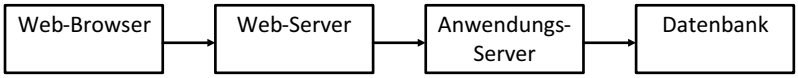

Bild 13.4 4-Schicht-Architektur

Java EE (Enterprise Edition) zielt auf ein solches Szenario ab und liefert eine Lösung für die Java-Welt:

- Java EE beinhaltet die Basis zur Entwicklung von Servlets (s. vorhergehendes Kapitel), mit der Web-Server um anwendungsspezifischen Code erweitert werden können.
- Analog dazu dient EJB zur Entwicklung von anwendungsspezifischem Code, der auf einem Anwendungs-Server ausgeführt werden soll.
- Mit JPA (Java Persistence Architecture) stellt Java EE schließlich noch Unterstützung zum Zugriff auf Datenbanken bereit.

Mit Hilfe des Servlet-Konzepts kann man webbasierte Anwendungen entwickeln, wobei man sich bei der Programmierung auf die spezifische Anwendungslogik konzentrieren kann. Wiederkehrende Aufgaben jeder webbasierten Anwendung wie zum Beispiel die Kommunikation mit dem Web-Browser über das HTTP-Protokoll, das Session-Management und die

Realisierung der parallelen Abarbeitung von HTTP-Anforderungen durch mehrere Threads werden von einem sogenannten Servlet-Container übernommen. Die Situation ist für EJBs ganz ähnlich. Auch hier soll sich die Anwendungsentwicklerin nicht um wiederkehrende Aufgaben kümmern müssen, sondern diese werden vom EJB-Container übernommen. Dazu gehört die Transaktionssteuerung und die Zugriffskontrolle. Da man bei EJB von einer sehr großen Zahl von Clients ausgeht, spielt auch das Management der von den Clients genutzten Objekte eine große Rolle. Darauf wird im Laufe dieses Kapitels noch detaillierter eingegangen.

Eine EJB-Anwendung ist eine Komponente, die im laufenden Betrieb unabhängig von anderen Komponenten installiert, neu installiert und deinstalliert werden kann. Ein EJB-Container bildet die Ausführungsumgebung für EJB-Anwendungen und stellt somit das Komponenten-Framework dar. Java-EE-Server wie Glassfish von der Firma Oracle oder JBoss von der Firma Redhat stellen in der Regel nicht nur einen EJB-Container, sondern auch einen Web-Server mit Servlet-Container sowie ein Datenbanksystem bereit, so dass mit einem einzigen Server die drei Server-Schichten einer 4-Komponenten-Architektur (s. Bild 13.4) abgedeckt werden können.

13.2 Interaktion mit EJB-Komponenten

Die Interaktion zwischen einer Client-Anwendung und einer EJB-Komponente (linker Pfeil in Bild 13.3) bzw. zwischen einem Servlet und einer EJB-Komponente (zweiter Pfeil von links in Bild 13.4) erfolgt in Form eines Methodenaufrufs. Da der Client und die EJB-Komponente in der Regel auf unterschiedlichen Rechnern, zumindest aber in unterschiedlichen Adressräumen (d. h. in unterschiedlichen Prozessen) laufen, handelt es sich bei diesem Methodenaufruf in der Regel um einen Fernmethodenaufruf (RMI: Remote Method Invocation).

Das Prinzip von RMI lässt sich in aller Kürze so zusammenfassen: In einer RMI-Server-Anwendung werden ein oder mehrere RMI-Objekte erzeugt. Ein RMI-Objekt ist ein Objekt einer Klasse, die bestimmte Bedingungen erfüllen muss. Insbesondere muss eine solche Klasse eine Schnittstelle implementieren, die zwar nicht von RMI vorgegeben wird, sondern anwendungsspezifisch vom Anwendungsentwickler definiert werden kann, die aber selbst wieder gewissen Regeln unterworfen ist. Ein RMI-Objekt wird dann vom RMI-Server in einer sogenannten RMI-Registry unter einem frei gewählten Namen angemeldet. In der RMI-Registry wird zu dem gewählten Namen aber nicht das RMI-Objekt selbst, sondern eine Art Fernsteuerung für das Objekt eingetragen. Ein RMI-Client, der in der Regel auf einem anderen Rechner läuft, muss den Rechner, auf dem die RMI-Registry und der RMI-Server laufen, sowie den Namen eines RMI-Objekts kennen. Er kann mit diesem Wissen eine Anfrage an die RMI-Registry stellen und erhält die Fernsteuerung zurück, über die er dann Methoden auf dem RMI-Objekt, welches auf dem RMI-Server liegt, aufrufen kann. Bei der Fernsteuerung handelt es sich um einen sogenannten Stub. Die Klasse des Stub-Objekts implementiert dieselbe Schnittstelle wie die Klasse des RMI-Objekts. Der Programmcode des Stubs ist für alle implementierten Methoden derselbe: Über eine Netzverbindung wer-

den eine Kennung für das RMI-Objekt, die aufzurufende Methode und die Parameter an den RMI-Server übermittelt. Innerhalb des RMI-Servers befindet sich ein sogenanntes Skeleton, welche das Gegenstück des Stubs darstellt. Das Skeleton nimmt die vom Stub übermittelten Daten entgegen, interpretiert sie und ruft die entsprechende Methode mit den übertragenen Parametern auf dem angegebenen RMI-Objekt auf. Nach dem Ende des Methodenaufrufs wird vom Skeleton eine Antwort an den Stub zurückgeschickt, die im Falle einer Methode, die nicht void ist, auch den Rückgabewert oder alternativ die geworfene Ausnahme enthält. Der Stub wartet auf die Antwort, interpretiert sie und kehrt entweder normal vom Methodenaufruf zurück oder wirft die übermittelte Ausnahme.

Die Interaktion eines Clients mit einer EJB-Komponente erfolgt auch über RMI. Es gibt jedoch zwei wesentliche Unterschiede gegenüber dem „normalen“ RMI:

- Bei RMI wird für ein RMI-Objekt ein Eintrag in der RMI-Registry vorgenommen (die Betonung in diesem Satz liegt auf Objekt). Es kann somit auch vorkommen, dass mehrere RMI-Objekte derselben Klasse erzeugt und unter jeweils unterschiedlichen Namen in der RMI-Registry angemeldet werden. Der RMI-Client kann sich durch Angabe des betreffenden Namen gezielt die Fernsteuerung für das gewünschte dazugehörige Objekt beschaffen. Im Gegensatz dazu wird bei EJB ein Eintrag in den Namensdienst für eine Bean-Klasse gemacht (die Betonung liegt hier auf Klasse im Gegensatz zu Objekt). Bean-Klassen sind solche, die „von außen“ über RMI angesprochen werden können. Die Erzeugung und auch das Löschen von Objekten einer Bean-Klasse wird vom EJB-Container und nicht von der Anwendung vorgenommen, wobei es dabei je nach Art der Bean-Klasse unterschiedliche Strategien gibt (dazu später mehr in Abschnitt 13.4).
- Bei RMI ruft das Skeleton direkt die Methode auf dem RMI-Objekt auf. Bei EJB befindet sich dagegen zwischen dem Skeleton und dem Bean-Objekt nochmals ein Proxy-Objekt, das Teil des EJB-Containers ist. Dieses Proxy-Objekt, das wir im Folgenden EJB-Proxy nennen werden, führt vor und nach dem Aufruf der eigentlichen Anwendungsmethode der Bean-Klasse zusätzliche Funktionen aus, die wesentlich für EJB sind und die den Einsatz eines EJB-Servers ausmachen. Dazu gehört u. a. die Auswahl eines geeigneten Objekts, um den Methodenaufruf durchzustellen, wobei ein solches Objekt unter Umständen dabei zuerst erzeugt werden muss. Aber auch andere Funktionen wie das Starten einer neuen Transaktion oder die Überprüfung, ob dem aufrufenden Client erlaubt ist diese Methode auszuführen, werden vom EJB-Proxy erledigt. Wenn man Stub und Skeleton zusammen als Proxy begreift, der die Funktion zur Überschreitung von Adressraum- und Rechnergrenzen erbringt, dann sind bei einem Aufruf einer Bean-Methode zwei Proxies involviert. Diese Situation ist in Bild 13.5 dargestellt. Nur die grau hinterlegten Teile führen dabei anwendungsspezifischen Code aus; der Rest kommt von EJB bzw. RMI.

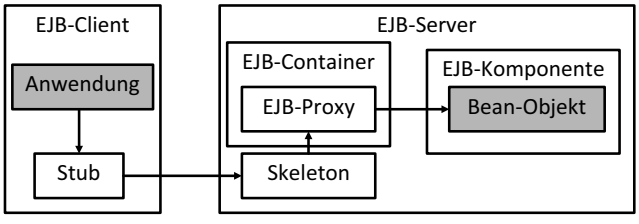

Bild 13.5 Interaktion zwischen EJB-Client und EJB-Server

13.3 Klassenarten einer EJB-Komponente

Es gibt mehrere Arten von Klassen, die in einer EJB-Komponente jeweils mehrfach vorkommen können. Die Kennzeichnung, von welcher Art eine Klasse ist, erfolgte früher über XML-Konfigurationsdateien; heute werden für diesen Zweck in der Regel jedoch vorwiegend Annotationen verwendet:

- Session Beans: Dies sind die „eigentlichen“ Bean-Klassen. Sie waren im vorigen Abschnitt mit Bean-Klassen gemeint. Von ihnen gibt es drei Unterarten, die im folgenden Abschnitt ausführlicher besprochen werden.
- Message-Driven Beans: Wie Session Beans können sie auch „von außen“ aufgerufen werden, allerdings nicht über einen Methodenaufruf, sondern durch das Einstellen einer Nachricht in eine Message Queue. Der EJB-Container holt sich aus der Message Queue die eingegangenen Nachrichten heraus und aktiviert entsprechende Message-Driven Beans. Wir werden diese Bean-Art in diesem Buch nicht weiter betrachten.
- Entities: Entities dienen dem Datenbankzugriff. Mit Hilfe von Entities kann man sich die Programmierung von Datenbankzugriffen mit Hilfe von SQL sparen. Eine Entity-Klasse repräsentiert im einfachsten Fall eine Datenbanktabelle, ein Entity-Objekt eine Zeile einer Tabelle. Allein durch das Ändern von Attributen eines Entity-Objekts können die Daten in der Datenbank verändert werden. Entities hießen früher Entity Beans. Der Begriff Bean fiel vermutlich deshalb weg, weil diese Bean-Art nicht „von außen“ angesprochen werden kann, sondern nur aus Session oder Message-Driven Beans heraus. Wir werden Entities in diesem Buch nur am Rande behandeln, da sie zwar für EJB von großer Bedeutung sind, nicht aber für das Thema Komponenten.
- Restliche Klassen: Weiterhin wird es in größeren EJB-Anwendungskomponenten weitere Hilfsklassen geben, die in keine der oben erwähnten Kategorien fallen und nicht speziell durch Annotationen gekennzeichnet sind.

13.4 Session Beans

Wie im vorigen Abschnitt erwähnt wurde, gibt es drei Unterarten von Session Beans, die wir im Folgenden betrachten wollen.

13.4.1 Stateful Session Beans

Diese Bean-Sorte stellt die „normalen“ Session Beans dar, was sich daran erkennen lässt, dass die Bezeichnung „Stateful Session Bean“ ein Pleonasmus (wie „weißer Schimmel“ oder „kaltes Eis“) ist, denn mit einer Session wird immer ein Zustand verbunden, der zwischen zwei zu einer Session gehörenden Aktionen gespeichert wird. Durch „Stateful“ und „Session“ wird letztlich dasselbe ausgedrückt, weshalb ich im Folgenden nur von „Stateful

Beans“ sprechen werde. Wenn ein Client sich über den Namensdienst mit lookup einen Stub für ein Session Bean beschafft, bekommt er eine Fernsteuerung für ein dazu neu erzeugtes Objekt. Der Client kann von der Vorstellung ausgehen, dass alle Methodenaufrufe, die er über diesen Stub absetzt, immer auf dasselbe Objekt weitergeleitet werden, so dass der Zustand des Bean-Objekts zu Beginn des nächsten Methodenaufrufs derselbe ist wie zum Ende des vorhergehenden Aufrufs. Zwar kann der Client davon ausgehen, als handele es sich immer um dasselbe Bean-Objekt. Tatsächlich kommt jetzt hier eine der Funktionen eines EJB-Servers ins Spiel: Wie schon erwähnt wurde, ist EJB für den Einsatz von sehr vielen Clients gedacht. Wenn sehr viele Clients vorhanden sind, dann bedeutet dies, dass man sehr viele Bean-Objekte benötigt. Wenn manche Clients aber für längere Zeiträume inaktiv sind, weil zum Beispiel bei einer interaktiven Anwendung der Benutzer eine Zeit lang keine Eingaben vornimmt, so kann der EJB-Container die gleichzeitig existierenden Bean-Objekte reduzieren, indem er Objekte, die längere Zeit nicht benutzt worden sind, in einen passiven Zustand versetzt. Dies bedeutet, dass der Zustand des Bean-Objekts abgespeichert und das Bean-Objekt gelöscht wird. Sobald wieder ein Zugriff auf das passive Objekt erfolgt, wird ein neues Objekt erzeugt und dieses auf den abgespeicherten Zustand gesetzt. Für den Client sieht es so aus, als würde er mit demselben Objekt arbeiten, obwohl es tatsächlich ein anderes Objekt ist, was aber für den Client irrelevant ist. In der Stateful-Bean-Klasse können Methoden mit speziellen Annotationen versehen werden, die unmittelbar vor der Passivierung bzw. nach der Reaktivierung aufgerufen werden. Es ist dadurch möglich, anwendungsspezifische Aktionen durchzuführen, die über die vom Container durchgeführte Funktionalität des Abspeicherns und Ladens der Attributwerte des Bean-Objekts hinausgehen.

Methodenaufrufe auf ein und demselben Stateful-Bean-Objekt werden übrigens immer sequenziell durchgeführt. Der Effekt ist also so, als wären alle öffentlichen Methoden als synchronized gekennzeichnet. Parallele Aufrufe von unterschiedlichen Clients können in der Regel nicht vorkommen. Es ist aber möglich, dass innerhalb eines Clients mehrere Threads erzeugt werden, die denselben Stub benutzen. Auf diese Art können tatsächlich parallele Aufrufe auf einem Stateful-Session-Objekt zustande kommen, die aber vom EJB-Container sequenzialisiert werden, wie man leicht ausprobieren kann (s. Abschnitt 13.7).

13.4.2 Stateless Session Beans

So wie der Begriff „Stateful Session Bean“ ein Pleonasmus ist, ist „Stateless Session Bean“ ein Oxymoron, also ein Widerspruch in sich (wie „schwarzer Schimmel“ oder „heißes Eis“), denn mit einer Session ist immer ein Zustand verbunden. Ich spreche daher lieber von einem „Stateless Bean“ (der Name „Service Bean“ wäre auch passend). Wie zuvor erwähnt wurde, muss für jedes Lookup eines Stateful Bean ein neues Bean-Objekt erzeugt werden. Wenn man ein Bean als Stateless kennzeichnet, eröffnet man dem EJB-Container weitaus größere Gestaltungsmöglichkeiten. Der EJB-Container kann nämlich entscheiden, wie viele Bean-Objekte er wann erzeugt und an welches Bean-Objekt ein Methodenaufruf weitergereicht wird. Das heißt, dass es möglich ist, dass zwei Methodenaufrufe, die von einem Client über denselben Stub abgesetzt werden, von demselben Objekt oder aber von unterschiedlichen Bean-Objekten ausgeführt werden.

Wie für Stateful Beans gilt, dass Aufrufe auf demselben Bean-Objekt niemals parallel durchgeführt werden. Wenn ein Methodenaufruf für ein Stateless-Bean-Objekt eintrifft, hat der EJB-Container mehrere Möglichkeiten:

- Der Container hält in der Regel eine Menge von Bean-Objekten zu jeder Stateless-Bean-Klasse in einem Pool. Falls aktuell nicht auf allen Objekten Methodenaufrufe durchgeführt werden, kann er irgendein im Moment nicht benutztes Objekt auswählen und den Methodenaufruf an dieses Objekt durchstellen.
- Falls alle Bean-Objekte im Gebrauch sind, kann er ein neues Objekt erzeugen, das nach seiner Benutzung in den Pool wandert und zur Abarbeitung von Methodenaufrufen desselben oder eines anderen Clients zur Verfügung steht.
- Wenn es kein im Moment unbenutztes Bean-Objekt gibt, der EJB-Container aber der Meinung ist, dass schon genügend Objekte dieser Bean-Klasse vorhanden sind, dann muss er den Aufruf so lange verzögern, bis wieder ein Bean-Objekt zur Verfügung steht.

Der EJB-Container löscht Objekte wieder, wenn er feststellt, dass er nur einen kleinen Teil davon tatsächlich braucht. Die Anzahl der Objekte wird so vom EJB-Container an die aktuelle Belastung angepasst. Wenn die Anzahl der gleichzeitig durchgeführten Aufrufe nicht zu hoch ist, dann ist die optimale Anzahl von Bean-Objekten die Anzahl der parallel auf dieser Bean-Klasse durchgeführten Aufrufe. Wenn also im Extremfall alle Aufrufe immer hintereinander ausgeführt würden, würde ein einziges Objekt reichen.

Ein Stateless Bean muss übrigens nicht tatsächlich zustandslos sein; es kann durchaus Attribute besitzen, die für unterschiedliche Objekte unterschiedliche Werte haben. Es kann aber eben passieren, dass ein Wert, den ein Client durch einen Methodenaufruf in einem solchen Objekt setzt, beim nächsten Aufruf ganz anders ist, weil er beim nächsten Mal mit einem anderen Objekt arbeitet oder weil er zwar mit demselben Objekt arbeitet, der Wert aber durch Methodenaufrufe anderer Clients inzwischen verändert wurde.

Um den EJB-Container die geschilderten Optimierungsmöglichkeiten einzuräumen, sollte man eine Bean-Klasse nach Möglichkeit immer Stateless machen, sofern man die Eigenschaft einer Stateful Bean nicht unbedingt braucht und man damit zurechtkommt, dass sich der Zustand eines Bean-Objekts zwischen zwei Methodenaufrufen desselben Clients ändern kann.

13.4.3 Singleton Session Beans

Diese Bean-Art ist die neueste. Wie der Name ausdrückt, gibt es zu einer Singleton-Bean-Klasse nur genau ein Objekt. Auch dieser Name ist ein Widerspruch in sich, da mit Session die Idee verbunden ist, dass es im Laufe der Zeit mehrere Sessions gibt, häufig auch mehrere Sessions gleichzeitig zu einem Zeitpunkt. Der Name „Singleton Bean“ scheint deshalb eher angebracht. Bei dieser Bean-Sorte wird von dem Prinzip der Stateful und Stateless Beans abgewichen, nämlich von dem, dass auf einem Objekt keine parallelen Methodenaufrufe möglich sind. Auf ein Singleton-Bean-Objekt kann durchaus parallel zugegriffen werden, falls die Anwendungsentwicklerin es erlaubt.

Es gibt zwei Möglichkeiten, um den Grad der Parallelität für ein Singleton Bean festzulegen:

- Container-Managed Concurrency: Diese erste Möglichkeit ist eine deklarative Form, bei der die Synchronisation nicht programmiert, sondern durch Annotationen festgelegt

wird. Sie ist der Standard, falls für eine Singleton-Bean-Klasse nichts anderes angegeben ist. Die Deklaration erfolgt durch die Angabe von Annnotationen in Form von Lese- und Schreibsperren (Read-Write-Locks). Alle Methoden, die mit einer Annotation für eine Lesesperre versehen sind, können gleichzeitig und auch mehrfach parallel aktiv sein, sofern keine Methode mit einer Schreibsperre ausgeführt wird. Eine Methode, die mit einer Schreibsperre annotiert ist, kann niemals parallel mit anderen Methodenaufrufen auf diesem Objekt ausgeführt werden. Wenn keine Annotation angegeben wird, ist die Wirkung so, als wäre eine Schreibsperre angegeben. Wenn man sich also bei einer Singleton-Bean-Klasse überhaupt nicht um die Synchronisation kümmert, gilt automatisch Container-Managed Concurrency mit lauter Schreibsperren, die bewirken, dass keinerlei Parallelität für das Singleton-Bean-Objekt möglich ist. Damit ist die Parallelität so stark wie irgend möglich eingeschränkt; man ist dafür aber auf der sicheren Seite.

- Bean-Managed Concurrency: Wenn man eine Singleton-Bean-Klasse entsprechend annotiert, gibt man bekannt, dass sich der EJB-Container nicht um die Synchronisation kümmern soll. Man muss in diesem Fall die Synchronisation selbst ausprogrammieren (durch Verwendung von synchronized, Locks, Semaphoren oder anderen Synchronisationsmitteln). Wenn man außer der Angabe der Bean-Managed Concurrency nichts weiter unternimmt, hat man den maximalen Grad der Parallelität, die einer Container-Managed Concurrency mit Lesesperren für alle Methoden entspricht.

Wenn man den minimalen Grad an Parallelität (Container-Managed Concurrency nur mit Schreibsperren) erhöht, sollte man sich mit dem Thema Parallelität gut auskennen, denn Synchronisationsfehler lassen sich in der Regel nur sehr schwer erkennen und beheben.

13.5 Komponentenmodell

Eine EJB-Komponente besteht aus beliebig vielen Klassen der in Abschnitt 13.3 erwähnten Sorten, wobei es bei den Session Beans die in Abschnitt 13.4 dargestellten Untersorten gibt. Die Bean-Klassen müssen durch entsprechende Annotationen als solche gekennzeichnet werden (in früheren EJB-Versionen gab es dafür eine spezielle Konfigurationsdatei). Für die Methoden einer Session-Bean-Klasse, die „von außen“ (d. h. von Clients oder von anderen Komponenten) genutzt werden sollen, muss eine Schnittstelle mit der Annotation @Remote definiert werden, die von der Session-Bean-Klasse implementiert wird. Die Schnittstellen und Klassen werden – wie für Java-Komponenten üblich – in eine Jar-Datei gepackt, in der sich ein Verzeichnis namens META-INF befindet mit einer Manifest-Datei. Diese Manifest-Datei muss so gut wie nichts enthalten. Für speziellere Anwendungen kann die Jar-Datei zusätzliche Konfigurationsdateien enthalten, deren Inhalt aber unglücklicherweise von der Art des verwendeten Java-EE-Servers abhängig ist. Das Ziel, Komponenten von einem Server auf einen Server eines anderen Herstellers unverändert zu übernehmen, ist in einem solchen Fall nicht erreichbar. Wir werden im Rahmen dieser Besprechung auf solche Server-spezifischen Konfigurationsdateien nicht eingehen.

Die Installation (Deployment) einer EJB-Komponente kann bei vielen Servern durch das Kopieren der Jar-Datei in einen speziellen Ordner des Servers erfolgen. Alternativ bieten

viele Server auch die Möglichkeit, die Jar-Dateien von EJB-Komponenten über eine webbasierte Administrationsschnittstelle auf den Server zu laden.

Auch bei EJB kann eine Komponente durch eine andere genutzt werden. Wie schon bei anderen Komponentensystemen kann dies unter Umständen bei der gemeinsamen Nutzung von Klassen oder Schnittstellen problematisch sein, da jede EJB-Komponente ihr eigenes Universum bezüglich des Klassenladens hat. Insbesondere, wenn man den Bezug zwischen EJB-Komponenten über Dependency Injection herstellen möchte, besteht eine Möglichkeit zur Lösung dieses Problem darin, die Jar-Dateien mehrerer EJB-Komponenten zu einem „Enterprise Application Project“ in Form einer Ear-Datei zusammenzufassen. Ear steht für „Enterprise Archive“ und ist wie jede Jar- und War-Datei („Java Archive“ bzw. „Web Archive“) eine Zip-Datei. Eine Ear-Datei kann beliebig viele Jar-Dateien enthalten. Da Java-EE-Server nicht nur die Funktionalität eines EJB-Containers, sondern in der Regel immer auch eines Servlet-Containers bereitstellen, können sich außerdem auch noch beliebig viele War-Dateien in einer Ear-Datei befinden. Es ist damit dann zum Beispiel möglich, in einem Servlet eine EJB-Komponente zu nutzen, wobei das Servlet die Referenz über Dependency Injection erhält.

13.6 Erste EJB-Beispielkomponente

Wenn man eine EJB-Komponente implementiert, will man sie natürlich auch ausprobieren. Dafür benötigen wir einen Client. Für die erste Beispielkomponente wird auch der Programmcode eines Clients angegeben. Der Client-Code für die folgenden Beispiele ist sehr ähnlich und wird in diesem Buch nicht wiedergegeben.

In unserer ersten Beispielkomponente wollen wir das bekannte Zähler-Beispiel durch jede der drei Session-Bean-Arten Stateful, Stateless und Singleton Bean implementieren und bei der Benutzung der Beans durch einen Client die Unterschiede zwischen den Bean-Arten demonstrieren.

13.6.1 Server-Seite

Wie zuvor erläutert wurde, braucht eine „von außerhalb“ nutzbare Bean-Klasse eine mit @Remote annotierte Schnittstelle. Für alle Bean-Klassen dieses ersten Beispiels benutzen wir dieselbe Schnittstelle aus Listing 13.1.

Listing 13.1 Counter-Schnittstelle

```
package javacomp.ejb.counter.server;

import javax.ejb.Remote;

@Remote
public interface Counter
{
```

```
    public int increment();
    public int reset();
    public void bye();
}
```

Mit increment soll der Zähler erhöht und mit reset auf 0 zurückgesetzt werden. In beiden Fällen wird der neue Wert zurückgegeben. Die Methode bye dient Demonstrationszwecken und wird später erläutert.

Wir implementieren die Schnittstelle zuerst durch eine Stateful Bean (s. Listing 13.2). Die Implementierung der Methoden increment und reset ist naheliegend. Bitte beachten Sie, dass der Zugriff auf das Attribut nicht synchronisiert werden muss, da der EJB-Container dafür sorgt, dass es keine parallele Ausführung von Methoden auf einem Objekt dieser Klasse gibt (vgl. Abschnitt 13.4.1).

Listing 13.2 Zählerimplementierung in Form einer Stateful-Bean-Klasse

```
package javacomp.ejb.counter.server;

import javax.annotation.*;
import javax.ejb.*;

@Stateful(mappedName="StatefulCounter")
public class StatefulCounterImpl implements Counter
{
    private int counter;

    public int increment()
    {
        counter++;
        return counter;
    }

    public int reset()
    {
        counter = 0;
        return counter;
    }

    @Remove
    public void bye()
    {
    }

    @PostConstruct
    public void constructor()
    {
        System.out.println("created: " + getClass().getName());
    }

    @PreDestroy
    public void destructor()
    {
        System.out.println("will be removed: " +
                           getClass().getName());
    }
}
```

Wie in Listing 13.2 zu sehen ist, ist die Klasse mit @Stateful annotiert. Ich habe die Erfahrung gemacht, dass ohne Angabe von mappedName sich unterschiedliche Server-Implementierungen unterschiedlich verhalten. Bei JBoss 6 wird in einem solchen Fall automatisch der Name „StatefulCounterImpl/remote“ erzeugt, unter dem das Bean über den Namensdienst erreichbar ist. Bei Glassfish funktioniert es aber mit diesem Namen nicht. Um vom Server unabhängig zu sein, ist es deshalb ratsam, den Namen, unter dem die Bean-Klasse im Namensdienst eingetragen wird, explizit anzugeben. Die Methode bye hat keine Funktion. Sie ist jedoch mit der Annotation @Remove versehen. Werden derart annotierte Methoden aufgerufen, dann zeigt der Client damit an, dass er die Session beenden möchte. Man kann in diesem Beispiel gerne mal ausprobieren, was passiert, wenn der Client nach dem Aufruf von bye nochmals reset oder increment aufruft. Dies führt dann zu einer Ausnahme. Die Klasse enthält dann noch zwei weitere Methoden, die nicht Teil der implementierten Schnittstelle sind. Sie sind mit @PostConstruct und @PreDestroy markiert. Wie die Namen der Annotationen andeuten, werden solche Methoden vom EJB-Container aufgerufen, nachdem er ein Objekt dieser Klasse erzeugt hat bzw. bevor er ein Objekt löscht. Man kann so auf der Console des Servers verfolgen, wann Objekte erzeugt und wann welche gelöscht werden.

Die beiden anderen Klassen unterscheiden sich (neben ihrem Klassennamen) nur in der Annotation vor der Klasse (Annotationen @Stateless und @Singleton mit einem jeweils anderen Wert für mappedName). Der Rest des Programmcodes ist exakt identisch. Das Stateless Bean muss übrigens auch nicht synchronisiert werden, weil wie beim Stateful Bean niemals zwei Aufrufe parallel auf dasselbe Objekt ausgeführt werden (vgl. Abschnitt 13.4.2). Dasselbe gilt für das Singleton Bean, aber mit einer anderen Begründung. Auf ein Objekt einer Singleton-Klasse, wovon es ja nur ein einziges gibt, könnte im Prinzip durchaus parallel zugegriffen werden. Wenn man aber bei einem Singleton Bean keine Angaben bezüglich der Synchronisation macht, so gilt Container-Managed Concurrency mit einer Schreibsperre für alle Methoden, wodurch für alle Methoden ein gegenseitiger Ausschluss realisiert wird (vgl. Abschnitt 13.4.3).

Die Class-Dateien für die Schnittstelle und die drei Bean-Klassen werden in eine Jar-Datei gepackt und auf einem Server installiert. Über eine Administrationsschnittstelle des Servers lässt sich überprüfen, ob die Komponente erfolgreich installiert wurde.

13.6.2 Client-Seite

Der Client führt zwei Lookups aus und erhält damit zwei unterschiedliche Referenzen in Form von Stubs. Über beide Referenzen wird jeweils 10 Mal increment durchgeführt. Die Rückgabewerte werden ausgegeben. Der Client-Code ist in Listing 13.3 zu sehen.

Listing 13.3 Counter-Client

```
package javacomp.ejb.counter.client;

import javacomp.ejb.counter.server.*;
import javax.naming.*;

public class CounterClient
```

```
{
    public static void main(String[] args) throws Exception
    {
        Context context = new InitialContext();
        Counter counter1 =
                        (Counter)context.lookup("StatefulCounter");
        Counter counter2 =
                        (Counter)context.lookup("StatefulCounter");

        for(int i = 0; i < 10; i++)
        {
            int newValue1 = counter1.increment();
            int newValue2 = counter2.increment();
            System.out.println(newValue1 + " " + newValue2);
        }
        counter1.bye();
        counter2.bye();
        //System.out.println("reset: value = " + counter1.reset());
    }
}
```

Die Ausgabe des Programms ist nicht besonders spektakulär und wie erwartet. Für jedes Lookup wird ein neues Objekt der Stateful-Bean-Klasse erzeugt, so dass die beiden Referenzen auf unterschiedliche Objekte zeigen. Die Ausgabe ist daher (auch bei mehrmaliger Ausführung):

```
1 1
2 2
3 3
...
10 10
```

Schaut man auf die Console des Servers, so sieht man, dass beim Starten eines Clients zwei Objekte erzeugt werden (Ausgabe der mit @PostConstruct annotierten Methode) und am Ende diese beiden Objekte wieder gelöscht werden (Ausgabe der mit @PreDestroy annotierten Methode). In der auskommentierten Zeile am Ende von main wird nach dem Aufruf der Methode bye, die mit @Remove annotiert ist, nochmals eine Methode auf dem Bean-Objekt aufgerufen. Dies führt im Falle einer Stateful Bean zu einer Ausnahme, wie man leicht ausprobieren kann, wenn man den Kommentar im Client-Code entfernt.

Die sehr einfache Erzeugung eines Context-Objekts im Client, mit dem man Zugriff auf den Namensdienst des Java-EE-Servers erhält, durch Aufruf des parameterlosen Konstruktors der Klasse InitialContext funktioniert übrigens dann, wenn man als Server Glassfish verwendet und Client und Server auf demselben Rechner laufen (wenn also der Rechner, auf dem der Server läuft, aus Sicht des Clients localhost ist). Andernfalls muss man die Initialisierung des Context-Objekts anpassen. Für JBoss 6 muss zum Beispiel die folgende Form der Initialisierung durchgeführt werden, auch wenn Client und Server auf demselben Rechner laufen:

```
Properties properties = new Properties();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
               "org.jnp.interfaces.NamingContextFactory");
properties.put(Context.PROVIDER_URL,
               "localhost:1099");
Context context = new InitialContext(properties);
```

Über ein Properties-Objekt gibt man unterschiedliche Name-Wert-Paare an. Properties ist eine Klasse aus dem Package java.util. Auch müssen zur Ausführung des Clients abhängig vom verwendeten Server unterschiedliche Jar-Dateien verwendet werden.

Wenn wir durch Änderung des Namens in den beiden Lookup-Aufrufen von Listing 13.3 nun das Stateless Bean verwenden, ergibt sich eine ganz andere Ausgabe. Bei der ersten Ausführung kann sich zum Beispiel die folgende Ausgabe ergeben, die bei Ihnen auch ganz anders aussehen kann, z. B. wenn Sie einen anderen Server oder eine andere Version eines Servers verwenden. Die folgende Ausgabe stammt von einem Probelauf mit einem JBoss6-Server:

```
1 1
1 2
2 2
3 3
3 4
4 4
5 5
5 6
6 6
7 7
```

Betrachten wir zum Beispiel nur die linke Spalte (das sind die Werte, die vom Aufruf von increment über den ersten Stub zurückgegeben werden), dann sieht man, dass man nach dem ersten Aufruf 1 erhält, nach dem zweiten Aufruf aber wieder 1. Ganz offensichtlich wurde also beim zweiten Aufruf über den ersten Stub ein anderes Bean verwendet als beim ersten Aufruf. Insgesamt wird der Wert 10 von keinem der Bean-Objekte erreicht. Dies lässt sich erklären, wenn man auf die Ausgabe des Servers auf der Console schaut: Es werden drei Bean-Objekte der Stateless-Bean-Klasse erzeugt. Diese werden offenbar zyklisch benutzt. Die ersten drei Ausgaben sind 1, dann folgt drei Mal 2 usw. Offenbar haben am Ende zwei der Bean-Objekte den Wert 7, das dritte Objekt muss folglich den Wert 6 besitzen, denn es wurde 20 Mal der Wert eines Bean-Objekts erhöht. Wenn wir genau dasselbe Programm nochmals ausführen, ergibt sich auf meinem Notebook die folgende Ausgabe:

```
8 8
7 9
9 8
10 10
9 11
11 10
12 12
11 13
13 12
14 14
```

Über die Server-Console lässt sich ablesen, dass dieses Mal keine neuen Objekte erzeugt werden. Der EJB-Container ist also offenbar der Auffassung, dass er mit den drei im Pool vorhandenen Objekten auskommt. Wie man sieht, wird das Objekt, das als einziges noch den Wert 6 hatte, beim zweiten Aufruf über den ersten Stub verwendet und erhält dadurch auch den Wert 7.

Das Beispiel zeigt, dass das Verhalten bei der Nutzung einer Stateless Bean in diesem Fall relativ schwer kalkulierbar ist. Es sollte beachtet werden, dass niemals behauptet wurde,

dass es sich dabei um ein sinnvolles Beispiel handelt; es geht hier um eine reine Demonstration der Stateless-Eigenschaft eines Beans.

Wiederum eine andere Ausgabe ergibt sich, wenn man schließlich die dritte Bean-Variante verwendet. Gleichgültig, wie viele Lookups man ausführt und wie viele Stubs man sich beschafft, es gibt nur ein einziges Objekt auf dem Server. Entsprechend sieht die Ausgabe so aus:

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
17 18
19 20
```

Und bei erneuter Ausführung des Clients wird dann entsprechend weitergezählt:

```
21 22
23 24
...
```

Die Ausgaben bei der Verwendung des Stateful und des Singleton Beans sollten bei Ihnen auch so aussehen wie gezeigt, unabhängig davon, welchen Server Sie verwenden. Die Ausgabe bei der Nutzung des Stateless Beans kann allerdings abweichen, denn sie hängt von der Strategie des EJB-Containers ab, wie viele Beans er erzeugt und wie er die eintreffenden Methodenaufrufe auf die Bean-Objekte aufteilt. In unserem Beispiel wäre zum Beispiel nur ein einziges Stateless-Bean-Objekt nötig, da es keine parallelen Aufrufe gibt. So erzeugt der Glassfish-Server in der Tat auch nur ein einziges Objekt, so dass für Glassfish die Ausgabe bei einem Stateless wie bei einem Singleton Bean aussieht.

Übrigens bleibt ein Aufruf der Methode bye für die Stateless Beans und das Singleton Bean ganz ohne Wirkung. Das Löschen eines Stateless-Bean-Objekts ist ganz allein Sache des EJB-Containers und kann nicht vom Client beeinflusst werden. Das Singleton Bean wird erst gelöscht, wenn der Server beendet oder die EJB-Komponente deinstalliert wird. Also auch in diesem Fall lässt sich das Löschen nicht von außen anstoßen. Da der Aufruf von bye also wirkungslos ist, führt ein weiterer Aufruf einer Methode wie reset in diesen beiden Fällen nicht zu einer Ausnahme, wie dies beim Stateful Bean der Fall war.

13.7 Zweite EJB-Beispielkomponente

Mit der ersten EJB-Komponente konnten wir experimentieren, wie ein EJB-Container Bean-Objekte erzeugt. Mit der zweiten EJB-Komponente soll nun die Parallelität der Methodenaufrufe untersucht werden. Dazu verwenden wir eine Methode, die eine längere Ausführungszeit hat, wobei diese Zeit (in Sekunden) über einen Parameter angegeben werden kann. Die Methode soll sleep heißen. Die dazugehörige EJB-Schnittstelle ist in Listing 13.4 enthalten.

Listing 13.4 Sleep-Schnittstelle

```
package javacomp.ejb.sleep.server;

import javax.ejb.Remote;

@Remote
public interface Sleep
{
    public void sleep(int secs);
}
```

Die Implementierung ist sehr einfach. Es wird einfach die statische Methode sleep der Klasse Thread benutzt, um die gewünschte Ausführungsdauer zu realisieren. Diese Methode erwartet die Zeitangabe jedoch in Millisekunden; deshalb wird der Parameter mit 1000 multipliziert. In Listing 13.5 ist die Stateless-Variante der Methode zu sehen. Wie im letzten Abschnitt gibt es aber zu dieser Klasse auch wieder die Stateful- und Singleton-Varianten.

Listing 13.5 Sleep-Implementierung in Form einer Stateless-Bean-Klasse

```
package javacomp.ejb.sleep.server;

import javax.ejb.Stateless;

@Stateless(mappedName="StatelessSleep")
public class StatelessSleepImpl implements Sleep
{
    public void sleep(int secs)
    {
        try
        {
            Thread.sleep(secs * 1000);
        }
        catch(InterruptedException e)
        {
        }
    }
}
```

Der Sleep-Client, dessen Code in diesem Buch nicht abgedruckt wird, den Sie aber über die Web-Seite zum Buch beziehen können, erzeugt mehrere Threads, die jeweils die Sleep-Methode auf dem Bean aufrufen. Es werden für alle Bean-Arten jeweils zwei Varianten realisiert: Einmal wird vor der Erzeugung der Threads ein einziges Mal lookup aufgerufen und allen Threads wird dieses eine von lookup zurückgelieferte Stub-Objekt übergeben, womit alle Threads mit demselben Stub-Objekt arbeiten. In der zweiten Variante führt jeder Thread selbstständig sein eigenes Lookup zu Beginn durch, so dass jeder Thread somit seinen eigenen Stub benutzt. Für die drei Bean-Sorten kann man aufgrund der Ausgaben der Threads Folgendes feststellen:

- Stateless Bean: Sinn und Zweck der Stateless Beans ist es gerade, durch eine genügend große Zahl von Bean-Objekten möglichst uneingeschränkte Parallelität zu ermöglichen. Entsprechend erkennt man auch bei der Verwendung des Stateless Beans, dass alle Sleep-Aufrufe parallel ausgeführt werden. Dies stimmt allerdings nur bis zu einer gewissen Zahl an Threads. Wenn man die Zahl sehr stark erhöht, erkennt man, dass manche Auf-

rufe vom EJB-Container verzögert und erst dann ausgeführt werden, wenn andere Aufrufe zu Ende gelaufen sind. Im Prinzip werden die Aufrufe aber parallel bearbeitet, und zwar unabhängig davon, ob alle Threads denselben Stub oder jeder Thread seinen eigenen Stub nutzt.

- Singleton Bean: Wenn man keine zusätzlichen Annotationen angibt, dann werden alle Aufrufe hintereinander ausgeführt; es gibt überhaupt keine Parallelität. Auch hier ist für die beiden Fälle „gemeinsamer Stub“ und „eigene Stubs“ kein Unterschied festzustellen, denn auch wenn jeder Thread seinen eigenen Stub besitzt, wird der Aufruf doch an das einzige Bean-Objekt, das es gibt, weitergeleitet. Die Aufrufe werden zwangsweise sequenzialisiert, weil Container-Managed Concurrency eingestellt ist und beim Aufruf der Methode sleep eine Schreibsperre gesetzt wird. Wenn man die Methode sleep stattdessen explizit mit @Lock(LockType.READ) annotiert, kann man sehen, dass dann die Aufrufe parallel ausgeführt werden.
- Stateful Bean: Bei der Benutzung des Stateful Beans ergibt sich zum ersten und einzigen Mal ein Unterschied zwischen der Benutzung eines einzigen Stubs und mehrerer Stubs, denn bei Stateful Beans wird bekanntlich für jedes Lookup ein eigenes Objekt erzeugt. Im Falle eines einzigen Lookups benutzen folglich alle Threads dasselbe Objekt. Da es keine Parallelität bei Stateful-Bean-Objekten gibt, werden folglich alle Aufrufe hintereinander ausgeführt. Wenn aber jeder Thread sein Lookup selbst durchführt, generiert der EJB-Container für jeden Thread ein neues Objekt der Stateful-Bean-Klasse. Somit können theoretisch alle Aufrufe parallel laufen. Dies ist auch bei der Verwendung eines Glassfish-Servers zu beobachten. Überraschenderweise werden bei JBoss 6 aber auch in diesem Fall die Aufrufe hintereinander abgearbeitet, was nicht notwendig wäre, denn auch der JBoss6-Server erzeugt korrekterweise unterschiedliche Objekte.

Die geschilderten Sacherhalte sind in Tabelle 13.1 in übersichtlicher Form noch einmal zusammengefasst.

Tabelle 13.1 Sequenzielle und parallele Ausführung für unterschiedliche Bean-Arten und bei der Nutzung eines gemeinsamen Stubs bzw. unterschiedlicher Stubs

	gemeinsamer Stub	unterschiedliche Stubs
Stateless Bean	parallel	parallel
Singleton Bean	sequenziell bei Nutzung einer Schreibsperre, parallel bei Nutzung einer Lesesperre	sequenziell bei Nutzung einer Schreibsperre, parallel bei Nutzung einer Lesesperre
Stateful Bean	sequenziell	im Prinzip parallel (bei JBoss6 jedoch sequenziell)

13.8 Dritte EJB-Beispielkomponente (Call-By-Value)

In diesem Beispiel soll demonstriert werden, in welcher Weise die Parameterübergabe bzw. die Übergabe des Rückgabwerts bei den Aufrufen von Bean-Methoden erfolgt. Zunächst betrachten wir wieder eine Counter-Klasse (s. Listing 13.6), die aber im Unterschied zu der Klasse von Listing 13.2 aus Abschnitt 13.6 keine Bean-Klasse ist und deshalb auch keine Schnittstelle implementieren muss. Gemäß Abschnitt 13.3 gehört diese Klasse zu der Sorte Hilfsklasse. Was es mit der Implementierung der Schnittstelle Serializable und der Kennzeichnung mit synchronized auf sich hat, wird später noch erläutert.

Listing 13.6 Counter-Hilfsklasse

```
package javacomp.ejb.counterprovider.server;

import java.io.Serializable;

public class Counter implements Serializable
{
    private int counter;

    public synchronized int increment()
    {
        counter++;
        return counter;
    }
}
```

Unsere EJB-Komponente soll eine Singleton-Bean-Klasse enthalten. Das einzige davon existierende Objekt erzeugt bei seiner Initialisierung ein Objekt der eben gezeigten Klasse Counter und speichert die Referenz darauf in einem Attribut ab. Somit gibt es wie vom Singleton Bean auch nur ein einziges Objekt von der Counter-Klasse auf dem Server. Von außen kann eine Methode aufgerufen werden, die dieses einzige Counter-Objekt zurückliefert. Die Schnittstelle und die dazugehörige Implementierung sind sehr einfach; sie sind in Listing 13.7 und Listing 13.8 zu finden.

Listing 13.7 Schnittstelle CounterProvider

```
package javacomp.ejb.counterprovider.server;

import javax.ejb.*;

@Remote
public interface CounterProvider
{
    public Counter getCounter();
}
```

Listing 13.8 Implementierung der CounterProvider-Schnittstelle

```
package javacomp.ejb.counterprovider.server;

import javax.annotation.*;
import javax.ejb.*;

@Singleton(mappedName="CounterProvider")
public class CounterProviderImpl implements CounterProvider
{
    private Counter counter;

    public Counter getCounter()
    {
        return counter;
    }

    @PostConstruct
    public void init()
    {
        counter = new Counter();
    }
}
```

Wenn wir nun einen Client programmieren und uns darin das Zählerobjekt beschaffen, können wir den Zähler wie erwartet erhöhen. Wenn wir uns aber das Zählerobjekt erneut beschaffen, beginnt die Zählung wieder von vorne, obwohl es doch auf dem Server nur ein einziges Counter-Objekt gibt:

```
Context context = new InitialContext();
CounterProvider cp =
                 (CounterProvider)context.lookup("CounterProvider");
Counter counter = cp.getCounter();
int value = counter.increment(); //value == 1
value = counter.increment(); //value == 2
counter = cp.getCounter();
value = counter.increment(); //value == 1 !!!!
```

Warum ist dies so? Die Antwort ist ganz einfach: Parameter und Rückgabewerte werden bei externen EJB-Methodenaufrufen als Wert übergeben (Call-By-Value). Mit einem externen EJB-Methodenaufruf ist ein Aufruf „von außen“ gemeint (später werden wir auch noch interne EJB-Methodenaufrufe kennenlernen). Durch das Prinzip Call-By-Value bzw. Return-By-Value wird bei jedem Aufruf von getCounter eine neue Kopie angefertigt, die auf dem Client landet. Wenn der Client dann darauf increment aufruft, erhöht sich zwar der Zähler, aber eben nur in dem auf den Client kopierten Objekt, nicht im Original-Objekt auf dem Server. Das heißt, dass man also in diesem Beispiel den Zähler auf dem Server grundsätzlich nicht ändern kann. Insofern ist das Beispiel ziemlich sinnlos. Aber es dient ja nur Demonstrationszwecken. Außerdem wird es im folgenden Abschnitt so verändert, dass sich der Zähler auch auf dem Server hochzählen lässt.

Wegen der Wertrückgabe muss die Klasse Counter aus Listing 13.6 die Schnittstelle Serializable implementieren. Wird „implements Serializable“ weggelassen, so gibt es beim Aufruf der Methode getCounter eine Ausnahme, da der Rückgabewert nicht serialisiert werden kann. Die Implementierung von Serializable ist damit geklärt. Im folgenden Abschnitt wird die Kennzeichnung der Methode increment durch synchronized begründet werden.

13.9 Vierte EJB-Beispielkomponente (Call-By-Reference)

Grundsätzlich kann ein EJB-Bean auch die Rolle eines Clients für ein anderes EJB-Bean einnehmen. Zur Demonstration entwickeln wir eine weitere EJB-Komponente mit der unvermeidbaren HelloWorld-Anwendung. Die Schnittstelle enthält eine Methode namens sayHello mit einem String-Parameter. Die Methode gibt einen String zurück, der einen Gruß für den als Parameter übergebenen Namen darstellt. Die Schnittstelle dazu finden Sie in Listing 13.9.

Listing 13.9 Hello-Schnittstelle

```
package javacomp.ejb.hello.server;

@javax.ejb.Remote
public interface Hello
{
    public String sayHello(String name);
}
```

Wir nehmen an, dass die Aufrufe der Methode sayHello gezählt werden sollen. Der zurückgegebene Gruß soll enthalten, um den wievielten Aufruf der Methode sayHello es sich handelt. Nehmen wir ferner an, dass wir die Schwachstelle des Beispiels aus dem vorigen Abschnitt nicht kennen würden. Deshalb wollen wir die EJB-Komponente CounterProvider aus dem vorigen Abschnitt zum Zählen verwenden.

13.9.1 Getrennte EJB-Jar-Dateien

In unserer ersten Lösungsvariante werden wir die nutzende Komponente, die wir im Folgenden entwickeln, und die genutzte Komponente aus dem vorigen Abschnitt in separate EJB-Jar-Dateien packen und somit für beide Komponenten ein getrenntes Deployment durchführen. In Listing 13.10 ist die erste Variante der Bean-Klasse unserer HelloWorld-Komponente gezeigt. Wir entscheiden uns für die Sorte Stateless Bean und beschaffen uns zu Beginn wie ein Client Zugriff auf den CounterProvider. Vom CounterProvider-Bean lassen wir uns dann bei jedem Methodenaufruf den Zähler geben, um die Methodenaufrufe zu zählen.

Listing 13.10 Implementierung der Hello-Schnittstelle mit Lookup (Variante 1)

```
package javacomp.ejb.hello.server;

import javacomp.ejb.counterprovider.server.*;
import javax.naming.*;
import javax.annotation.PostConstruct;
import javax.ejb.*;

@Stateless(mappedName="Hello")
public class HelloImpl implements Hello
{
```

```
    private CounterProvider cp;

    public String sayHello(String name)
    {
        Counter counter = cp.getCounter();
        return "Hallo " + name + " (Aufruf Nr. " +
               counter.increment() + ")";
    }

    @PostConstruct
    public void init()
    {
        try
        {
            Context context = new InitialContext();
            cp = (CounterProvider)context.lookup("CounterProvider");
        }
        catch(NamingException e)
        {
        }
    }
}
```

Die erste Schwierigkeit mit dieser EJB-Komponente ist, dass der EJB-Container einen Fehler bei der Installation meldet. Der Grund liegt darin, dass wir in dieser Komponente die Schnittstelle CounterProvider und die Klasse Counter verwenden, die sich in einer anderen EJB-Komponente befinden. Damit die Hello-Komponente erfolgreich übersetzt werden konnte, mussten natürlich die Counter- und CounterProvider-Klassen in den Klassenpfad aufgenommen worden sein. Das ist aber eine andere Baustelle. Nun geht es um die Ausführung und nicht um die Übersetzung. Und hier ist es wie auch bei anderen Komponentensystem so, dass jede EJB-Komponente ihren eigenen Klassenlader hat, so dass die Klassen aus einer anderen Komponente im Normalfall nicht verfügbar sind. Dass sich auch hier JBoss 6 wieder anders verhält, soll an dieser Stelle nicht weiter diskutiert werden.

Vielleicht zur Überraschung mancher Leserinnen und Leser hilft es hier, wenn wir die Class-Dateien für die Schnittstelle CounterProvider und die Klasse Counter mit in die EJB-Jar-Datei der Hello-Komponente aufnehmen. Ähnliche Versuche für andere Komponentensysteme in den vorhergehenden Kapiteln führten zu der kuriosen ClassCast-Ausnahme der Art „X kann man nicht zu X casten“. Dies ist hier anders. Der Grund liegt in der Art der Rückgabe des Counter-Objekts durch Call-By-Value: Auf dem Server gibt es ein Counter-Objekt o1, wobei die Counter-Klasse mit einem Klassenlader K1 geladen wurde. Nicht möglich ist es, dieses Objekt o1 auf eine Counter-Klasse zu casten, die mit einem anderen Klassenlader K2 geladen wurde. Aber das passiert hier nicht. Durch Call-By-Value wird ein ganz neues Objekt o2 als Kopie von o1 erzeugt. Dieses neue Objekt ist ein Objekt der Klasse Counter, die mit K2 geladen wurde. Das ist überhaupt kein Problem. Dies ist übrigens auch der Grund, warum wir im vorigen Abschnitt kein Problem in unserem CounterProvider-Client hatten; auch der Client hat natürlich seinen eigenen Klassenlader, aber das Counter-Objekt, das der Client verwendet, wird auch auf dem Client erzeugt.

Somit funktioniert diese Anwendung. Sie tut natürlich nicht das, was sie soll. Denn es wird bei jedem Aufruf „Aufruf Nr. 1“ zurückgegeben, da mit jedem Aufruf von getCounter eine neue Kopie eines Counter-Objekts entsteht. Vielleicht denken nun einige von Ihnen, dass man das Problem ganz leicht beheben kann, indem man den Aufruf von getCounter nicht

bei jedem Aufruf von sayHello wiederholt, sondern nur einmal bei der Initialisierung durchführt, die Referenz auf das Counter-Objekt in einem Attribut speichert und dann bei jedem Aufruf von sayHello dasselbe Counter-Objekt verwendet (s. Listing 13.11).

Listing 13.11 Implementierung der Hello-Schnittstelle mit Lookup (Variante 2)

```
package javacomp.ejb.hello.server;

import javacomp.ejb.counterprovider.server.*;
import javax.naming.*;
import javax.annotation.PostConstruct;
import javax.ejb.*;

@Stateless(mappedName="Hello")
public class HelloImpl implements Hello
{
    private Counter counter;

    public String sayHello(String name)
    {
        return "Hallo " + name + " (Aufruf Nr. " +
               counter.increment() + ")";
    }

    @PostConstruct
    public void init()
    {
        try
        {
            Context context = new InitialContext();
            CounterProvider cp =
                 (CounterProvider)context.lookup("CounterProvider");
            counter = cp.getCounter();
        }
        catch(NamingException e)
        {
        }
    }
}
```

Oberflächliche Tests zeigen scheinbar, dass nun alles in Ordnung ist. Wenn man aber die Belastung des Servers durch mehrere Clients etwas erhöht, kann man zum Beispiel folgende Ausgabe sehen:

```
Hallo Welt (Aufruf Nr. 622)
Hallo Welt (Aufruf Nr. 1)
Hallo Welt (Aufruf Nr. 627)
Hallo Welt (Aufruf Nr. 3)
```

Hier kommt das Zählen offenbar durcheinander. Der Grund liegt darin, dass unser Hello-Bean ein Stateless Bean ist. Das heißt, dass bei entsprechendem Bedarf mehrere Objekte vom EJB-Container erzeugt werden. Jedes Bean-Objekt hat dann aber seinen eigenen Zähler. Man kann an der Ausgabe erkennen, dass nach einer größeren Zahl von Aufrufen ein weiteres Bean-Objekt zum Einsatz gelangte und die Zählung wieder bei 1 begann. Dass die folgenden Nummern weder 623 noch 2 waren, liegt daran, dass parallel ein weiterer Hello-Client zur Erhöhung der Belastung des EJB-Containers lief, bei dem diese Zahlen vorkamen.

Wenn man also nicht die Aufrufe der Methode sayHello zählen wollte, sondern die Aufrufe pro Bean-Objekt, dann wäre die gezeigte Lösung aus Listing 13.11 in Ordnung. Natürlich könnte man diesen Effekt durch ein einfaches Attribut des Typs int wesentlich einfacher erreichen, aber es geht hier um die Nutzung eines Beans durch ein anderes.

Selbstverständlich sollte man bei einem ernsthaften Programm auch die Möglichkeit berücksichtigen, dass die andere Komponente, die man über lookup kontaktiert, nicht vorhanden ist. Ein wesentliches Merkmal von Komponentensystemen ist ja gerade, dass die Komponenten weitgehend unabhängig voneinander installiert und deinstalliert werden können. Deshalb sollte man bei der Programmierung immer von der Möglichkeit ausgehen, dass eine andere Komponente nicht verfügbar ist, und geeignet darauf reagieren. Die NullPointerException, die in Listing 13.11 aus einem solchen Fall resultieren würde, sollte abgefangen werden. Um Codezeilen zu sparen, wurde das im Beispielcode weggelassen.

Aber abgesehen von diesen Unzulänglichkeiten kann diese Komponente durchaus als ein Beispiel dafür dienen, wie eine EJB-Komponente eine andere benutzt:

- Der Kontakt kann wie bei einem Client über einen Context und den Aufruf der Methode lookup hergestellt werden.
- Genutzte Klassen der anderen Komponente müssen auch in die Jar-Datei der nutzenden Komponente aufgenommen werden.

Damit ist eine lose Kopplung von EJB-Komponenten möglich, wobei die einzelnen Komponenten unabhängig voneinander installiert und deinstalliert werden können.

13.9.2 Gemeinsame Ear-Datei mit Dependency Injection

Der EJB-Container bietet für die Software-Entwicklung einen gewissen Komfort in Form einer Dependency Injection an. Wenn man wie im letzten Beispiel in einer EJB-Komponente eine andere Komponente nutzen möchte, muss man die Kontaktaufnahme nicht selbst ausprogrammieren, sondern es genügt, wenn man ein Attribut mit der Annotation @EJB in seine Klasse aufnimmt. Der Typ des Attributs muss die von der anderen Bean implementierte Schnittstelle sein. Das Programm aus Listing 13.11 vereinfacht sich dadurch zu dem aus Listing 13.12.

Listing 13.12 Implementierung der Hello-Schnittstelle mit Dependency Injection

```
package javacomp.ejb.hello.server;

import javacomp.ejb.counterprovider.server.*;
import javax.annotation.PostConstruct;
import javax.ejb.*;

@Stateless(mappedName="Hello")
public class HelloImpl implements Hello
{
    @EJB
    private CounterProvider cp;
    private Counter counter;

    public String sayHello(String name)
```

```
    {
        return "Hallo " + name + " (Aufruf Nr. " +
               counter.increment() + ")";
    }

    @PostConstruct
    public void init()
    {
        counter = cp.getCounter();
    }
}
```

Das Problem war, dass bei meinen Versuchen mit Glassfish die Depenedency Injection nicht funktionierte, wenn man die beiden EJB-Komponenten CounterProvider und Hello in separaten EJB-Jar-Dateien installiert. Wenn man aber die beiden zusammen in eine Ear-Datei packt (vgl. Abschnitt 13.5) und diese Ear-Datei installiert, klappt alles wie erwartet. Die Ear-Datei ist wiederum eine Jar-Datei, die die beiden Jar-Dateien enthält sowie ein Verzeichnis META-INF mit einer Manifest-Datei, in der nichts Wesentliches steht. Die Jar-Datei der Hello-Komponente sollte in diesem Fall die Class-Dateien der CounterProvider-Komponente nicht mehr enthalten, denn die Klassen aller in einer Ear-Datei enthaltenen EJB-Jar-Dateien werden durch einen gemeinsamen ClassLoader geladen.

Für diesen Komfort der Dependency Injection zahlen wir aber einen gewissen Preis. Die EJB-Komponenten werden jetzt zusammen in eine Anwendung gepackt und werden somit zusammen installiert und deinstalliert. Ich würde jetzt das, was sich in der Ear-Datei befindet, als Komponente bezeichnen, denn das ist die Installationseinheit. Die darin enthaltenen Jar-Dateien stellen folglich Teilkomponenten dar.

Übrigens zeigt dieses Beispiel auch ganz schön, warum man die Initialisierung einer Bean nicht im Konstruktor, sondern in einer mit @PostConstruct annotierten Methode durchführen sollte. Wenn der Konstruktor aufgerufen wird, hat das Attribut cp noch den Wert null. Der Aufruf von getCounter im Konstruktor würde folglich zu einer NullPointerException führen. In unserem Beispiel wird aber zuerst das Objekt erzeugt (mit einem leeren Konstruktor), dann wird die Dependency Injection durch den EJB-Container vorgenommen und erst dann wird die Initialisierungsmethode aufgerufen. Zu diesem Zeitpunkt ist cp dann eben nicht mehr null und der Aufruf von getCounter gelingt.

Nun ist es allerdings nach wie vor so, dass die Aufrufe der Methode sayHello nicht richtig gezählt werden. Dieses Problem soll nun im letzten Schritt behoben werden.

13.9.3 Lokale Schnittstellen

Wir führen jetzt an unserer Anwendung eine letzte, sehr harmlos wirkende Änderung durch, die aber eine relativ große Wirkung hat. Wir ändern die Annotation der Schnittstelle CounterProvider von

```
@Remote
```

auf

```
@Local
```

in Listing 13.7. Die @Local-Annotation drückt aus, dass die Schnittstelle CounterProvider nur intern benutzt wird. Mit „intern“ ist gemeint, dass über diese Schnittstelle die Methoden des Beans nur aus Klassen derselben Komponente (d. h. Klassen, die sich in derselben EJB-Jar-Datei oder in derselben Ear-Datei befinden) heraus aufgerufen werden. Der gewaltige Unterschied ist jetzt aber, dass sich die Parameterübergabe und die Übergabe von Rückgabewerten dadurch von Call-By-Value auf Call-By-Reference bzw. von Return-By-Value auf Return-By-Reference ändert. Zur Erinnerung: Es gibt ja nur ein einziges CounterProvider-Objekt (da Singleton Bean) und folglich auch nur ein einziges Counter-Objekt. Wenn jetzt die Methode getCounter eine Referenz zurückliefert, dann haben damit alle Stateless-Bean-Objekte Zugriff auf dieses eine Counter-Objekt. Die Zählung erfolgt jetzt in korrekter Weise.

Wie gesehen kann es mehrere Objekte der Stateless-Bean-Klasse HelloImpl geben (das war ja die Ursache in früheren Implementierungen für das inkorrekte Zählen). Wenn diese Bean-Objekte alle eine Referenz auf dasselbe Counter-Objekt besitzen und die Methode sayHello für diese unterschiedlichen Bean-Objekte parallel aufgerufen wird, dann erfolgt der Zugriff auf das Counter-Objekt in paralleler Weise. Aus diesem Grund wurde die Methode increment in Listing 13.6 vorsorglich bereits mit synchronized versehen.

Übrigens führt die Referenzübergabe zusammen mit der gemeinsamen Nutzung der Klasse Counter in den Teilkomponenten Hello und CounterProvider zu keinem Problem, da die beiden Teilkomponenten in einem Enterprise Application Project in Form einer Ear-Datei zusammengepackt werden und somit alle Klassen durch denselben ClassLoader geladen werden.

Durch die Änderung der Annotation von @Remote in @Local ist es nun natürlich nicht mehr möglich, das CounterProvider-Bean von außen zu nutzen, was in diesem speziellen Fall auch nicht schlimm ist, da eine solche Nutzung sowieso ziemlich sinnlos war. In anderen Fällen kann es aber auch durchaus vorkommen, dass ein Bean sowohl intern über Call-By-Reference als auch extern über Call-By-Value genutzt werden soll. Dies ist jedoch überhaupt kein Problem: Man kann zu diesem Zweck einfach zwei Schnittstellten definieren, die sich außer durch ihren Namen nur in der Annotation @Local und @Remote unterscheiden. In der Bean-Klasse muss man dann einfach angeben, dass die Klasse beide Schnittstellen implementiert. Damit erreicht man mühelos das gewünschte Ziel.

In einer Ear-Datei können sich nicht nur EJB-Jar-Dateien befinden, sondern auch War-Dateien mit Servlets. Es ist dann problemlos möglich, auch in einem Servlet über die Annotation @EJB mit Dependency Injection eine Referenz auf ein Bean zu setzen und dieses Bean im Servlet zu verwenden.

13.10 Entities und Transaktionssteuerung

Neben den Session Beans sind die Entities ein zweiter, äußerst wichtiger Bereich von EJB. Da diese aber für das Komponententhema keine Rolle spielen, wollen wir dieses Thema nur in aller Kürze durch ein sehr einfaches Beispiel streifen.

Durch Entities wird ein Zugriff auf eine Datenbank in einfacher Weise ermöglicht: Entity-Klassen stehen im einfachsten Fall für Datenbanktabellen und Entity-Objekte für Zeilen in

einer Tabelle. Das Erzeugen, Ändern und Löschen von Einträgen in der Datenbank erfolgt ohne die Programmierung von SQL-Anweisungen. Mit Entities wird eine Abbildung aus der objektorientierten in die relationale Welt der Datenbanken realisiert, weshalb man hier auch von einem Objekt-Relationalen Mapper (ORM) spricht. Die JPA (Java Persistence Architecture), welche die Grundlage der Entities bildet, ist dem ORM Hibernate sehr ähnlich. Hibernate kann sogar als eine Form der Realisierung der JPA genutzt werden.

Wie eine Session-Bean-Klasse ist eine Entity-Klasse über eine spezielle Annotation gekennzeichnet. Für Entities ist es die Annotation @Entity. Die Klasse in Listing 13.13 ist eine Klasse, die eine Person mit ihrem Vor- und Nachnamen repräsentiert. Datenbanktabellen benötigen einen Primärschlüssel. Diesem Zweck dient das Attribut id des Typs int in der Klasse Person. Mit der Annotation @Id wird die Funktion des Primärschlüssels angezeigt. Durch die weitere Annotation @GeneratedValue wird festgelegt, dass der Schlüssel automatisch und nicht von der Anwendung erzeugt werden soll.

Listing 13.13 Beispiel einer Entity-Klasse

```
package javacomp.ejb.person.server;

import java.io.Serializable;
import javax.persistence.*;

@Entity
public class Person implements Serializable
{
    @Id
    @GeneratedValue
    private int id;

    private String firstName;
    private String secondName;

    //Getter- und Setter-Methoden für alle drei Attribute
    ...
}
```

Entities werden aus Session Beans heraus genutzt. Für den Zugriff auf die Datenbank benötigt man einen EntityManager. Diesen erhält man am einfachsten durch Dependency Injection mit der Annotation @PersistenceContext. Die Stateless-Bean-Klasse PersonManagerImpl (s. Listing 13.14) besitzt die Methoden createPerson, findPerson und removePerson (die dazugehörige @Remote-Schnittstelle PersonManager ist hier nicht gezeigt). In der Methode createPerson wird ein Objekt der Klasse Person erzeugt und durch Aufruf der Methode persist auf dem EntityManager ein entsprechender Eintrag in der Datenbank erzeugt. Nach dem Abspeichern ist der erzeugte Schlüssel im Entity-Objekt eingetragen. Er wird von der Methode createPerson zurückgegeben. Ein solcher Schlüssel kann später der Methode findPerson übergeben werden. Diese Methode verwendet die Methode find des EntityManagers, welche die Daten aus der Datenbank liest und ein passendes Person-Objekt erzeugt. Die Variante der hier verwendeten Methode find ist eine generische Methode, bei der man neben dem Schlüssel die Klasse des zu erzeugenden Objekts als Parameter angibt; der Rückgabetyp ist dann von diesem Typ, ohne dass man dazu casten muss. Da die Klasse Person aus Listing 13.13 Serializable implementiert, kann eine Kopie eines Person-Objekts an den Aufrufer zurückgegeben werden. Die letzte Methode removePerson löscht – wie ihr

Name sagt – einen Eintrag in der Datenbanktabelle. Dazu wird die Methode remove des EntityManagers verwendet.

Listing 13.14 Stateless-Bean-Klasse PersonManagerImpl

```
package javacomp.ejb.person.server;

import javax.ejb.Stateless;
import javax.persistence.*;

@Stateless(mappedName="PersonManager")
public class PersonManagerImpl implements PersonManager
{
    @PersistenceContext
    private EntityManager em;

    public int createPerson(String firstName, String secondName)
    {
        Person person = new Person();
        person.setFirstName(firstName);
        person.setSecondName(secondName);
        em.persist(person);
        return person.getId();
    }

    public Person findPerson(int id)
    {
        return em.find(Person.class, id);
    }

    public boolean removePerson(int id)
    {
        Person person = findPerson(id);
        if(person == null)
        {
            return false;
        }
        em.remove(person);
        return true;
    }
}
```

Weitere interessante Gesichtspunkte sind unidirektionale und bidirektionale 1:1-, 1:n-, n:1- und m:n-Beziehungen sowie die Abbildung von Vererbungsbeziehungen in die Datenbank. Darauf kann aber hier nicht weiter eingegangen werden.

Wie hoffentlich bekannt spielen Transaktionen eine wichtige Rolle beim Zugriff auf Datenbanken. Der Start und die Beendigung (durch Commit oder Rollback) einer Transaktion werden bei der Einstellung CONTAINER als TransactionManagementType (diese Einstellung gilt, wenn die Bean-Klasse keine Annotation bezüglich der Transaktionssteuerung hat) vom EJB-Container, genauer vom EJB-Proxy-Objekt, durchgeführt. Normalerweise läuft jede Bean-Methode in einer eigenen Transaktion ab. Wenn man dies aber anders haben möchte, kann man es durch Annotierungen der Bean-Methoden erreichen. Da man die Transaktionssteuerung in diesem Fall nicht ausprogrammiert, sondern nur angibt, wie sie sein soll, spricht man von einer deklarativen Transaktionssteuerung. Bei der deklarativen Transaktionssteuerung ist von besonderem Interesse, wenn eine Bean-Methode eine andere Bean-

Methode aufruft. Wenn dieser Aufruf über eine direkte Referenz auf das Bean-Objekt erfolgt (z. B. bei einem Methodenaufruf derselben Klasse und desselben Objekts über this), dann muss man sich darüber im Klaren sein, dass für die aufgerufene Methode keine automatische Transaktionskontrolle erfolgt. Wenn man das haben möchte, muss man die Methode stattdessen indirekt über das EJB-Proxy-Objekt aufrufen.

Alternativ kann man bei der Transaktionssteuerung auf die Dienste des EJB-Containers auch ganz verzichten (Einstellung BEAN als TransactionManagementType in der Klassenannotation). Dann muss der Umgang mit Transaktionen explizit in der Bean-Klasse ausprogrammiert werden.

13.11 Funktionen eines EJB-Containers

Als eine Form einer Zusammenfassung sollen die Funktionen, die ein EJB-Container erbringt, im Folgenden aufgezählt werden:

- Ressourcen-Management: Damit sind primär die Strategien zur Erzeugung und Vernichtung von Stateless-Bean-Objekten und zur Passivierung und Aktivierung von Stateful-Bean-Objekten gemeint, die im Laufe dieses Kapitels demonstriert wurden.
- Nebenläufigkeitskontrolle: Hierunter ist die Steuerung bezüglich der Parallelität zu verstehen. Dazu gehört die Strategie zur Festlegung, wie groß der Grad an Parallelität für eine Stateless Bean sein soll, was sehr eng mit dem Ressourcen-Management zusammenhängt, denn ein höherer Grad an Parallelität kann nur durch eine entsprechende Zahl von Stateless-Bean-Objekten erreicht werden. Weiterhin muss dafür gesorgt werden, dass es keine Parallelität beim Zugriff auf ein Stateful-Bean-Objekt gibt. Und schließlich müssen die Lese- und Schreibsperren für Singleton-Bean-Objekte implementiert werden.
- Namensdienst: Wie gesehen bietet ein EJB-Server einen Namensdienst an. Beim Deployment einer EJB-Komponente müssen die Beans der Komponente in den Namensdienst eingetragen werden. Beim Undeployment müssen die Einträge entsprechend wieder entfernt werden.
- Persistenz: Damit sind die Entities gemeint, die im vorigen Abschnitt skizziert wurden.
- Transaktionssteuerung: Auch die Transaktionssteuerung wurde im vorigen Abschnitt behandelt.
- Zugriffskontrolle: Auf die Zugriffskontrolle wird in diesem Buch nicht näher eingegangen. Sie weist aber viele prinzipielle Ähnlichkeiten mit der Transaktionssteuerung auf. Auch hier hat man die Wahl, die Kontrolle vom EJB-Container durchführen zu lassen, indem man die Methoden mit entsprechenden Annotationen versieht (deklarative Zugriffskontrolle), oder alternativ kann man den Zugriff auf die Dienste eines Beans im Programmcode kontrollieren.

13.12 Bewertung

EJBs sind in vieler Hinsicht vergleichbar mit Servlets. Beide Technologien hängen auch eng zusammen: Beide sind Teil von Java EE. Typische Java-EE-Server enthalten sowohl einen Servlet- als auch einen EJB-Container. Außerdem können in einem Enterprise Application Project Servlet- und EJB-Teilkomponenten zu einer Komponente zusammengefasst werden. Wir gelangen für EJBs auch weitgehend zur selben Einschätzung bezüglich der Erfüllung der Eigenschaften E1 bis E4 aus Kapitel 7 wie für Servlets:

- Zu E1: Eine Komponente kann sowohl eine reine EJB-Komponente (in Form einer EJB-Jar-Datei) oder ein „Enterprise Application Project“ (in Form einer Ear-Datei) sein. Für beide Arten gibt es klare Vorgaben. Die Einstiegsklassen sind die Bean-Klassen, von denen es beliebig viele pro Komponente geben kann. Sie sind durch Annotationen als solche gekennzeichnet und brauchen entsprechende Schnittstellen.
- Zu E2: Der Kopplungsmechanismus für EJBs erfolgt über den Namensdienst. Er kann wie in einem gewöhnlichen Client ausprogrammiert werden oder durch Dependency Injection vom Framework (d. h. vom EJB-Container) unterstützt werden. Bei der Kopplung von Teilkomponenten innerhalb einer Ear-Komponente hat man die Möglichkeit, die Beans auch über @Local-Schnittstellen zu koppeln.
- Zu E3: Objekte der Bean-Einstiegsklassen werden vom EJB-Conainer, also vom Framework und nicht von den Anwendungen selbst erzeugt. Für alle Bean-Arten sind Lebenszyklen definiert, auf die im Buch nicht eingegangen wurde. Diese Lebenszyklen sind für Stateless und Singleton Beans sehr einfach. Für Stateful Beans haben sie etwas mehr Zustände wegen der möglichen De- und Reaktivierung dieser Sorte von Beans. Der EJB-Container bietet eine ganze Reihe von Dienstleistungen an, die weitgehend ohne Zutun der Software-Entwicklerinnen erbracht werden. In speziellen Fällen kann man auch explizit aus einem Bean heraus auf die Dienste des Frameworks zugreifen, indem man Methoden auf einem SessionContext aufruft. Den Zugriff auf einen SessionContext erhält man über Dependency Injection. Da der SessioContext für einfache Beans in der Regel nicht gebraucht wird, wurde er im Buch nicht behandelt.
- Zu E4: Explizite Angaben über das Angebot und den Bedarf sind bei EJBs eher schwach ausgeprägt.

Auch EJBs erfüllen genügend viele der Kriterien, damit diese Technologie nach den Vorgaben dieses Buchs als Komponentensystem gelten kann.

14 Spring

Die Grundidee von Spring ist, Techniken wie Dependency Injection und Proxy-Unterstützung, die beispielsweise bei EJB eingesetzt werden, für alle Arten von Java-Anwendungen in einheitlicher Weise zur Verfügung zu stellen. Statt Dependency Injection wird in Spring auch der Ausdruck „Inversion of Control“ (IoC) verwendet. Damit soll ausgedrückt werden, dass die Steuerung, welches Objekt eine Referenz auf welches andere Objekt bekommt, nicht mehr von der Anwendung, sondern vom Framework durchgeführt wird; diese Steuerungsfunktion wird dadurch quasi auf den Kopf gestellt. Die Proxy-Technik ist für EJB wesentlich, weil damit wichtige Funktionen eines EJB-Containers wie das Ressourcen-Management, die deklarative Transaktionssteuerung und die deklarative Zugriffskontrolle realisiert werden. Die Proxy-Technik kommt in Spring in Gestalt der sogenannten aspektorientierten Programmierung (AOP) vor. AOP steht übrigens für zusätzliche anwendungsabhängige Funktionen auch in EJB zur Verfügung. In diesem Buch wird AOP aber nur im Rahmen von Spring behandelt.

14.1 Komponentenmodell

Das Komponentenmodell von Spring ist nicht besonders stark ausgeprägt, was sicher auch daran liegt, dass Spring in allen möglichen Kontexten („normale“ Java-Anwendungen, webbasierte Anwendungen mit Servlets, Anwendungen mit Datenbank-Zugriff über JPA bzw. Hibernate, Android-Anwendungen usw.) eingesetzt werden soll. Es existiert nicht die Vorstellung, dass eine Komponente in einer Jar-Datei vorliegen muss. Auch spielt die dynamische Installation einer Komponente und der Austausch einer Komponente zur Laufzeit keine herausragende Rolle. Das Komponentenmodell ähnelt sehr stark der Vorstellung von Java Beans. Eine Komponente in Spring ist eine Klasse bzw. ein Objekt einer Klasse, wobei es zu einer Klasse entweder nur ein einziges Objekt oder beliebig viele Objekte geben kann, abhängig von der Konfigurationseinstellung für die betreffende Spring-Komponente.

Eine Spring-Anwendung besteht aus den folgenden Bestandteilen:

- Programmcode in Form von Java-Klassen: Es gibt keine Vorgaben für die Klassen, von denen die Objekte, welche die Komponenten darstellen, stammen. Bei Java Beans gibt es

das auch nicht, aber es gibt doch zumindest gewisse Empfehlungen. Bei Spring ist man ganz frei. Nicht einmal ein parameterloser Konstruktor ist unbedingt nötig.

- Konfigurationsinformationen: Neben dem Programmcode, der sich wie beschrieben aus beliebigen Klassen zusammensetzen kann, sind die Konfigurationsinformationen für eine Spring-Anwendung wesentlich. Diese befinden sich traditionell in einer oder mehreren XML-Dateien. In neueren Spring-Versionen können auch Annotationen verwendet werden. Da aber XML-Dateien immer noch stark verbreitet sind, wird auf ihre Verwendung in diesem Buch nicht verzichtet. Eine Konfigurationsdatei enthält im Wesentlichen eine Liste von Komponenten, die als Beans bezeichnet werden, wobei auch damit die gedankliche Nähe zu den Java Beans ausgedrückt wird. Im Folgenden werden die Begriffe Spring-Komponente und Bean als Synonyme verwendet. Die minimale Angabe für eine Spring-Komponente ist der Name der Klasse, von der ein Objekt erzeugt werden soll. Wenn die Komponente unter einem Namen angesprochen werden soll, was den Regelfall darstellt, bekommt sie auch noch einen frei wählbaren Namen. Weitere Angaben sind u. a., ob es zu der Klasse nur ein Objekt oder mehrere Objekte geben soll, sowie auf welche anderen Komponenten diese Komponente eine Referenz erhalten soll (Dependency Injection).
- Hauptprogramm (für „normale“ Java-Spring-Anwendungen): Im Hauptprogramm wird ein ApplicationContext erzeugt, wobei als Konstruktor-Parameter der Name der Konfigurationsdatei bzw. die Namen mehrerer Konfigurationsdateien angegeben werden. Spring baut dann die Anwendung mit ihren „Verdrahtungen“ auf. Unter Umständen wird damit die Anwendung durch den Aufruf von Konstruktoren, Initialisierungsmethoden und Setter-Methoden schon so weit gestartet, dass keine weiteren Aktionen im Hauptprogramm mehr nötig sind. Es ist aber auch möglich, dass das Hauptprogramm sich unter Angabe des Namens einer Komponente diese vom ApplicationContext geben lässt, um beliebige weitere Methoden auf die Komponenten anzuwenden und damit das Hauptprogramm die Anwendung treibt oder startet. Falls die Spring-Anwendung rein durch Annotationen statt durch XML-Dateien konfiguriert wird, muss man bei der Erzeugung des ApplicationContext ein Class-Objekt angeben, wobei sich in der dadurch repräsentierten Klasse entsprechende Annotationen befinden müssen.

Als einer der wesentlichen Vorteile von Spring wird gesehen, dass die „Verdrahtung“ der Objekte durch Dependency Injection leicht änderbar ist, da sie nicht in den Tiefen der Anwendung verborgen und auf eventuell viele Stellen verteilt ist. Dadurch wird es einfach möglich, unterschiedliche Anwendungen aus denselben Komponenten, aber mit anderen Strukturen aufzubauen. Ferner wird somit auch das Testen wesentlich erleichtert, weil für unterschiedliche Testszenarien jeweils unterschiedliche einfachere Strukturen, eventuell mit zusätzlichen Dummy- oder Mock-Objekten, eingerichtet werden können.

14.2 Erste Spring-Anwendung: Singletons und Prototypes

Unsere erste Spring-Anwendung besteht aus lediglich einer Komponente. Nicht ganz überraschend handelt es sich dabei um einen Zähler (s. Listing 14.1).

Listing 14.1 Spring-Klasse Counter

```
package javacomp.spring.counter;

public class Counter
{
    private int counter;

    public void increment()
    {
        counter++;
    }

    public int get()
    {
        return counter;
    }
}
```

Da weder unsere Anwendung noch das Spring-Framework Threads erzeugen, wurde auf eine Synchronisation verzichtet.

Der zweite Bestandteil ist die Konfigurationsdatei counter.xml. Sie enthält im Wesentlichen eine Deklaration der Counter-Komponente. In vereinfachter Form (d. h. unter Weglassung der zusätzlichen Angaben im Tag <beans>) sieht diese so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    <bean id="counter" class="javacomp.spring.counter.Counter">
    </bean>
</beans>
```

Das Hauptprogramm, in dem die Zählerkomponente genutzt wird, ist in Listing 14.2 zu finden.

Listing 14.2 Spring-Hauptprogramm für Zählerkomponente

```
package javacomp.spring.counter;

import org.springframework.context.*;
import org.springframework.context.support.*;

public class CounterApp
{
    public static void main(String[] args) throws Exception
    {
        ApplicationContext context =
                  new ClassPathXmlApplicationContext("counter.xml");
```

```
        Counter counter = context.getBean("counter", Counter.class);
        for(int i = 0; i < 10; i++)
        {
            counter.increment();
        }
        System.out.println("counter = " + counter.get());
        counter = context.getBean("counter", Counter.class);
        System.out.println("counter = " + counter.get());
    }
}
```

Zunächst wird unter Angabe der XML-Konfigurationsdatei ein ApplicationContext erzeugt. Vom ApplicationContext kann man sich unter Angabe ihres Namens eine Referenz auf die Zählerkomponente aus der Konfigurationsdatei geben lassen. Die Methode getBean ist eine generische Methode. Durch Vorgabe der Klasse Counter als zweites Argument in der Methode getBean kann man sich das Casting ersparen. Man kann aber auch weiterhin die alte Variante verwenden, bei der man casten muss:

```
Counter counter = (Counter) context.getBean("counter");
```

Der Zähler wird danach zehn Mal erhöht. Dann wird sein Wert ausgegeben. Nicht überraschend ist dieser 10. Anschließend wird die Zählerkomponente über getBean erneut vom ApplicationContext beschafft. Über diese neue Referenz wird dann wieder der aktuelle Wert des Zählers ausgegeben. An der erneuten Ausgabe des Werts 10 erkennt man, dass der ApplicationContext bei Angabe des Namens „counter“ jedes Mal dasselbe Objekt zurückgibt. Dies liegt an der Einstellung in der Konfigurationsdatei. Ohne Angabe von scope ist die oben gezeigte Konfigurationsinformation äquivalent mit dieser hier:

```
<bean id="counter" class="javacomp.spring.counter.Counter"
 scope="singleton">
</bean>
```

Im Normalfall sind Spring-Komponenten also Singletons. Das heißt, dass der ApplicationContext nur ein einziges Objekt erzeugt. Anders sieht es bei diesem Inhalt in der Konfigurationsdatei aus:

```
<bean id="counter" class="javacomp.spring.counter.Counter"
 scope="prototype">
</bean>
```

Jetzt wird bei jedem Aufruf von getBean ein neues Objekt erzeugt. Das Programm aus Listing 14.2 gibt in diesem Fall die Werte 10 und 0 aus.

Wenn man für die Klasse Counter einen Konstruktor anlegt, in dem etwas ausgegeben wird, dann kann man sehen, dass im Singleton-Fall das einzige Counter-Objekt bereits beim Erzeugen des ApplicationContext angelegt wird, während im Fall Prototype bei jedem Aufruf von getBean ein Objekt entsteht. Für Singletons kann man allerdings auch die Konfiguration auf „lazy-init“ setzen. In diesem Fall wird dieses eine Objekt nur dann erzeugt, wenn es benötigt wird, und nicht schon standardmäßig beim Verarbeiten der Konfigurationsdatei. Ein Spring-Objekt wird nicht nur dann benötigt, wenn es durch getBean in einer Anwendung angefordert wird (wie in Listing 14.2), sondern auch dann, wenn eine andere Komponente, die erzeugt wird, eine Referenz auf die erste Komponente besitzt (s. folgender Abschnitt).

Bitte machen Sie sich klar, dass durch die Angabe Singleton nicht erzwungen wird, dass es nur ein einziges Objekt der Klasse geben kann. Zum einen kann es für dieselbe Klasse weitere Definitionen von Spring-Komponenten geben (ebenfalls als Singleton oder auch als Prototype). Zum anderen könnten mehrere ApplicationContext-Objekte zur selben XML-Konfigurationsdatei erzeugt werden. Singleton gilt dann für eine Spring-Komponente nur pro ApplicationContext. Und natürlich ist es auch möglich, dass an anderen Stellen im Programmcode explizit weitere Objekte dieser Klasse angelegt werden.

14.3 Zweite Spring-Anwendung: Dependency Injection

Für unsere zweite Spring-Anwendung benötigen wir eine zweite Spring-Komponente, die von der HelloWorld-Sorte ist (s. Listing 14.3).

Listing 14.3 Spring-Klasse Hello (Variante 1)

```
package javacomp.spring.hello;

public class Hello
{
    private String greeting;

    public void setGreeting(String greeting)
    {
        this.greeting = greeting;
    }

    public void sayGreeting(String name)
    {
        System.out.println(greeting + " " + name);
    }
}
```

Im Gegensatz zu unserer ersten Komponente ist diese Komponente durch das Attribut greeting parametrisiert. Der Wert des Attributs kann in der XML-Datei konfiguriert werden:

```
<bean id="hello" class="javacomp.spring.hello.Hello">
    <property name="greeting" value="Hallo"/>
</bean>
```

Oder alternativ auch so:

```
<bean id="hello" class="javacomp.spring.hello.Hello">
    <property name="greeting">
        <value>Hallo</value>
    </property>
</bean>
```

Falls es einen Konstruktor mit einem String-Parameter zum Setzen des Attributs greeting gibt, kann man die Greeting-Komponente so konfigurieren:

```
<bean id="hello" class="javacomp.spring.hello.Hello">
    <constructor-arg value="Hallo"/>
</bean>
```

Oder alternativ auch so:

```
<bean id="hello" class="javacomp.spring.hello.Hello">
    <constructor-arg>
        <value>Hallo</value>
    </constructor-arg>
</bean>
```

Im nächsten Schritt fügen wir der Klasse Hello jetzt noch ein weiteres Attribut des Typs Counter sowie eine Methode zum Setzen des Counter-Attributs hinzu (s. Listing 14.4). Der Zähler zählt die Aufrufe der Methode sayGreeting und fügt die aktuelle Zahl mit in den zurückgegebenen Gruß ein.

Listing 14.4 Spring-Klasse Hello (Variante 2)

```
package javacomp.spring.hello;

import javacomp.spring.counter.Counter;

public class Hello
{
    private String greeting;
    private Counter counter;

    public void sayGreeting(String name)
    {
        counter.increment();
        System.out.println(greeting + " " + name + " (" +
                           counter.get() + ")");
    }

    public void setGreeting(String greeting)
    {
        this.greeting = greeting;
    }

    public void setCounter(Counter counter)
    {
        this.counter = counter;
    }
}
```

Entscheidend ist hierbei, dass wir in der Konfigurationsdatei sowohl eine Counter-Komponente als auch eine Hello-Komponente definieren und die „Verdrahtung“ (d. h. die Vorgabe, dass die Hello-Komponente eine Referenz auf die Counter-Komponente haben soll) ebenfalls in der Konfigurationsdatei festlegen:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans  ...>
```

```
    <bean id="counter" class="javacomp.spring.counter.Counter">
    </bean>
    <bean id="hello" class="javacomp.spring.hello.Hello">
        <property name="greeting" value="Hallo"/>
        <property name="counter" ref="counter"/>
    </bean>
</beans>
```

Wenn in einer Anwendung, die hier nicht abgedruckt ist, die Hello-Komponente auch mehrfach vom ApplicationContext angefordert wird, wird der Zähler bei jedem Aufruf von sayHello wie erwartet hochgezählt. Da für beide Komponenten keine Scope-Angabe vorliegt, gilt für beide Singleton. Es gibt also jeweils nur ein Objekt zu jeder Klasse. Wenn der Scope für Hello auf Prototype gesetzt wird, gibt es bezüglich der Ausgabe keine sichtbare Änderung. Es gibt jetzt zwar mehrere Hello-Objekte durch mehrfaches Aufrufen von getBean mit dem String-Parameter „hello“. Da aber der Scope von Counter immer noch Singleton ist, referenzieren alle Hello-Objekte dasselbe Counter-Objekt. Erst wenn der Scope von Hello auch auf Prototype geändert wird, gibt es zu jedem Hello-Objekt ein eigenes Counter-Objekt. In diesem Fall werden die Aufrufe von sayHello separat für jedes Hello-Objekt gezählt.

14.4 Factory-Methoden und Factory-Beans

Es ist möglich, die Spring-Komponenten nicht durch das Spring-Framework, sondern durch eigenen Code zu erzeugen. Wenn eine Klasse eine statische Methode enthält, in der ein Objekt dieser Klasse angelegt (z. B. durch Nutzung eines privaten Konstruktors) und zurückgegeben wird, kann man durch Angabe des Namens dieser statischen Methode als „factory-method“ im Tag <bean> dem Spring-Framework vorschreiben, dass es diese statische Methode zur Erzeugung von Bean-Objekten dieser Klasse verwenden soll.

Möglich ist auch, dass die Factory-Methode nicht-statisch ist und somit auf ein Objekt angewendet werden muss. Nehmen wir an, die Objekte einer Bean-Klasse X sollen durch die (nicht-statische) Factory-Methode m der Klasse Y erzeugt werden. Die Konfiguration sieht dann so aus, dass man ein Objekt der Klasse Y als Komponente unter einem Namen (zum Beispiel „y“) definiert. Im Tag <bean> für die Klasse X gibt man nicht nur den Namen „m“ als Factory-Methode an, sondern zusätzlich als „factory-bean“ den Namen der Y-Komponente („y“).

14.5 Autowiring

Autowiring bedeutet „automatische Verdrahtung“. Damit ist gemeint, dass man nicht mehr explizit vorgeben muss, auf welches Spring-Objekt ein Attribut eines anderen Spring-Objekts zeigen soll, sondern dass das Spring-Framework dies selbst herausfinden soll. Im zweiten Spring-Beispiel im Abschnitt 14.3 gibt es z. B. nur einen einzigen Eintrag für die Klasse Counter, was übrigens nicht so sein muss. Es ist durchaus möglich, dass es zu

einer Klasse mehrere Bean-Definitionen gibt, typischerweise mit unterschiedlichen Namen und unterschiedlichen Property-Festlegungen, möglicherweise auch mit unterschiedlichen Scope-Einstellungen. Wenn es aber wie in obigem Beispiel nur eine einzige Bean-Definition für die Klasse Counter gibt, kann man in der Konfigurationsdatei bei der Hello-Komponente angeben, dass ein Autowiring über den Typ erfolgen soll:

```
<bean id="hello" class="javacomp.spring.hello.Hello"
 autowire="byType">
    <property name="greeting" value="Hallo"/>
</bean>
```

Durch die Autowiring-Einstellung braucht man keine Angabe für die Property mit dem Namen „counter“. Das Spring-Framework findet selbst heraus, dass es für die Eigenschaft „counter“ des Hello-Beans nur die Möglichkeit gibt, sie auf das Counter-Bean zu setzen, und führt die Dependency Injection in dieser Weise durch.

Die anderen Varianten des Autowiring wie z. B. „byName“ können interessierte Leserinnen und Leser in der Spring-Dokumentation nachschlagen.

14.6 Dritte Spring-Anwendung: Konfiguration durch Annotationen

Die Konfiguration einer Spring-Anwendung kann statt durch XML-Dateien ausschließlich mit annotierten Java-Klassen vorgenommen werden. Ein Beispiel einer solchen Klasse für die Spring-Anwendung aus Abschnitt 14.3 zeigt Listing 14.5.

Listing 14.5 Spring-Konfigurationsklasse mit @Bean-Annotationen

```
package javacomp.spring.annoconfig;

import javacomp.spring.counter.Counter;
import javacomp.spring.hello.Hello;
import org.springframework.context.annotation.*;

public class HelloConfig
{
    @Bean
    public Hello hello()
    {
        Hello result = new Hello();
        result.setGreeting("Hallo");
        return result;
    }

    @Bean
    public Counter counter()
    {
        return new Counter();
    }
}
```

Wie in Listing 14.5 zu sehen ist, gibt es statt eines <bean>-Tags eine mit @Bean annotierte Methode, in der ein Objekt einer Bean-Klasse angelegt wird. Die Möglichkeit der im vorhergehenden Abschnitt besprochenen automatischen Verdrahtung kann man auch in diesem Fall in Anspruch nehmen, wenn man die Methode setCounter der Klasse Hello mit @Autowired annotiert:

```
@Autowired
public void setCounter(Counter counter)
{
    this.counter = counter;
}
```

Die Erzeugung des ApplicationContext muss jetzt wie folgt geändert werden:

```
ApplicationContext context =
          new AnnotationConfigApplicationContext(HelloConfig.class);
```

Die restliche Anwendung kann so bleiben, wie sie war. Insbesondere lassen sich auch von diesem ApplicationContext die Spring-Komponenten über die Methode getBean in genau derselben Weise wie zuvor beschaffen.

Die restlichen Spring-Anwendungen werden wieder durch XML-Dateien konfiguriert.

14.7 Vierte Spring-Anwendung: BeanPostProcessor

In einer XML-Konfigurationsdatei können ein oder mehrere BeanPostProcessor-Objekte als Spring-Komponenten definiert werden. Ein BeanPostProcessor ist eine Klasse, welche die Schnittstelle BeanPostProcessor des Spring-Frameworks (s. Listing 14.6) implementiert.

Listing 14.6 Schnittstelle BeanPostProcessor des Spring-Frameworks

```
package org.springframework.beans.factory.config;

public interface BeanPostProcessor
{
    public Object postProcessBeforeInitialization(Object bean,
                                                   String name)
        throws BeansException;
    public Object postProcessAfterInitialization(Object bean,
                                                  String name)
        throws BeansException;
}
```

Das Spring-Framework erkennt selbstständig, ob es sich bei einer Spring-Komponente um einen BeanPostProcessor handelt oder nicht (Autodetection). Wenn es ein BeanPostProcessor ist, wird dieser registriert. Nach der Erzeugung jeder „normalen“ Spring-Komponente werden dann die beiden Methoden der BeanPostProcessor-Schnittstelle auf allen registrierten BeanPostProcessor-Objekten aufgerufen. Diesen Methoden wird jeweils das soeben neu

erzeugte Bean-Objekt selbst sowie sein Name übergeben. Die Methode kann dann zum Beispiel das Objekt in irgendeiner Form manipulieren (zum Beispiel weitere anwendungsabhängige Initialisierungen durchführen). Entscheidend ist, dass das Spring-Framework im Folgenden das Objekt benutzt, das von den Methoden der BeanPostProcessor-Schnittstelle zurückgegeben wird. Wenn also eine Methode das Objekt zurückgibt, das es als Parameter übergeben bekommen hat, dann wird im Folgenden tatsächlich das Objekt verwendet, das vom Spring-Framework erzeugt wurde. Eine Methode eines BeanPostProcessors kann aber auch ein ganz anderes Objekt zurückgeben. Im Folgenden wird dann dieses Objekt benutzt statt desjenigen, das das Spring-Framework angelegt hat. Der typische Anwendungsfall für einen solchen Fall ist die Zurückgabe eines Proxy-Objekts für das eigentliche Bean-Objekt. Damit wird die in den folgenden Abschnitten behandelte aspektorientierte Programmierung in Spring realisiert. Dies ist der Grund, warum das Thema BeanPostProcessor in diesem Buch überhaupt behandelt wird.

Wie in Listing 14.6 zu sehen ist, gibt es zwei Methoden, die ein BeanPostProcessor implementiert und die beide nach der Erzeugung einer Spring-Komponente aufgerufen werden. Um zu verstehen, warum es zwei Methoden sind, muss erklärt werden, dass auf einem Bean-Objekt nach seiner Erzeugung und der vom Spring-Framework durchgeführten Dependency Injection auch noch Initialisierungsmethoden ausgeführt werden können (z. B. durch Angabe des Namens einer Methode als „init-method“ im Tag <bean>). Die Methode postProcessBeforeInitialization wird – wie der Name sagt – vor dem Aufruf einer solchen Initialisierungsmethode aufgerufen und postProcessAfterInitialization danach. Beim Aufruf beider Methoden wurde die Dependency Injection für die betreffende Spring-Komponente aber bereits durchgeführt. Da wir in diesem Buch keine Initialisierungsmethoden haben, werden bei uns die Methoden direkt hintereinander aufgerufen. Im Folgenden werden wir uns deshalb nur auf eine der beiden Methoden konzentrieren.

In unserer Anwendung verwenden wir die Komponenten Counter und Hello aus der ersten und zweiten Anwendung (Abschnitte 14.2 und 14.3) wieder. Wir fügen lediglich einen BeanPostProcessor hinzu. Durch unseren BeanPostProcessor soll demonstriert werden, dass zum einen ein Objekt durch einen BeanPostProcessor noch verändert werden kann und dass zum anderen der BeanPostProcessor ein ganz anderes Objekt sogar einer anderen Klasse zurückliefern kann. In unserem BeanPostProcessor wird geprüft, ob es sich bei dem übergebenen Bean-Objekt um ein Hello-Objekt handelt. In diesem Fall wird über setGreeting der in der XML-Konfigurationsdatei angegebene Wert verändert. Wenn der BeanPostProcessor feststellt, dass er ein Counter-Objekt erhalten hat, erzeugt er rein aus Demonstrationsgründen ein Objekt der Klasse DoubleCounter (s. Listing 14.7), die in Zweierschritten zählt.

Listing 14.7 Klasse DoubleCounter

```
package javacomp.spring.postprocessor;

import javacomp.spring.counter.Counter;

public class DoubleCounter extends Counter
{
    public void increment()
    {
        super.increment();
```

```
        super.increment();
    }
}
```

Unser Beispiel eines BeanPostProcessors wird durch die Klasse ExamplePostProcessor (s. Listing 14.8) realisiert.

Listing 14.8 Klasse ExamplePostProcessor als BeanPostProcessor-Beispiel

```
package javacomp.spring.postprocessor;

import javacomp.spring.counter.Counter;
import javacomp.spring.hello.Hello;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.*;

public class ExamplePostProcessor implements BeanPostProcessor
{
    public Object postProcessBeforeInitialization(Object bean,
                                                  String name)
        throws BeansException
    {
        return bean;
    }

    public Object postProcessAfterInitialization(Object bean,
                                                 String name)
        throws BeansException
    {
        System.out.println("nach Initialisierung von " + name);
        if(bean instanceof Hello)
        {
            Hello h = (Hello) bean;
            h.setGreeting("Hallo hallo");
        }
        else if(bean instanceof Counter)
        {
            return new DoubleCounter();
        }
        return bean;
    }
}
```

Um den BeanPostProcessor zu aktivieren, muss in der XML-Konfigurationsdatei der zweiten Spring-Anwendung (s. Abschnitt 14.3) lediglich diese eine Zeile ergänzt werden:

```
<bean class="javacomp.spring.postprocessor.ExamplePostProcessor"/>
```

Dieser Eintrag ist ein Beispiel dafür, dass eine Spring-Komponente nicht unbedingt einen Namen haben muss. Dies ist dann möglich, wenn sie in keiner anderen Komponente referenziert wird und wenn auch im Anwendungsprogramm über getBean nicht darauf zugegriffen wird. Wie oben erwähnt kann die Komponente wegen der Autodection des Spring-Frameworks dennoch ihre Rolle als BeanPostProcessor spielen.

Das Anwendungsprogramm kann unverändert vom zweiten Spring-Beispiel übernommen werden. Die Ausgabe zeigt den Effekt des BeanPostProcessors. Es wird sowohl der von ihm gesetzte Gruß ausgegeben als auch in Zweierschritten gezählt.

14.8 Aspektorientierte Programmierung (AOP)

Die aspektorientierte Programmierung (AOP) dient zur Realisierung sogenannter Querschnittsbelange (Cross Cutting Concerns). Darunter werden klassenübergreifende Aufgaben verstanden. Das Standard-Beispiel für AOP ist Logging. Damit ist gemeint, dass alle Methodenaufrufe einer Anwendung protokolliert werden sollen. Wenn man dies ohne AOP umsetzen möchte, bleibt einem nichts anderes übrig, als eine entsprechende Ausgabeanweisung in allen Methoden einzufügen. Damit verstößt man gegen das DRY-Prinzip (DRY: Don't Repeat Yourself). Mit AOP wird es möglich, die Ausgabeanweisung einmal zu programmieren und dann festzulegen, dass sie vor jedem Methodenaufruf ausgeführt wird, ohne dabei den restlichen Programmcode anfassen zu müssen.

Andere Beispiele für Querschnittsbelange sind

- die Transaktionssteuerung (zu Beginn vieler Methoden soll eine Transaktion gestartet und am Ende soll die Transaktion wieder beendet werden),
- die Zugriffskontrolle (bei jedem Methodenaufruf soll überprüft werden, ob der Aufrufer berechtigt ist, diese Methode zu nutzen)
- oder Caching (zu Beginn einer Methode soll geprüft werden, ob das gewünschte, eventuell aufwändig zu bestimmende Ergebnis früher schon einmal berechnet wurde; wenn ja, wird das vorher berechnete Ergebnis aus einem Cache gelesen und zurückgeliefert, andernfalls wird es berechnet, in den Cache übertragen und dann zurückgegeben).

AOP ist keine Technik, die es nur in Spring gibt. Sie kommt beispielsweise auch in AspectJ oder EJB vor. Wir behandeln aber AOP in diesem Buch im Zusammenhang mit Spring, da eine wesentliche Motivation für Spring war, Elemente aus EJB wie die deklarative Transaktionssteuerung oder die deklarative Zugriffskontrolle auch für andere Anwendungen verfügbar zu machen. Dies wird durch AOP möglich.

Bevor wir uns ein Beispiel zu AOP ansehen, sollen einige wichtige Begriffe der AOP eingeführt werden:

- Aspekt: Ein Aspekt ist eine „Querschnittsangelegenheit" (Cross Cutting Concern) wie z. B. das Logging.
- Advice (Ratschlag): Darunter wird der „einzuwebende" Code verstanden, also z. B. die Ausgabeanweisungen zur Realisierung des Logging.
- Join Points: Das sind mögliche (abstrakte) Punkte, an denen Code grundsätzlich „eingewoben" werden kann. Als Join Points kann man sich Methodenaufrufe, die Rückkehr von Methodenaufrufen, das Werfen von Ausnahmen, Zuweisungen, Zugriffe auf Felder, die Ausführung von if-Anweisungen usw. vorstellen. In Spring AOP kommen als Join Points nur Methodenaufrufe und die Rückkehr von Methodenaufrufen (auch durch Ausnahmen) in Frage.
- Pointcuts: Dies ist eine konkrete Menge von Punkten in einem konkreten Programm, an denen ein Advice „eingewebt" werden soll. Diese Punkte werden meistens durch eine Bedingung angegeben. Ein Beispiel für einen Pointcut wäre für ein konkretes Anwendungsprogramm die Menge aller Methodenaufrufe aus den Packages p. q. und r. s. t, wobei die aufgerufenen Methoden öffentlich sein müssen, drei Parameter haben müssen, von

denen der erste vom Typ int ist, sowie ihre Namen mit dem Buchstaben m beginnen müssen. Pointcuts werden häufig durch reguläre Ausdrücke über Package-, Klassen- und Methodennamen und deren Parameter definiert.

- Advisor: Ein Advisor ist ein Advice und ein dazugehöriger Pointcut, der festlegt, wo der Advice „einzuweben“ ist. Zu einem Programm kann es mehrere Advisors geben.

14.9 Fünfte Spring-Anwendung: AOP

Als AOP-Anwendung wollen wir das „klassische“ Logging realisieren. Dazu muss ein Advice programmiert werden, in dem sich die Ausgaben zur Protokollierung befinden. Ein Advice muss in Spring AOP immer eine bestimmte Schnittstelle implementieren. Im Folgenden verwenden wir die Schnittstelle MethodInterceptor, die eine einzige Methode namens invoke besitzt. Als Parameter von invoke wird ein Objekt des Typs MethodInvocation übergeben. Von diesem kann man sich über getMethod die aufgerufene Methode als Method-Objekt geben lassen, über getThis das „eigentliche“ Objekt, auf das der Methodenaufruf, der vom Advice abgefangen wurde, angewendet wurde, und über getArguments die Parameter des Methodenaufrufs. Ferner kann man durch Aufruf von proceed auf das MethodInvocation-Objekt die „eigentliche“ Methode auf das „eigentliche“ Objekt anwenden. Mit anderen Worten: Mit proceed kann man den Methodenaufruf durchführen, der vom Advice abgefangen wurde. Der Advice-Code ist nicht verpflichtet, proceed auszuführen. Denken Sie an das Caching-Beispiel! Wenn festgestellt wird, dass das Ergebnis schon einmal berechnet wurde und im Cache vorliegt, soll eben gerade in einem solchen Fall die Neuberechnung eingespart werden. Für das Logging-Beispiel wird aber proceed in allen Fällen durchgeführt, wie Listing 14.9 zeigt, das den Advice-Code enthält.

Listing 14.9 Advice-Klasse LoggingInterceptorAdvice

```
package javacomp.spring.aop;

import java.lang.reflect.*;
import org.aopalliance.intercept.*;

public class LoggingInterceptorAdvice implements MethodInterceptor
{
    public Object invoke(MethodInvocation invocation)
        throws Throwable
    {
        Object target = invocation.getThis();
        Method m = invocation.getMethod();
        Object[] args = invocation.getArguments();
        String message =
            "Aufruf von " + target.getClass().getName() +
            "." + m.getName() + "(";
        for(int i = 0; i < args.length; i++)
        {
            message += args[i];
            if(i < args.length - 1)
            {
```

```
                message += ", ";
            }
        }
        message += ")";
        System.out.println("vor " + message);
        Object result;
        try
        {
            result = invocation.proceed();
        }
        catch(Throwable t)
        {
            System.out.println("Ausnahme " + t + " in " + message);
            throw t;
        }
        if(m.getReturnType()==void.class)
        {
            System.out.println("nach " + message);
        }
        else
        {
            System.out.println("nach " + message + " => " + result);
        }
        return result;
    }
}
```

Auch wenn man proceed ausgeführt hat, ist man nicht verpflichtet, das Ergebnis der „eigentlichen“ Methode an den Aufrufer zurückzugeben. Man kann „unliebsame“ Ergebnisse auch austauschen. Ähnlich verhält es sich mit geworfenen Ausnahmen; man kann eine andere Ausnahme werfen oder die geworfene Ausnahme ganz unterdrücken. In dem Logging-Beispiel machen wir aber davon keinen Gebrauch. Es wird nach einer Protokollierung immer das von proceed errechnete Ergebnis zurückgegeben bzw. genau die geworfene Ausnahme erneut ausgelöst.

Als Anwendung können wir wieder die Hello- und Counter-Komponenten verwenden. In der XML-Konfigurationsdatei müssen wir zusätzlich die Advice-Komponente (s. Listing 14.9), den Pointcut und den Advisor bestehend aus Advice und Pointcut definieren. Sowohl für den Pointcut als auch für den Advisor benutzen wir vom Spring-Framework vorgegebene Klassen. Der Pointcut wird so parametrisiert, dass er alle Methodenaufrufe in Klassen, deren Package-Namen mit javacomp.spring beginnt, umfasst. Schließlich benötigen wir noch einen BeanPostProcessor, der beim Erzeugen von Objekten den Advice-Code mit Hilfe dynamischer Proxies „einwebt“. Die XML-Konfigurationsdatei sieht damit wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

<bean id="counter" class="javacomp.spring.counter.Counter">
</bean>

<bean id="hello" class="javacomp.spring.hello.Hello">
    <property name="greeting" value="Hallo"/>
    <property name="counter" ref="counter"/>
</bean>
```

```
<bean id="interceptorAdvice"
 class="javacomp.spring.aop.LoggingInterceptorAdvice"/>

<bean id="loggingPointcut"
 class="org.springframework.aop.aspectj.AspectJExpressionPointcut">
    <property name="expression"
     value="execution(* javacomp.spring.*.*.*(..))"/>
</bean>

<bean id="loggingAdvisor"
 class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="advice" ref="interceptorAdvice"/>
    <property name="pointcut" ref="loggingPointcut"/>
</bean>

<bean class="org.springframework.aop.framework.autoproxy.Default-
 AdvisorAutoProxyCreator"/>

</beans>
```

Der BeanPostProcessor erzeugt Proxies und liefert diese statt der „eigentlichen“ Objekte zurück. Eine Anwendung, die sich über getBean die Komponente mit dem Namen „hello“ beschafft, erhält somit das entsprechende Proxy-Objekt zurück. Aber auch bei der Dependency Injection wird statt des Objekts mit dem Namen „counter“ ein Proxy geliefert und als Attributwert im Hello-Objekt gespeichert. Es werden damit alle Aufrufe wie gewünscht protokolliert. Eine Anwendung, die sich über getBean die Spring-Komponente mit dem Namen „hello“ beschafft und darauf die Methode sayGreeting mit dem Parameter „Welt“ aufruft, erzeugt folgende Ausgabe:

```
vor Aufruf von javacomp.spring.hello.Hello.sayGreeting(Welt)
vor Aufruf von javacomp.spring.counter.Counter.increment()
nach Aufruf von javacomp.spring.counter.Counter.increment() => 1
vor Aufruf von javacomp.spring.counter.Counter.get()
nach Aufruf von javacomp.spring.counter.Counter.get() => 1
Hallo Welt (1)
nach Aufruf von javacomp.spring.hello.Hello.sayGreeting(Welt)
```

Eine Advice-Klasse muss nicht notwendig die Schnittstelle MethodInterceptor implementieren. Alternativ kommen auch die folgenden Schnittstellen in Frage:

- MethodBeforeAdvice: Die Methode before dieser Schnittstelle wird vor dem „eigentlichen“ Methodenaufruf ausgeführt. Danach wird ohne Zutun des Advice-Codes die „eigentliche“ Methode aufgerufen; es sei denn, die Methode before hat eine Ausnahme ausgelöst.
- AfterReturningAdvice: Die Methode afterReturning dieser Schnittstelle wird nach dem „eigentlichen“ Methodenaufruf ausgeführt; es sei denn, die „eigentliche“ Methode hat eine Ausnahme ausgelöst. Der Rückgabewert der „eigentlichen“ Methode kann in diesem Fall im Gegensatz zu MethodInterceptor nicht geändert werden, denn er wird ohne Zutun des Advice-Codes an den Aufrufer zurückgeliefert.
- ThrowsAdvice: Die Methoden dieses Advice werden ausgeführt, falls im „eigentlichen“ Methodenaufruf eine Ausnahme ausgelöst wurde.

In einer Anwendung können mehrere Advisors definiert werden. Dabei kann ein und derselbe Advice in mehreren Advisors vorkommen. Dasselbe gilt für einen Pointcut.

Advice-Komponenten sind im Normalfall Singletons, wenn – wie im Beispiel oben – kein Scope angegeben wird. Das bedeutet, dass dasselbe Advice-Objekt für unterschiedliche Zielobjekte eingesetzt wird. Es ist kein Problem, das richtige Zielobjekt zu finden, denn es wird im MethodInvocation-Parameter (getThis) mitgeliefert. Falls man für jedes Objekt ein eigenes Advice-Objekt haben möchte, so lässt sich dies leicht durch die Angabe Prototype als Scope in der Advice-Komponente erreichen. In der Regel ist es nicht notwendig. Bitte machen Sie sich klar, dass das Advice-Objekt nicht das Proxy-Objekt ist! Wie im ersten Teil des Buchs über Dynamic Proxies erklärt wurde, sind dynamische Proxies Objekte dynamisch erzeugter Klassen. Die Advice-Methoden werden von den Methoden dieser dynamisch erzeugten Proxy-Klassen aufgerufen. Ein Advice-Objekt stellt also einen Invocation-Handler dar. Wenn es also auch nur ein einziges Advice-Objekt gibt, hat jedes Objekt dennoch sein eigenes Proxy-Objekt.

Eine AOP-Anwendung lässt sich in Spring übrigens auch durch Annotationen konfigurieren. Ein Beispiel dazu finden Sie auf der Web-Seite zu diesem Buch.

14.10 Bewertung

Wie erwähnt ist die Vorstellung, was eine Komponente ist, bei Spring gleich wie bei Java Beans. Unter einer Spring-Komponente verstehen wir also je nach Kontext eine Bean-Klasse oder ein Bean-Objekt. Im Gegensatz zu Java Beans lässt sich bei Spring aber eindeutig ein Komponenten-Framework identifizieren, das wir in diesem Kapitel als ApplicationContext kennengelernt haben. Betrachten wir nun die Merkmale E1 bis E4 genauer:

- Zu E1: Eine Spring-Komponente ist eine Klasse, die in einem <bean>-Tag vorkommen kann. Dies gilt prinzipiell für jede Bean-Klasse, selbst wenn sie keinen parameterlosen Konstruktor hat, denn man kann in einem solchen Fall die benötigten Parameter im <bean>-Tag angeben. Insofern gibt es nur ein sehr schwach ausgeprägtes Komponentenmodell bei Spring, wobei man – wie schon bei Java Beans erläutert – es durchaus positiv sehen kann, dass es keine Vorgaben gibt.
- Zu E2: Die Kopplung der Java-Beans-Komponenten erfolgt durch das Komponenten-Framework von Spring aufgrund von XML-Konfigurationsdateien, die durch die Software-Entwickler vorgegeben werden. Dies ist der Kern von Spring und leistet den größten Beitrag zu Spring als Komponentensystem. Durch BeanPostProcessing kann die Kopplung anwendungsspezifisch oder für die Zwecke der AOP noch beeinflusst werden.
- Zu E3: Als „Einstiegsklassen“ kann man die Klassen sehen, die in den <bean>-Tags der Konfigurationsdateien vorkommen. Objekte dieser Klassen werden vom Spring-Framework erzeugt. Ein Lebenszyklus ist in schwach ausgeprägter Weise vorhanden. So kann man beispielsweise im <bean>-Tag unter „init-method“ und „destroy-method“ die Namen von Methoden einer Klasse angeben, die nach der Erzeugung bzw. vor dem Löschen von Spring-Komponenten automatisch vom Spring-Framework aufgerufen werden. Das Spring-Framework in Form des ApplicationContext bietet einige Funktionen für seine Komponenten an, auf die wir im Rahmen dieser Besprechung nicht eingegangen sind, die aber vorhanden sind.

- Zu E4: Was eine Komponente benötigt und bereitstellt, wird bei Spring nicht explizit angegeben.

Aufgrund der Ähnlichkeit bezüglich des Komponentenbegriffs ergibt sich für Spring eine ähnliche Bewertung wie für Java Beans bezüglich der Vorgaben für ein Komponentensystem aus Kapitel 7. Es ist durchaus möglich, so viele Übereinstimmungen zwischen den Vorgaben und den charakteristischen Merkmalen von Spring zu identifizieren, dass Spring als Komponentensystem bezeichnet werden darf. Durch die Existenz des Spring-Frameworks erscheint diese Entscheidung sogar noch mehr gerechtfertigt als für Java Beans.

15 Ereignisbusse

Das Komponenten-Framework Spring hilft zwar bei der „Verdrahtung“ der Objekte einer Anwendung durch Dependency Injection. Es ist aber dennoch ausschließlich eine Angelegenheit der Anwendungsentwickler, wie übersichtlich, wie verständlich und entsprechend änderungsfreundlich die Struktur der „verdrahteten“ Objekte ist. Spring hilft dabei nicht. Als Alternative zur „Verdrahtung“ von Objekten können Vernetzungsstrukturen verwendet werden, die auf der Hardware-Ebene seit langer Zeit als Bussysteme bekannt sind. Im Software-Bereich spricht man in der Regel von Ereignisbussen. Es gibt unterschiedliche Implementierungen von Ereignisbussen. In diesem Buch wird exemplarisch RRiBbit verwendet. Es ist ein sehr schlankes, übersichtliches und quelloffenes (Open Source) Framework. Zunächst beschäftigen wir uns aber in allgemeiner Form mit Ereignisbussen.

15.1 Grundkonzepte von Ereignisbussen

Mit Bussystemen können sowohl Hardware- als auch Software-Komponenten in sehr einfacher Weise verbunden werden (s. Bild 15.1). Eine Komponente muss lediglich an den Bus angeschlossen werden. Sie kann damit mit allen anderen Komponenten, die ebenfalls am Bus hängen, kommunizieren.

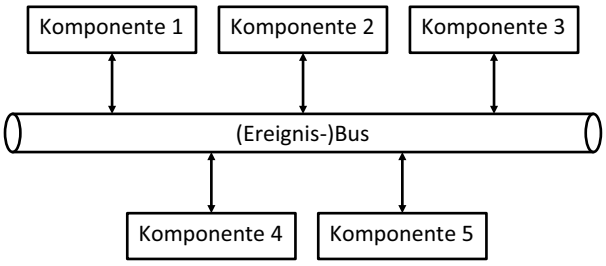

Bild 15.1 Prinzip eines Bussystems

Von großer Bedeutung bei einem Bussystem ist die Festlegung, welche Komponenten durch eine auf den Bus gelegte Nachricht angesprochen werden. Beim „klassischen“ Ethernet, bei dem ein Kabel durch ein Gebäude verlegt wurde, an das die Rechner angeschlossen wurden,

erfolgt dies durch Adressen, wobei es neben Unicast-Adressen, mit denen genau ein Rechner angesprochen wird, auch Sammeladressen in Form von Multicast- und Broadcast-Adressen zur Adressierung von Rechnergruppen bzw. zur Adressierung aller am Ethernet angeschlossenen Rechner gibt. Im Software-Bereich sind die Komponenten Objekte, die Methoden bereitstellen, welche auf die auf einem Bus gesendeten Ereignisse (daher der Name Ereignisbus) reagieren. Die Auswahl der passenden Methoden, die aufgerufen werden, erfolgt in der Regel anhand der Typen der im Ereignis übermittelten Parameter. Im einfachsten Fall dürfen die Ereignisbehandlungsmethoden nur einen einzigen Parameter haben, dessen Typ von einer vorgegebenen Ereignisbasisklasse abgeleitet sein muss. In dem im Folgenden behandelten Framework RRiBbit können die Ereignisbehandlungsmethoden beliebige Parameter haben. Beim Senden kann entsprechend eine beliebige Anzahl von Parametern beliebigen Typs angegeben werden. Die Auswahl der Methoden erfolgt dann anhand der Anzahl und des Typs der Parameter. Zusätzlich kann bei RRiBbit einer Methode durch eine Annotation noch ein String zugeordnet werden, der beim Senden (evtl. neben weiteren Parametern) angegeben werden kann, damit diese Methode durch das Sendeereignis aktiviert wird.

Hardware-Bussysteme stoßen bei einer sehr großen Anzahl von Komponenten an ihre Grenzen. Dies liegt vor allem an zwei Gründen: Zum einen erfolgt das Senden auf einem Bus sequenziell. Wenn also viele Komponenten gleichzeitig senden wollen, behindern diese sich gegenseitig. Zum anderen gibt es für das klassische Ethernet das Problem, dass in vielen Netzprotokollen mit Broadcast gearbeitet wird, wobei die Broadcast-Nachrichten nicht wirklich für alle sind, sondern ein Rechner erst durch die Analyse des Nachrichteninhalts herausfinden muss, ob die Nachricht für ihn ist oder nicht. Wenn ein Rechner direkt adressiert wird, erfolgt die Filterung, ob eine Nachricht angenommen werden muss, in Hardware. Die Analyse einer Broadcast-Nachricht erfolgt hingegen in Software. Wenn also immer mehr Rechner an einem Ethernet angeschlossen werden und somit immer mehr Broadcast-Nachrichten versendet werden, werden die Rechner immer stärker belastet mit dem Ausfiltern von Nachrichten, die gar nicht an sie gerichtet sind.

Für Software-Bussysteme hängt es stark von der Implementierung ab, ob die geschilderten Nachteile von Hardware-Bussystemen auch auf sie zutreffen. Das Senden auf dem Ereignisbus kann, muss aber nicht unter gegenseitigem Ausschluss erfolgen. RRiBbit erlaubt beispielsweise einen parallelen Zugriff auf den Ereignisbus. Auch für die Empfängerseite sind mehrere Formen der Implementierung denkbar. Eine Form zur Auswahl der passenden Methode ahmt das zuvor geschilderte Ethernet-Broadcast-Szenario nach: Jedes Objekt wird benachrichtigt und muss selbst überprüfen, ob es eine Methode für die aktuelle Nachricht hat. Damit handelt man sich dieselben Nachteile ein wie bei Ethernet: Es wird in der Regel vielfach Code ausgeführt, der nicht der Anwendung dient. In RRiBbit wird die Auswahl der durch ein Sendeereignis angesprochenen Objekte und Methoden wesentlich effizienter durchgeführt: Beim Anmelden von Komponenten am Ereignisbus extrahiert das RRiBbit-Framework mit Hilfe von Reflection Informationen und legt diese in Form sogenannter Listener-Objekte so ab, dass die passenden Objekte und Methoden schnell und ohne Zutun der betroffenen Objekte gefunden und die Ereignisbehandlungsmethoden aufgerufen werden können. Dennoch wird auch RRiBbit für sehr große Systeme an seine Grenzen stoßen. Zum einen muss immer noch nach den passenden Methoden gesucht werden. Auch wenn die Suche effizient ist, so ist ein direkter Methodenaufruf schneller. Zum anderen wird es bei sehr großen Systemen immer schwerer durchschaubar, welche Komponenten durch eine

gesendete Nachricht nun tatsächlich aktiv werden. Außerdem werden bei RRiBbit für jede Aktivierung Threads erzeugt, wie später noch näher erläutert wird. Die Thread-Erzeugung stellt einen deutlichen Mehraufwand dar.

Mit Software-Bussen lassen sich übrigens auch verteilte Anwendungen realisieren. Es gibt Busimplementierungen, die sich über mehrere Rechner erstrecken können. Alles, was auf einem Bus in einem Rechner gesendet wird, wird an die anderen Rechner weitergeleitet und dort auf den jeweiligen Bus gelegt. RRiBbit unterstützt ebenfalls verteilte Bussysteme.

Im Folgenden sehen wir uns RRiBbit etwas genauer an, wobei wir uns auf eine rein lokale Busbenutzung beschränken.

15.2 Komponentenmodell von RRiBbit

Auf den Web-Seiten zu RRiBbit wird angegeben, dass RRiBbit für „Request-Response-Bus“ steht. Dies erklärt die Großbuchstaben in RRiBbit. Ob die Kleinbuchstaben auch etwas zu bedeuten haben und gegebenenfalls was, entzieht sich meiner Kenntnis. Ich vermute, dass sie eingefügt wurden, um die Abkürzung besser aussprechen zu können. Der Name „Request-Response-Bus“ weist auf eine ganz entscheidende Eigenschaft des RRiBbit-Frameworks hin: Anders als bei einem Hardware-Bus geht es nicht um das Senden von Nachrichten, sondern die Leitidee von RRiBbit ist das Aufrufen von Methoden über den Bus. Das heißt, dass beim Senden nicht eine Nachricht verschickt und irgendwann später asynchron zum Sender die Methodenaufrufe durchgeführt werden, sondern bei RRiBbit handelt es sich um ein synchrones Senden, wobei der Aufruf der Sendemethode erst dann zurückkehrt, wenn alle dadurch angesprochenen Methoden abgearbeitet worden sind. Alle passenden Methoden werden übrigens parallel aufgerufen (jeder Aufruf wird in einem eigenen Thread ausgeführt). Wie oben schon erwähnt können während eines Sendevorgangs, der länger dauern kann, wenn die aufgerufenen Methoden länger brauchen, parallel weitere Sendeaufrufe von anderen Threads durchgeführt werden, u. a. auch von den Methoden, die durch das Senden aktiv geworden sind. Der Rückgabewert, den eine durch das Senden aufgerufene Methode zurückgibt, kann vom Sendeaufruf an den Aufrufer zurückgeliefert werden.

Eine Komponente ist bei RRiBbit, wie bei Java Beans und bei Spring, ein Objekt bzw. eine Klasse. Beides kann dem Framework als Komponente übergeben werden. Wenn man Objekte übergibt, ist es möglich, mehrere Objekte derselben Klasse zu übergeben. Damit kann es also zu einer Klasse mehrere Komponentenobjekte geben. Wenn eine Klasse statt eines Objekts an das RRiBbit-Framework übergeben wird, dann erzeugt das Framework ein Objekt dazu. Dies ist allerdings nur dann möglich, falls die Klasse einen parameterlosen Konstruktor besitzt. Die Klasse einer Komponente muss ansonsten keine weiteren Bedingungen erfüllen. Eine Methode, die über den Request-Response-Bus aufgerufen werden kann, muss lediglich mit der Annotation @Listener versehen werden, wobei diese Annotation optional den String-Parameter hint haben kann.

Für das Senden einer Nachricht über den Ereignisbus gibt es mehrere Varianten. In allen Fällen kann man eine beliebige Anzahl von Parametern beliebigen Typs angeben (dies wird möglich durch Varargs-Parameter des Typs Object). Eine mit @Listener annotierte Methode

wird nur dann aufgerufen, falls die übergebenen Argumente in der Sendemethode typkompatibel sind mit den Parametern der Methode, wobei die Reihenfolge der Parameter eine Rolle spielt. Bei den unterschiedlichen Varianten des Sendens gibt es folgende Unterschiede:

- Es gibt Sendemethoden, bei denen man außer den Parametern zusätzlich einen String (hint) angeben kann. Eine @Listener-Methode wird nur dann aufgerufen, wenn sie diesen String in der @Listener-Annotation als Hint angegeben hat.
- Neben den Parametern und einem eventuellen Hint kann in Form eines Class-Objekts der Rückgabetyp angegeben werden. Eine Methode wird nur dann aufgerufen, wenn sie diesen Rückgabetyp besitzt.
- Beim Senden einer Nachricht über den Bus werden bekanntlich alle passenden Methoden aufgerufen. Der Sendeaufruf kehrt erst zurück, wenn alle aufgerufenen Methoden ausgeführt wurden. Es gibt Sendemethoden, die void sind und damit Rückgabewerte der aufgerufenen Methoden ignorieren. Es gibt aber auch solche Sendemethoden, die genau einen Wert (irgendeinen) Wert zurückliefern, und solche, die alle Rückgabewerte in einer Collection zurückgeben.

Es gibt nicht für alle theoretisch möglichen Kombinationen Sendemethoden (mit/ohne Hint, mit/ohne Angabe eines Rückgabetyps usw.). RRiBbit bietet über neun Methoden acht unterschiedliche Sendevarianten an (eine Variante steht unter zwei unterschiedlichen Namen zur Verfügung, darunter ein Mal unter dem einfachen Namen „send“).

Will man in den Methoden einer Klasse den Request-Response-Bus benutzen, so besitzen diese in der Regel entweder ein Attribut auf den Bus, das im Konstruktor gesetzt wird, oder sie beschaffen sich die Referenz selbst über die statische Methode get der Klasse RRB.

Eine RRiBbit-Anwendung besteht neben Hilfsklassen somit aus Klassen, in denen Nachrichten auf den RRiBbit-Bus gesendet werden und Klassen, die mit @Listener annotierte Methoden besitzen, um auf das Senden von Nachrichten zu reagieren (wobei natürlich auch in Klassen mit @Listener-Methoden etwas auf den Bus gesendet werden darf).

Im Hauptprogramm wird zuerst ein ListenerObjectCreator erzeugt. Wie der Name sagt erzeugt dieser die Listener-Objekte. Ein Listener-Objekt ist keines, dessen Klasse die @Listener-Methoden enthält, sondern ein Listener-Objekt kommt vom Framework und repräsentiert die @Listener-Methode eines Objekts. Es enthält u. a. die annotierte Methode in Form eines Method-Objekts, den Hint und die Referenz auf das Objekt der Klasse mit den @Listener-Methoden. Durch die Listener-Objekte wird eine schnelle Auswahl und ein schneller Aufruf der betroffenen Methoden realisiert. Es gibt mehrere ListenerObjectCreator-Varianten. Man kann entweder Objekte angeben oder Klassen (wobei das Framework zu jeder Klasse dann ein Objekt erzeugt) oder Packages, die nach Klassen mit @Listener-Annotationen durchsucht werden. Nachdem man im ersten Schritt einen ListenerObjectCreator erzeugt hat, muss man im zweiten Schritt einen RequestResponseBus erzeugen, wobei der ListenerObjectCreator als Parameter übergeben wird. Schließlich muss nur noch die Anwendung in Gang gesetzt werden, was eventuell dadurch erfolgt, dass eine erste Nachricht auf den Bus gesendet wird.

15.3 Erste RRiBbit-Anwendung

In unserer ersten Anwendung benutzen wir eine Zählerklasse, deren Methode increment durch die Annotation @Listener als Listener-Methode gekennzeichnet ist. Vorsorglich machen wir diese Methode synchronized. Außerdem verlängern wir zu Demonstrationszwecken die Ausführung dieser Methode um fünf Sekunden durch einen Aufruf von sleep und fügen Ausgabeanweisungen zu Beginn und am Ende der Methode ein (s. Listing 15.1).

Listing 15.1 Counter-Klasse mit @Listener-Annotation

```
package javacomp.eventbus.counter;

import org.rribbit.Listener;

public class Counter
{
    private int counter;

    @Listener(hint="increment")
    public synchronized int increment()
    {
        System.out.println("   Beginn Counter.increment");
        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException e)
        {
        }
        counter++;
        System.out.println("   Ende Counter.increment: " +
                           "counter = " + counter);
        return counter;
    }
}
```

Ferner existiert ein zweite Klasse DoubleCounter, in der die Zeile

```
counter++;
```

durch die Zeile

```
counter += 2;
```

ersetzt wird. Diese Klasse, die also in Zweierschritten zählt, hat aber genau dieselbe @Listener-Annotation (auch mit demselben Hint-String).

Das Hauptprogramm unserer ersten Ereignisbusanwendung findet sich in Listing 15.2.

Listing 15.2 Hauptprogramm der ersten Ereignisbusanwendung (Variante 1)

```
package javacomp.eventbus.counter;

import org.rribbit.*;
import org.rribbit.creation.*;
```

```
public class CounterApp
{
    public static void main(String[] args) throws Exception
    {
        ListenerObjectCreator creator =
            new InstantiatingClassBasedListenerObjectCreator(
                                Counter.class, DoubleCounter.class);
        RequestResponseBus bus =
            RRiBbitUtil.createRequestResponseBusForLocalUse(creator,
                                                            true);
        for(int i = 0; i < 10; i++)
        {
            int value = bus.send("increment");
            System.out.println("" + value);
        }
    }
}
```

Wie zuvor beschrieben, wird unter der Angabe der Klassen Counter und DoubleCounter ein ListenerObjectCreator erzeugt. Dieser wird einer statischen Methode zur Erzeugung eines RRiBbit-Busses übergeben. Der Name der Methode createRequestResponseBusForLocalUse zeigt, dass es sich hier um keine verteilte Anwendung handelt. Der zweite Parameter true gibt an, dass die Referenz auf den erzeugten Bus in dem statischen Attribut der Klasse RRB abgelegt wird, so dass man auf den Bus über die statische Methode get der Klasse RRB Zugriff erhält, was wir aber in dieser ersten Anwendung nicht brauchen (in der zweiten Anwendung aber schon). Im Folgenden wird dann zehn Mal die Methode send aufgerufen. Dies ist eine der angebotenen Varianten von Sendemethoden, in der ein Hint, aber kein Rückgabetyp angegeben wird und die ein einziges Ergebnis zurückliefert. Weitere Parameter wären als Varargs möglich, werden aber hier nicht übergeben, da die Zielmethoden increment parameterlos sind. Die Methode send ist übrigens eine generische Methode, wobei der Rückgabetyp der Typparameter ist. Im vorliegenden Fall setzt der Compiler als Typparameter Integer ein. Durch Auto-Unboxing kann dann der Rückgabetyp einer Variablen des Typs int ohne Casting zugewiesen werden. Eine Zuweisung an jeden anderen Typ wäre ebenfalls ohne Casting möglich.

An der Ausgabe der beiden Methoden increment erkennen wir, dass beide parallel ausgeführt werden (zwei Mal die Beginn-Meldung hintereinander und nach nur ca. 5, nicht nach ca. 10 Sekunden zwei Mal die Endemeldung). Die Ausgabe des Hauptprogramms kann zum Beispiel so aussehen:

```
1
4
6
4
5
...
```

Wie zu sehen ist, kann nicht vorhergesagt werden, von welchem der beiden Increment-Aufrufe das von send zurückgelieferte Ergebnis kommt. Im ersten, vierten und fünften Aufruf ist es der einfache Zähler (1, 4 und 5), im zweiten und dritten Aufruf dagegen der Doppelzähler (4 und 6).

Ein ListenerObjectCreator lässt sich auch aus Objekten erzeugen. Im folgenden Programm (s. Listing 15.3) werden zwei Counter- und drei DoubleCounter-Objekte von der Anwendung angelegt und dem Konstruktor zur Erzeugung eines ListenerObjectCreators übergeben (es wird dabei eine andere Klasse als zuvor verwendet). Der Rest des Programms ist exakt identisch mit dem vorigen.

Listing 15.3 Hauptprogramm der ersten Ereignisbusanwendung (Variante 2)

```
package javacomp.eventbus.counter;

import org.rribbit.*;
import org.rribbit.creation.*;

public class CounterApp
{
    public static void main(String[] args) throws Exception
    {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        DoubleCounter c3 = new DoubleCounter();
        DoubleCounter c4 = new DoubleCounter();
        DoubleCounter c5 = new DoubleCounter();
        ListenerObjectCreator creator =
            new ObjectBasedListenerObjectCreator(c1, c2,
                                                 c3, c4, c5);
        RequestResponseBus bus =
            RRiBbitUtil.createRequestResponseBusForLocalUse(creator,
                                                            true);
        for(int i = 0; i < 10; i++)
        {
            int value = bus.send("increment");
            System.out.println("" + value);
        }
    }
}
```

Führt man das Programm aus, kann man sehen, dass jetzt bei jedem Senden fünf Methoden aufgerufen werden. Auch diese Aufrufe erfolgen alle parallel und dauern zusammen nur etwa 5 Sekunden. Bitte machen Sie sich dazu klar, dass die Increment-Methoden zwar synchronized sind, dass aber synchronized seine Wirkung in diesem Programm noch gar nicht entfaltet, weil durch synchronized die gleichzeitige Ausführung der Increment-Methoden auf demselben Objekt verhindert wird. Hier haben wir aber unterschiedliche Objekte, so dass die Parallelität an keiner Stelle eingeschränkt wird. Für die Ausgabe des Hauptprogramms gibt es prinzipiell dieselben Möglichkeiten wie zuvor: bei der i-ten Ausgabe wird entweder der Wert i oder der Wert 2*i ausgegeben. In diesem Fall ist der Wert 2*i vielleicht etwas wahrscheinlicher, da wir drei DoubleCounter-Objekte, aber nur zwei einfache Counter-Objekte nutzen.

Durch die Verwendung einer anderen Methode zum Senden kann man sich auch alle Rückgabewerte geben lassen. Dies erreicht man durch Ersetzung der For-Schleife in Listing 15.3 durch die folgende Schleife:

```
for(int i = 0; i < 10; i++)
{
```

```
    Collection<Integer> values =
                        bus.sendForMultipleWithHint("increment");
    System.out.println("" + values);
}
```

Nun erhält man die Rückgabewerte aller fünf aufgerufenen Methoden in unterschiedlicher Reihenfolge:

```
[1, 1, 2, 2, 2]
[2, 4, 4, 2, 4]
[6, 6, 3, 6, 3]
[4, 8, 8, 4, 8]
...
[10, 20, 20, 10, 20]
```

15.4 Zweite RRiBbit-Anwendung

Die zweite RRiBbit demonstriert zum einen, wie eine Komponente eine andere Komponente benutzt (das wurde für alle anderen Komponenten-Frameworks ebenfalls gezeigt), und zum anderen unterschiedliche Möglichkeiten zur Auswahl von @Listener-Methoden.

Als neue Komponentenklasse benutzen wir die Klasse SomeClass aus Listing 15.4. Diese Klasse hat mehrere Methoden. Alle Methodenaufrufe sollen von einem einzigen Zähler gezählt werden, wobei dafür der Counter aus Listing 15.1 verwendet werden soll. Wir gehen davon aus, dass im Hauptprogramm die beiden Klassen SomeClass und Counter zur Erzeugung des ListenerObjectCreators übergeben werden, so dass es also zu jeder Klasse genau ein Objekt geben wird.

Listing 15.4 Klasse SomeClass mit @Listener-Annotationen

```
package javacomp.eventbus.match;

import org.rribbit.Listener;
import org.rribbit.RRB;

public class SomeClass
{
    private int increment()
    {
        return RRB.get().send("increment");
    }

    @Listener
    public void m(String s)
    {
        System.out.println("m(String) - " + increment());
    }

    @Listener
    public void m(int i, String s)
    {
```

```
        System.out.println("m(int,String) - " + increment());
    }

    @Listener
    public void m(String s, int i)
    {
        System.out.println("m(String,int) - " + increment());
    }

    @Listener(hint="m1")
    public void m1(int i)
    {
        System.out.println("m1(int) - Beginn");
        System.out.println("m1(int) - Ende - " + increment());
    }

    @Listener(hint="m2")
    public int m2(int i)
    {
        System.out.println("m2(int) - Beginn");
        System.out.println("m2(int) - Ende - " + increment());
        return 0;
    }

    @Listener(hint="m3")
    public void m3()
    {
        System.out.println("m3() - " + increment());
    }
}
```

Die Klasse SomeClass hat die private Methode increment, die von allen anderen Methoden verwendet wird. Die Methode increment beschafft sich über die statische Methode get der Klasse RRB eine Referenz auf den Request-Response-Bus und wendet darauf die Methode send in derselben Weise wie das Hauptprogramm aus Listing 15.2 bzw. Listing 15.3 an. Da es nur ein Objekt geben wird, das auf den Hint „increment“ reagieren wird, gibt es nur einen einzigen Rückgabewert, der damit eindeutig ist.

Alle anderen Methoden sind Listener-Methoden. Diese Methoden können also durch eine Sendemethode aufgerufen werden. Die Sendemethode kehrt erst zurück, wenn alle Methoden abgearbeitet wurden. In diesem Zeitraum wird nun eine weitere Sendemethode in den Listener-Methoden aufgerufen, was kein Problem darstellt. Daran erkennt man, dass der Request-Response-Bus „reentrant“ ist, also während einer Benutzung eine weitere Benutzung erlaubt. Somit ist also in einfacher Weise gezeigt, dass eine Komponente (Counter) durch eine andere Komponente (SomeClass) problemlos genutzt werden kann. Im Folgenden betrachten wir nun unterschiedliche Varianten des Sendens zur Auswahl von Listener-Methoden.

Die ersten drei öffentlichen Methoden der Klasse SomeClass unterscheiden sich durch ihre Parameter, wobei aber jede Parameterkombination eindeutig ist. Es ist deshalb möglich, jede Methode eindeutig durch Angabe von Argumenten des entsprechenden Typs aufzurufen, wobei es auf die Reihenfolge der Parameter ankommt (die Variable bus zeigt dabei wie im Hauptprogramm von Listing 15.2 bzw. Listing 15.3 auf den Request-Response-Bus):

```
bus.sendForNothing("hallo");
bus.sendForNothing(1, "hallo");
bus.sendForNothing("hallo", 2);
```

Wie an der Ausgabe des Programms abgelesen werden kann, wird durch den ersten Aufruf die erste Variante der Methode m, durch den zweiten Aufruf die zweite und durch den dritten Aufruf die dritte Variante angesprochen.

Bei dem folgenden Aufruf ändert sich die Situation jedoch:

```
bus.sendForNothing(1);
```

Zu dem Argument des Typs int gibt es zwei passende Methoden, nämlich m1 und m2. Die Ausgabe des Programms zeigt, dass auch in diesem Fall die beiden Methoden m1 und m2 parallel ausgeführt werden. Der Unterschied zur ersten Anwendung ist jetzt aber der, dass beide Methoden auf dasselbe Objekt angewendet werden. Für unsere Beispielklasse ist das kein Problem, da es nichts zu synchronisieren gibt. Würde man die beiden Methoden m1 und m2 zu Probezwecken aber trotzdem synchronized machen, würde man einen Unterschied bei der Ausgabe sehen (ohne synchronized: „Beginn Beginn Ende Ende“, mit synchronized „Beginn Ende Beginn Ende“). Da in den beiden parallel ausgeführten Methodenaufrufen der Counter aufgerufen wird, wird somit also auch das Counter-Objekt parallel genutzt. Für genau diesen Fall wurde die Methode increment in Listing 15.1 schon synchronized gesetzt. Übrigens konnte ich bei mehreren Ausführungsversuchen ohne synchronized tatsächlich einmal sehen, dass die beiden parallel ausgeführten Aufrufe denselben Wert zurückgeliefert haben, was nicht vorkommen sollte. Die Synchronisation von increment ist in diesem Fall also zwingend notwendig.

Wenn man über den Request-Response-Bus nicht beide Methoden m1 und m2, die denselben Parametersatz haben, aufrufen möchte, gibt es in diesem Fall die Möglichkeit, über den Rückgabetyp, der bei beiden Methoden unterschiedlich ist, zu differenzieren. Durch den folgenden Aufruf wird nur m1 aufgerufen, da m1 den Rückgabetyp void hat:

```
bus.sendForSingleOfClass(void.class, 2);
```

Eine andere Form der Differenzierung kann in diesem Fall über den Hint-String erfolgen, der ebenfalls für beide Methoden unterschiedlich ist. Nur die Methode m2 kann zum Beispiel so aktiviert werden:

```
bus.sendForSingleWithHint("m2", 3);
```

Diese Variante des Sendens sendForSingleWithHint ist auch einfacher über die Methode send verfügbar, die in der ersten Anwendung benutzt wurde:

```
bus.send("m2", 3);
```

Man kann auch sowohl den Hint-String als auch den Rückgabetyp angeben. Auch auf diese Weise wird nur m1 aufgerufen:

```
bus.sendForSingleOfClassWithHint(void.class, "m1", 4);
```

Wenn beim Senden etwas spezifiziert wird, wofür keine passende Methode gefunden wird, so hat der Sendeaufruf keine Wirkung. Dies ist beim folgenden Aufruf der Fall:

```
bus.sendForSingleOfClassWithHint(int.class, "m1", 4);
```

Schließlich soll noch die parameterlose Methode m3 aufgerufen werden und anschließend nochmals die Methode m mit String-Parameter:

```
bus.sendForNothing();
bus.sendForNothing("servus");
```

Überraschenderweise springt der Wert des Zählers von der ersten zur zweiten Ausgabe nicht um 1 wie bei allen vorigen Ausgaben, sondern um 2. Der Grund ist darin zu sehen, dass durch den ersten Aufruf alle parameterlosen Methoden aktiviert werden. Außer der Methode m3 der Klasse SomeClass ist dies auch noch die Methode increment der Klasse Counter. In m3 wird dann increment nochmals aufgerufen. Dies erklärt, warum im folgenden Aufruf der Wert des Zählers um 2 gestiegen ist. Dieses Beispiel sollte eine Warnung sein: Auch wenn die Aktivierung von Methoden über den Request-Response-Bus bequem ist, kann es doch passieren, dass ungewollt zusätzliche Methoden auf Objekten aufgerufen werden. Man sollte also darauf achten, dass die Methoden, die angesprochen werden sollen, möglichst genau spezifiziert werden, was allerdings den Programmieraufwand wieder etwas erhöht.

15.5 Bewertung

RRiBbit hat bezüglich der Bewertung als Komponenten-Framework viele Ähnlichkeiten mit Spring. Wie bei Spring ist eine RRiBbit-Komponente eine Klasse oder ein Objekt; beides kann ja zur Erzeugung eines ListenerObjectCreators angegeben werden. Die von Spring durchgeführte Vernetzung der Komponenten durch Dependency Injection wird bei RRiBbit durch das Vorhandenseins und die Nutzung des Request-Response-Bus durch die Komponenten ersetzt. Der Bus bildet zusammen mit dem ListenerObjectCreator das Komponenten-Framework. Wir betrachten die Merkmale E1 bis E4 im Einzelnen:

- Zu E1: Eine RRiBbit-Komponente ist eine Klasse oder ein Objekt einer Klasse, die mit @Listener annotierte Methoden besitzt und in der der Request-Response-Bus benutzt wird. Zu Beginn werden Objekte oder Klassen als Komponenten bei der Erzeugung des ListenerObjectCreators angegeben.
- Zu E2: Die Kopplung der RRiBbit-Komponenten erfolgt durch den Request-Response-Bus, welcher den Kern des Frameworks darstellt.
- Zu E3: Als „Einstiegsklassen“ kann man die Klassen sehen, die bei der Erzeugung des ListenerObjectCreators angegeben werden. Das Framework erzeugt dann die entsprechenden Objekte. Alternativ können auch Objekte von der Anwendung erzeugt und an den Konstruktor eines ListenerObjectCreators übergeben werden. Ein Lebenszyklus wird nicht implementiert. Das RRiBbit-Framework bietet außer dem Request-Response-Bus wenige weitere Funktionen an.
- Zu E4: Bei RRiBbit wird nicht explizit spezifiziert, was eine Komponente benötigt und bereitstellt.

RRiBbit erfüllt die charakteristischen Merkmale eines Komponenten-Frameworks gerade noch. Man könnte aber RRiBbit so erweitern, dass es dem Ideal eines Komponenten-Frame-

works deutlich näher kommt. Wie bei anderen Frameworks kann man sich vorstellen, dass RRiBbit-Komponenten in Jar-Dateien gepackt werden, die dynamisch zur Laufzeit installiert, deinstalliert und neu installiert werden. Bei der Installation untersucht das Framework alle Klassen und erzeugt Listener-Objekte zu den Klassen, die mit @Listener annotierte Methoden haben. Diese werden damit sozusagen am Bus angeschlossen. Genauso werden bei einer Deinstallation diese Objekte wieder vom Bus getrennt. Auch unterschiedliche ClassLoader könnten für die unterschiedlichen Komponenten verwendet werden. So wäre es ein aus meiner Sicht sehr schönes Komponenten-Framework. Vielleicht wird diese Erweiterung eines Tages tatsächlich durchgeführt.

16 Android

Android ist eine Software-Plattform für Smartphones und Tablets. Die wesentlichen Bestandteile von Android sind ein Linux-Betriebssystemkern sowie ein Framework zur Ausführung von Android-Anwendungen. Die Nutzung dieses Android Application Frameworks, die bei der Programmierung von Android-Anwendungen notwendig ist, erfordert eine gewisse Einarbeitungszeit. Das Android-Framework unterscheidet sich von den anderen in diesem Buch behandelten Frameworks zum Teil deutlich. Das Bemerkenswerteste an Android im Kontext dieses Buches über Java-Komponenten ist die Möglichkeit, dass eine Anwendung relativ leicht Teile einer anderen Anwendung (inklusive Teile der grafischen Benutzeroberfläche einer anderen Anwendung) benutzen kann. Eine weitere Besonderheit von Android ist, dass Linux-Prozesse und Android-Anwendungen unabhängiger voneinander sind als in anderen Umgebungen. So kann eine Anwendung auf mehrere Prozesse aufgeteilt werden oder umgekehrt kann ein Teil einer anderen Anwendung im selben Prozess laufen wie die nutzende Anwendung. Auch kann der Prozess einer Anwendung beendet werden, wenn der Anwender darauf im Moment nicht zugreift. Wenn er dann zu der Anwendung zurückkehrt, wird ein neuer Prozess gestartet und die Anwendung an der Stelle fortgesetzt, wo sie zuletzt war, so dass der Benutzer in der Regel nicht bemerkt, dass seine Anwendung jetzt von einem anderen Prozess ausgeführt wird. Weiterhin bietet Android neben einem Teil der „normalen“ Java-Bibliothek eine umfangreiche Bibliothek an, über die Android-Anwendungen zum Beispiel auf die Hardware zur Positionsbestimmung des Geräts zugreifen können.

Die Verkaufszahlen von Smartphones und Tablets (mit Android, aber auch mit iOS von Apple) sind in den letzten Jahren explodiert. Parallel dazu steigt die Zahl der verfügbaren Apps (Anwendungen für diese Geräte) sehr schnell an. Auch bei den Informatik-Studierenden ist ein sehr starkes Interesse an der Programmierung von Smartphones und Tablets erkennbar, was neben dem Modetrend vermutlich daran liegt, dass die Studierenden diese Geräte immer bei sich haben, häufig am Tag benutzen und somit eng mit ihnen verwachsen sind.

Bevor wir auf das Komponentenmodell von Android zu sprechen kommen, werden einige grundlegenden Prinzipien von Android erläutert. Das Apple-System iOS wird in diesem Buch übrigens nicht behandelt, da es im Gegensat zu Android nicht auf Java basiert.

16.1 Software-Architektur von Android

Die Basis der Android-Software-Architektur (s. Bild 16.1) bildet ein Linux-Betriebssystemkern, der für die Realisierung von Prozessen und Threads, die Speicherverwaltung, die Realisierung des Dateisystems sowie den Zugriff auf die Hardware durch unterschiedliche Treiber zuständig ist. Auf dem Linux-Kern sitzen einige in C bzw. C++ programmierte Bibliotheken. Dazu gehört beispielweise eine Grafikbibliothek, eine Medienbibliothek zum Codieren und Decodieren der wichtigsten Audio- und Videoformate sowie eine Datenbank. Das Android Application Framework ist in Java geschrieben. Es bildet die Basis für alle Android-Anwendungen. Zu den Android-Anwendungen gehören solche, die beim Erwerb eines Android-Geräts bereits vorhanden sind wie zum Beispiel die Home-Anwendung (Start-Bildschirm), ein Browser oder bei Smartphones die Anwendung zum Telefonieren. Android-Anwendungen sind aber auch aus dem Internet heruntergeladene oder selbst entwickelte Apps.

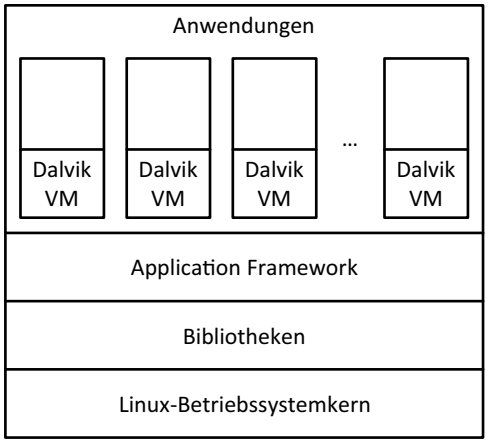

Bild 16.1 Software-Architektur von Android

Alle Anwendungen führen Java-Code aus, der wie üblich von einer Virtual Machine (VM) interpretiert wird. In Android wird aus Effizienzgründen allerdings eine ganz spezielle VM verwendet, die sogenannte Dalvik VM. Der Code, der von einem üblichen Java-Compiler generiert und in Dateien mit der Endung .class geschrieben wird, ist geeignet für einen virtuellen Prozessor, der durch eine „normale“ Java VM simuliert wird. Dieser virtuelle Prozessor arbeitet als Kellermaschine (das heißt, alle Operanden der Maschinenbefehle werden einem Kellerspeicher entnommen). Für Android wird jedoch eine Registermaschine als virtueller Prozessor eingesetzt (das heißt, die Operanden der Maschinenbefehle stehen in Registern, die in den Maschinenbefehlen explizit angegeben werden). Der von einem üblichen Java-Compiler erzeugte Code kann von dem virtuellen Prozessor, der durch die Dalvik VM realisiert wird, nicht ausgeführt werden. Deshalb müssen die Class-Dateien in DEX-Dateien (Dalvik Executable) transformiert werden, damit sie von einer Dalvik VM interpretiert werden können (dabei wird aus mehreren Class-Dateien eine einzige DEX-Datei). In der Regel läuft jede Android-Anwendung in einem eigenen Linux-Prozess, in dem der DEX-Code dieser Anwendung von einer Dalvik VM ausgeführt wird. Es gibt aber durchaus auch die Möglichkeit, dass eine Anwendung auf mehrere Prozesse verteilt ist.

16.2 Prinzipien der Ausführung von Apps

Um das Programmiermodell von Android-Anwendungen zu verstehen, sollte man die grundsätzlichen Abläufe bei der Ausführung solcher Anwendungen kennen:

- Die für eine Nutzerin sichtbaren Bestandteile einer App in Android sind Activities. Jede Activity entspricht dem Inhalt einer Bildschirmseite sowie dem Programmcode, der auf die Eingaben der Nutzerin reagiert. Als Beispiel kann man sich eine Anwendung zum Telefonieren vorstellen. Diese besteht aus einer Activity zum Wählen, einer Activity während eines Telefonats, einer Activity zur Anzeige der gespeicherten Kontakte und einer Activity zum Anzeigen der letzten Gespräche. Eine E-Mail-Anwendung hat eine Activity zum Anzeigen der eingegangenen E-Mails, eine Activity zum Lesen einer ausgewählten E-Mail und eine Activity zum Schreiben und Abschicken einer neuen E-Mail.
- In erster Näherung gehen wir davon aus, dass es eine 1:1-Beziehung zwischen einer laufenden Anwendung und einem Prozess gibt. Das heißt, dass alle Activities, die zu derselben Anwendung gehören, durch denselben Prozess ausgeführt werden und Activities unterschiedlicher Anwendungen durch unterschiedliche Prozesse. Später wird diese Annahme relativiert. Aber diese vereinfachte Vorstellung einer 1:1-Beziehung hilft erst einmal weiter.
- Eine Activity startet die auf sie folgende Activity, wobei die folgende Activity zu derselben Anwendung oder zu einer anderen Anwendung gehören kann. Wenn sie zur selben Anwendung gehört, läuft die folgende Activity (in der Regel) im selben Prozess. Wenn die Folge-Activity zu einer anderen Anwendung gehört, dann läuft sie im Prozess, der zu dieser anderen Anwendung gehört. Existiert für die andere Anwendung bereits ein Prozess, wird dieser verwendet. Andernfalls wird ein neuer Prozess erzeugt.
- Da eine Activity eine Activity einer anderen Anwendung starten kann, kann man in Android bequem Teile einer anderen Anwendung in seiner eigenen Anwendung nutzen. Beispielsweise kann man eine Anwendung schreiben, aus der heraus jemand per Telefon angerufen werden soll. Man muss dazu einfach die entsprechende Activity der Telefonanwendung mit der anzurufenden Telefonnummer aktivieren.
- Die Activities bilden einen Keller. Jede neue Activity kommt als oberstes Element auf den Keller. Wie in einem Browser gibt es in Android eine Zurück-Schaltfläche. Wenn man diese drückt, wird das oberste Element vom Keller entfernt und es wird die Activity angezeigt, die jetzt als oberstes Element auf dem Keller liegt. In Wirklichkeit gibt es auch zu diesem Prinzip einige Ausnahmen. Insbesondere gibt es mehrere Keller, auf die man beim Hin- und Herspringen zwischen unterschiedlichen Anwendungen umschaltet. Aber für die folgenden Äußerungen genügt die Vorstellung eines einzigen Activity-Kellers.

Zum besseren Verständnis der geschilderten Prinzipien betrachten wir einen beispielhaften Ablauf. Wir gehen von der Situation in Bild 16.2 aus. In dieser und den folgenden Abbildungen sind die Activities auf der y-Achse und die dazugehörigen Prozesse auf der x-Achse angeordnet. In Bild 16.2 sehen Sie ganz links den Prozess der Home-Anwendung mit einer entsprechenden Activity. Von der Home-Anwendung wurde durch Anklicken eines Symbols (eines Icons) auf dem Startbildschirm eine E-Mail-Anwendung gestartet. Dafür gibt es einen eigenen Prozess, der in Bild 16.2 rechts neben dem Prozess der Home-Anwendung dargestellt ist. Wir nehmen an, dass die erste Activity der E-Mail-Anwendung alle eingegangenen E-Mails anzeigt. Weiter gehen wir davon aus, dass die Nutzerin dann eine der eingegange-

nen E-Mails zum Lesen angeklickt hat. Darauf wurde eine neue Activity im selben Prozess gestartet, die die selektierte Mail zum Lesen anzeigt. Das unterste Element auf dem Activity-Keller ist also die Home-Activity, dann folgt die Posteingang-Activity und das oberste Element ist die gerade angezeigte Activity zum Lesen der ausgewählten E-Mail.

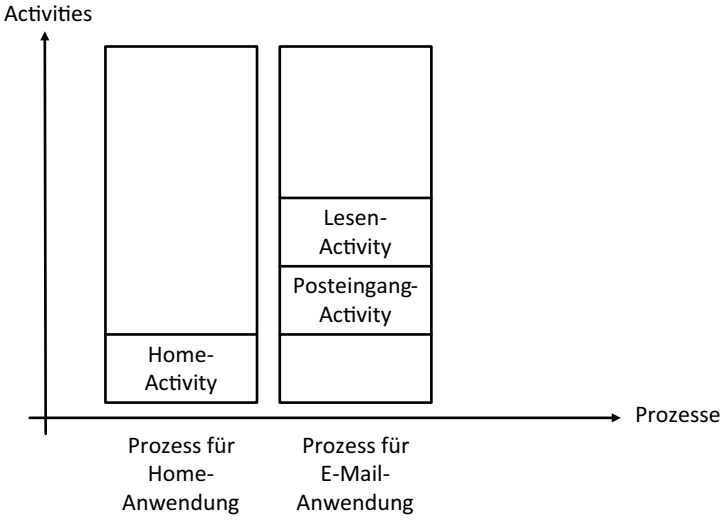

Bild 16.2 Beispiel für existierende Activities und Prozesse (Situation 1)

Im Folgenden klickt die Nutzerin in der gelesenen E-Mail jetzt auf einen Link zu einer HTML-Seite. Daraufhin startet die Browser-Anwendung in einem neuen Prozess und zeigt die entsprechende Web-Seite in einer ersten Browser-Activity an. Auf dieser Seite werde dann ein weiterer Link angeklickt, worauf eine neue Seite in einer neuen Activity angezeigt werde. Die neue Situation ist in Bild 16.3 dargestellt.

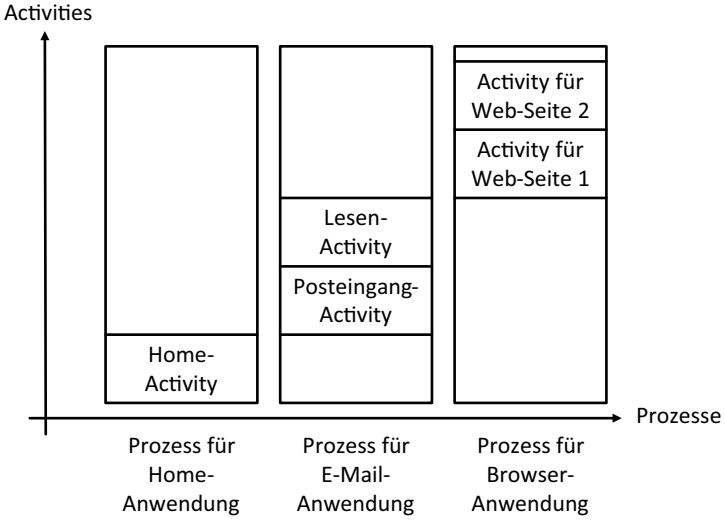

Bild 16.3 Beispiel für existierende Activities und Prozesse (Situation 2)

Wir können uns jetzt vorstellen, dass weitere Prozesse und Activities erzeugt werden, so dass die Ressourcen (insbesondere Speicher) knapp werden. Wenn das Android-Framework einen neuen Prozess starten muss, aber nicht mehr genügend Ressourcen hat, schießt es einen oder mehrere der existierenden Prozesse ab. Für die Situation aus Bild 16.3 ist eine Ressourcenknappheit unwahrscheinlich. Wir nehmen aber vereinfachend eine solche Situation an, ohne dass wir dazu neue Prozesse und neue Activities einzeichnen. Android wählt nur sehr ungern den Prozess aus, dessen Activity gerade angezeigt wird. Auch der Home-Prozess hat gute Chancen zu überleben. Die schlechtesten Karten hat in diesem Fall der E-Mail-Prozess. Wir gehen also davon aus, dass der E-Mail-Prozess zwangsweise beendet wird. Die Situation ist dann wie in Bild 16.4.

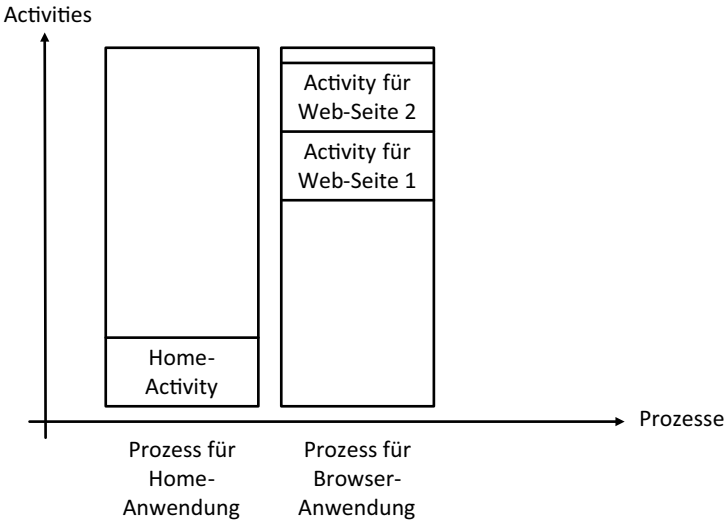

Bild 16.4 Beispiel für existierende Activities und Prozesse (Situation 3)

Wenn die Nutzerin jetzt zwei Mal die Zurück-Schaltfläche betätigt, werden die beiden obersten Elemente auf dem Activity-Keller entfernt. Die jetzt anzuzeigende Activity ist nicht mehr vorhanden. Das Android-Framework hat sich aber gemerkt, welche Activity dies war und zu welcher Anwendung die Activity gehörte. Das Framework startet deshalb einen neuen E-Mail-Prozess, der die Activity zum Lesen einer E-Mail anzeigt. Die Activities können im Android-Framework Informationen abspeichern, damit sie beim Neustart denselben Inhalt wie zuvor anzeigen. Die Activity wird mit den von ihr abgespeicherten Informationen vom Framework versorgt, so dass sie die zuvor angezeigte E-Mail erneut anzeigen kann. Die neue Situation ist in Bild 16.5 zu sehen. Die Nutzerin hat somit von dem Abschießen und Neustarten des E-Mail-Prozesses nichts mitbekommen.

Geht die Benutzerin noch einen Schritt zurück, wird die oberste Activity vom Keller entfernt. Die Activity, zu der man zurückkehren möchte, existiert ebenfalls nicht mehr; allerdings ist der dazugehörige Prozess vorhanden. In diesem Fall wird jetzt nur die Posteingang-Activity neu erzeugt und angezeigt (s. Bild 16.6).

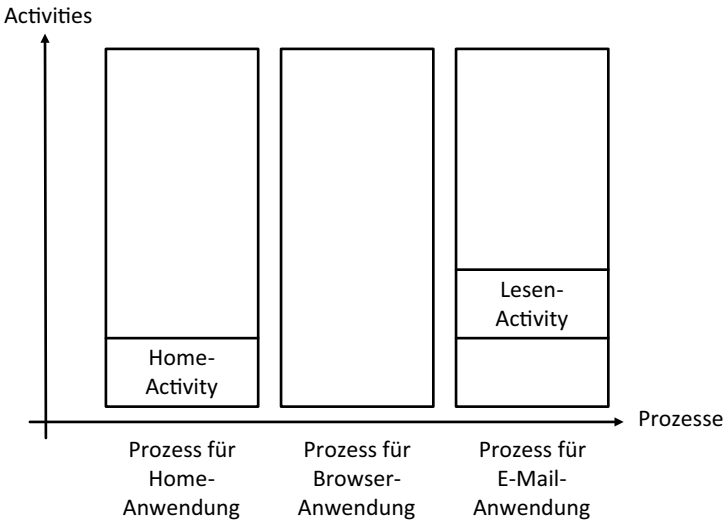

Bild 16.5 Beispiel für existierende Activities und Prozesse (Situation 4)

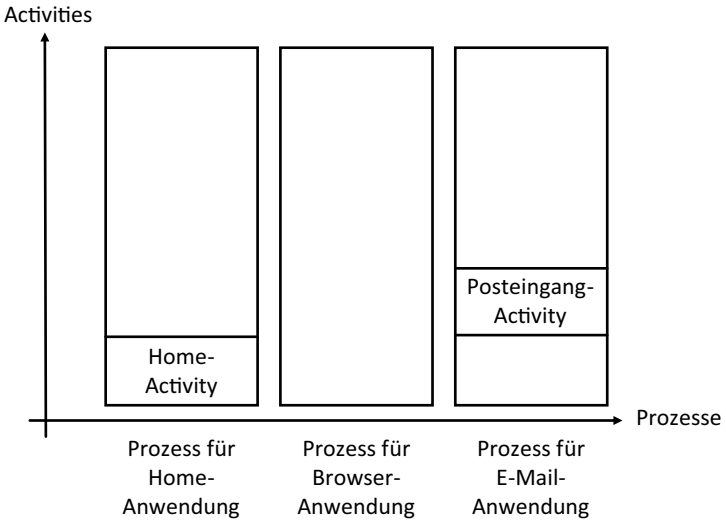

Bild 16.6 Beispiel für existierende Activities und Prozesse (Situation 5)

In Bild 16.5 und Bild 16.6 wurde mit Absicht der Prozess für die Browser-Anwendung nicht entfernt, auch wenn keine Activity dieser Anwendung mehr vorhanden ist. Android löscht nämlich Prozesse nur dann, wenn – wie oben beschrieben – die Ressourcen knapp werden. Ansonsten bleiben auch Prozesse am Leben, für die es keine Activities mehr auf dem Keller gibt. Die Idee ist, dass in Zukunft eventuell wieder eine Activity zu dieser Anwendung gestartet wird und dann der Aufwand zur Prozesserzeugung eingespart werden kann.

Sollten Sie sich übrigens Gedanken gemacht haben, welche und wie viele Threads in den Anwendungsprozessen jeweils laufen, so können Sie für dieses einfache Beispiel erst ein-

mal davon ausgehen, dass es zu jedem Prozess genau einen Thread gibt, der den Bildschirminhalt anzeigt und auf die Eingaben der Benutzerin reagiert (vergleichbar mit dem Event Dispatching Thread in Swing). Zum Thema Threads, die in einer Anwendung laufen, gibt es im Zusammenhang mit den Beispiel-Apps weitere Informationen.

16.3 Komponentenmodell

Wie zuvor beschrieben wurde, besteht eine Anwendung aus einer Menge von Activities. Activities sind aber nicht die einzige Art von Bausteinen einer Android-Anwendung. Außer Activities kann eine App auch einen oder mehrere Services, Content Providers und Broadcast Receivers enthalten. Ein Service ist ein „gesichtsloser" Dienst, also ein Baustein ohne eine grafische Benutzeroberfläche. Ein Service kann wie eine Activity von anderen Bestandteilen derselben oder aber auch einer anderen Anwendung benutzt werden. Ein typisches Beispiel für einen Service ist das Abspielen einer Audio-Datei. Ein Content Provider ermöglicht einen Zugriff auf einen Datenbestand über eine fest vorgegebene Schnittstelle, die aus der Gedankenwelt relationaler Datenbanken stammt. Ein Content Provider muss auf Suchanfragen reagieren, die nicht syntaktisch, aber vom Prinzip her die Form „SELECT … FROM … WHERE …" haben. Weitere von einem Content Provider bereitgestellte Operationen sind wie bei Datenbanken das Einfügen, Ändern und Löschen von Datensätzen. Ein Beispiel für einen Content Provider ist natürlich eine relationale Datenbank, aber auch eine Anwendung, mit der die Kontaktdaten angezeigt und verändert werden können. Wie bei Activities und Services kann ein Content Provider nicht nur von der eigenen Anwendung, sondern auch von einer anderen Anwendung benutzt werden. Die vierte und letzte Art von App-Bausteinen sind Broadcast Receiver, die auf das Senden von Nachrichten reagieren. In diesem Buch beschränken wir unsere Betrachtung auf Activities und Services, während Content Providers und Broadcast Receivers nicht behandelt werden.

Bevor wir unsere Betrachtung des Android-Komponentenmodells fortsetzen, sollte ein terminologisches Problem geklärt werden. Die einzelnen Bestandteile einer App wie Activities und Services werden in der Android-Terminologie als Komponenten bezeichnet. Diese Verwendung des Begriffs Komponente passt aber nur teilweise zu dem Begriff, wie er bisher in diesem Buch gebraucht wurde. Eine Android-Komponente im Sinne dieses Buches ist vielmehr eine komplette Android-Anwendung, denn das ist die Einheit, die verpackt und auf dem Application Framework installiert wird (vgl. Bild 16.1). Der Begriff Komponente ist also problematisch: Wird er im Sinne des Buches für eine Android-App verwendet, so kann das für Android-Leute verwirrend sein; benutzen wir den Begriff aber im Android-Sinne für Activities, Services usw., dann kommt es zu Unstimmigkeiten zwischen der bisherigen Verwendung des Begriffs und der jetzigen. Als Ausweg aus diesem Dilemma werde ich den Begriff Komponente in diesem Kapitel deshalb nur sehr spärlich verwenden. Stattdessen spreche ich von einer Android-Anwendung oder App einerseits und von Bestandteilen oder Bausteinen andererseits.

Für Android-Bausteine wie Activities und Services müssen gewisse Vorgaben eingehalten werden. Insbesondere muss die „Einstiegsklasse" einer Activity bzw. eines Services von der

Android-Klasse Activity bzw. von der Android-Klasse Service abgeleitet werden. Für Activities und Services sind jeweils Lebenszyklen mit mehreren Zuständen definiert. Bei jedem Zustandswechsel wird vom Android Framework eine entsprechende Methode auf das Activity- bzw. Service-Objekt angewendet. Eine ausführlichere Beschreibung folgt bei der Besprechung der Beispiele. Der Programmcode einer Android-App besteht außer den selbst geschriebenen Klassen (Activity- und Service-Klassen sowie darin verwendete Hilfsklassen) auch noch aus einigen automatisch generierten Klassen, auf die ebenfalls bei der Besprechung der Beispiele näher eingegangen wird.

Außer dem Programmcode benötigt eine Android-Anwendung in der Regel noch einige Ressourcen-Dateien. Dazu gehören XML-Dateien zur Beschreibung des Bildschirminhalts einer Activity. Man kann zwar auch in Android ähnlich wie in Swing entsprechende Interaktionselemente wie Labels und Buttons erzeugen und zu der Oberfläche hinzufügen. Typisch in Android ist allerdings die Beschreibung der Oberfläche in einer XML-Datei. Dazu gehört das Layout, aber auch die einzelnen Elemente (Labels, Buttons, Texteingabefelder usw.) mit ihren Eigenschaften (u. a. Beschriftungen). Die Beschriftung der Elemente wird in der Regel aber nicht explizit angegeben, sondern durch einen Verweis auf eine andere XML-Ressourcen-Datei, in der für jeden verwendeten Bezeichner der anzuzeigende Text eingetragen ist. Durch die Benutzung einer solchen XML-Datei können die Beschriftungen der Oberfläche einfach ausgetauscht werden. Insbesondere fällt die Internationalisierung leicht, indem für unterschiedliche Sprachen unterschiedliche Text-Dateien bereitgestellt werden und je nach Spracheinstellung die dazu passende XML-Datei verwendet wird. Aber auch für die Beschreibung der Oberflächen können unterschiedliche XML-Dateien vorgesehen werden, unter denen abhängig vom Typ des verwendeten Geräts (insbesondere abhängig von der Bildschirmgröße) und der aktuellen Lage des Geräts (Hoch- oder Querformat) die jeweils passende ausgewählt wird. Weitere Ressourcen-Dateien sind zum Beispiel Bilddateien für Icons oder Audio- und Videodateien.

Eine weitere wichtige Datei einer Android-App ist die Datei AndroidManifest.xml. In dieser XML-Datei wird die gesamte Anwendung mit all ihren Bausteinen (Activities, Services, Content Providers und Broadcast Receivers) beschrieben. Für jeden Baustein, der von anderen Anwendungen aktiviert werden kann (auch die Aktivierung von der Home-Anwendung ist eine Aktivierung durch eine andere Anwendung), muss festgelegt werden, durch was dieser Baustein aktiviert werden kann. Außerdem enthält die AndroidManifest.xml-Datei die Berechtigungen, welche die Anwendung zur Ausführung benötigt (z. B. das Recht, auf die Lokalisierungsfunktion zuzugreifen oder eine SMS zu versenden). Bei der Installation wird ein Nutzer gefragt, ob der zu installierenden Anwendung die benötigten Rechte eingeräumt werden sollen oder nicht. Auch kann die Anwendung in dieser Konfigurationsdatei eigene Rechte definieren, die andere Anwendungen brauchen, um diese Anwendung zu benutzen.

Eine Android-Anwendung wird in eine APK-Datei (APK: Android Package oder Application Package) gepackt. Wie bei anderen Komponentensystemen auch handelt es sich bei einer APK-Datei um eine Datei im Jar- bzw. Zip-Format. Die APK-Datei enthält den Programmcode in Form einer einzigen DEX-Datei (classes.dex), die Konfigurationsdatei AndroidManifest.xml, die Ressourcendateien sowie ein Verzeichnis META-INF, in dem sich u. a. eine Manifest-Datei MANIFEST.MF und eine Datei CERT.RSA mit einem Zertifikat der Anwendung befinden.

16.4 Anwendung mit einer Activity

16.4.1 Activity mit programmierter Oberfläche

In der ersten App wird demonstriert, dass in Android der Aufbau einer Oberfläche wie in Swing ausprogrammiert werden kann. Bild 16.7 zeigt die Oberfläche im Simulator.

Bild 16.7 Beispiel einer Oberfläche im Android-Simulator

Die Oberfläche besteht aus einem Label mit der Beschriftung „Eingabe“, einem Eingabefeld, in das bereits „Hallo Welt!“ eingetippt wurde, einem Button mit der Beschriftung „Absenden“ und einem Label, in das der eingetippte Text übernommen wird, falls der Button geklickt wird.

Wer Swing programmieren kann, sollte das Programm dazu in Listing 16.1 problemlos verstehen, wobei es nicht auf alle Details ankommt. Die Klasse Activity1 ist aus Activity abgeleitet und überschreibt die Methode onCreate, die vom Framework beim Erzeugen der Activity aufgerufen wird. In der Methode onCreate werden die unterschiedlichen Elemente (ein Label heißt TextElement, ein Eingabefeld EditText) erzeugt. An den Button wird zur Reaktion auf das Klicken ein Listener angemeldet, der den Text aus dem EditText-Element ausliest und in das TextFeld schreibt (Klasse Listener). Alle Elemente werden zu einem Element des Typs LinearLayout hinzugefügt. Anders als der Name andeutet, handelt es sich dabei nicht nur um einen Layout-Manager nach Swing-Terminologie, sondern um einen Container, der mehrere Elemente aufnehmen kann. Das Setzen der vertikalen Orientierung bewirkt, dass alle Elemente untereinander angeordnet sind. Am Ende wird das Linear-Layout-Element mit all seinen Elementen als Inhalt der Activity gesetzt.

Listing 16.1 Klasse Activity1 mit ausprogrammiertem Aufbau der Oberfläche

```
package javacomp.android.activities;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.LinearLayout;
import android.widget.TextView;

public class Activity1 extends Activity
{
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);

        TextView tv1 = new TextView(this);
        tv1.setText("Eingabe:");
        EditText editText = new EditText(this);
        Button b = new Button(this);
        b.setText("Absenden");
        TextView tv2 = new TextView(this);
        Listener l = new Listener(editText, tv2);
        b.setOnClickListener(l);

        LinearLayout ll = new LinearLayout(this);
        ll.setOrientation(LinearLayout.VERTICAL);
        ll.addView(tv1);
        ll.addView(editText);
        ll.addView(b);
        ll.addView(tv2);
        setContentView(ll);
    }
}

class Listener implements View.OnClickListener
{
    private EditText editText;
    private TextView tv;

    public Listener(EditText editText, TextView tv)
    {
        this.editText = editText;
        this.tv = tv;
    }
    public void onClick(View v)
    {
        String input = editText.getText().toString();
        tv.setText(input);
    }
}
```

16.4.2 Activity mit XML-definierter Oberfläche

In Android ist die programmierte Konstruktion der Oberfläche unüblich. In der Regel wird die Oberfläche in einer XML-Datei definiert, die manuell oder über entsprechende Werkzeuge erstellt werden kann. Die folgende XML-Datei erzeugt dieselbe Oberfläche wie in Bild 16.7:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/inputlabel"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/inputText" />

    <EditText
        android:id="@+id/input"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:ems="10"
        android:inputType="text" >
        <requestFocus />
    </EditText>

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/submitText" />

    <TextView
        android:id="@+id/output"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />

</LinearLayout>
```

Zu den einzelnen Elementen gibt es Angaben zum Layout sowie eine Kennung „android:id“, über die im Programm auf die Elemente zugegriffen werden kann. Für das erste TextElement und den Button könnte man den anzuzeigenden Text direkt unter „android:text“ festlegen. In Android ist dies allerdings nicht üblich. Stattdessen wird durch „@string/inputText“ bzw. „@string/submitText“ auf Einträge in der Ressourcendatei strings.xml verwiesen:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">App1</string>
    <string name="inputText">Eingabe:</string>
    <string name="submitText">Absenden</string>
</resources>
```

Mit diesen Ressourcendateien kann die Oberfläche durch Ausführung eines einzigen Java-Befehls aufgebaut werden. Allerdings werden wir zunächst den Listener für den Button noch wie bisher erzeugen und am Button anmelden. Dazu ist ein Zugriff auf die unterschiedlichen Interaktionselemente nötig. Dieser Zugriff kann über die Methode findViewById realisiert werden. Dazu muss man die Kennung eines Elements in Form eines Werts vom Typ int angeben. Für alle in den obigen Dateien vorkommenden Elemente werden in einer automatisch generierten Klasse R (s. Listing 16.2) entsprechende Konstanten definiert.

Listing 16.2 Automatisch generierte Klasse R

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package javacomp.android.activities;

public final class R
{
    public static final class attr
    {
    }
    public static final class drawable
    {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id
    {
        public static final int button=0x7f050002;
        public static final int input=0x7f050001;
        public static final int inputlabel=0x7f050000;
        public static final int output=0x7f050003;
    }
    public static final class layout
    {
        public static final int main=0x7f030000;
    }
    public static final class string
    {
        public static final int app_name=0x7f040000;
        public static final int inputText=0x7f040001;
        public static final int submitText=0x7f040002;
    }
}
```

Die Konstante main in der inneren Klasse layout bezieht sich auf die Datei mit dem Namen main.xml.

Die Methode onCreate aus Listing 16.1 vereinfacht sich dadurch nun erheblich (s. Listing 16.3). Bei setContentView wird statt des LinearLayout-Objekts die Kennung für die Datei main.xml angegeben.

Listing 16.3 Klasse Activity1 mit XML-definierter Benutzeroberfläche (Variante 1)

```
package javacomp.android.activities;

import ...

public class Activity1 extends Activity
{
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        EditText editText = (EditText) findViewById(R.id.input);
        TextView textView = (TextView) findViewById(R.id.output);
        Listener l = new Listener(editText, textView);
        Button b = (Button) findViewById(R.id.button);
        b.setOnClickListener(l);
    }
}

class Listener implements View.OnClickListener
{
    //wie bisher: ...
}
```

Das Programm kann weiter vereinfacht werden, indem in der XML-Datei auch die Methode definiert wird, die beim Klicken aufgerufen werden soll:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="clicked"
    android:text="@string/submitText" />
```

Durch „android:onClick“ wird als Reaktionsmethode die Methode mit dem Namen clicked bestimmt. Eine solche Methode muss eine öffentliche Methode unserer Activity-Klasse sein, die den Rückgabetyp void und einen Parameter des Typs View hat. Die Klasse Listener, welche die Schnittstelle View.OnClickListener implementierte, ist jetzt nicht mehr nötig. In Listing 16.4 findet sich eine weitere Version unserer Activity-Klasse.

Listing 16.4 Klasse Activity1 mit XML-definierter Benutzeroberfläche (Variante 2)

```
package javacomp.android.activities;

import ...

public class Activity1 extends Activity
{
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
```

```
    public void clicked(View view)
    {
        EditText editText = (EditText) findViewById(R.id.input);
        String input = editText.getText().toString();
        TextView textView = (TextView) findViewById(R.id.output);
        textView.setText(input);
    }
}
```

Um das Programm im Simulator oder auf einem realen Android-Gerät ausführen zu können, wird noch eine AndroidManifest.xml-Datei benötigt. In dieser muss die Activity definiert sein. Zu dieser Activity muss ein Intent-Filter angegeben werden, der es ermöglicht, dass die Activity durch Anklicken ihres Icons gestartet werden kann:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="javacomp.android.activities"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="15" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity
            android:name=".Activity1"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
                 android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Der angegebene Intent-Filter legt diese Activity als Einstiegs-Activity der Anwendung fest. Über Intent-Filter erfahren Sie im kommenden Abschnitt mehr.

16.5 Anwendung mit mehreren Activities

16.5.1 Start einer Activity mit explizitem Intent

Wir wollen unsere Anwendung aus dem vorigen Abschnitt nun so verändern, dass beim Drücken auf den Absenden-Button in Bild 16.7 der eingegebene Text nicht in dieser Activity, sondern in einer Folge-Activity angezeigt wird. Zum Starten einer Folge-Activity braucht man Objekte des Typs Intent. Intent heißt auf Deutsch Absicht. Es wird damit ausgedrückt, dass ein Intent-Objekt die Absicht anzeigt, eine neue Aktivität (oder wie später zu sehen

sein wird auch einen neuen Service) zu starten. Man unterscheidet explizite und implizite Intents. Ein expliziter Intent gibt direkt die Klasse in Form eines Class-Objekts der zu startenden Activity an. Wenn die startende und die zu startende Activity zur selben Anwendung gehören, können explizite Intents verwendet werden. Bei impliziten Intents wird statt einer Klasse eine Action in Form eines Strings angegeben. Dieser Action-String muss dann aber als Intent-Filter für die zu startende Activity in der AndroidManifest.xml-Datei angegeben sein. Implizite Intents können zum Starten von Activities (und Services) derselben oder auch anderer Anwendungen eingesetzt werden. Sowohl in expliziten als auch in impliziten Intents können anwendungsabhängige Daten gespeichert und wieder ausgelesen werden. Für das im Folgenden zu realisierende Beispiel übergeben wir im Intent-Objekt den eingetippten Text.

Zur Realisierung des beschriebenen Vorhabens sind drei Dinge notwendig:

- eine weitere XML-Datei zur Festlegung der Oberfläche für die neue Activity sowie dadurch eventuell notwendige Erweiterungen in der Datei strings.xml,
- Änderungen an der Klasse Activity1 zum Starten der neuen Activity sowie die Klasse für die neue Activity
- und schließlich ein zusätzlicher Eintrag für die neue Activity in der Datei AndroidManifest.xml.

Die neue XML-Layout-Datei sieht sehr ähnlich aus wie main.xml. Ihre Wiedergabe sowie die Ergänzungen in strings.xml ersparen wir uns. Die Methode clicked der Klasse Activity1 wird so verändert, dass der in das Textfeld eingetippte Text nicht in das Ausgabefeld geschrieben, sondern in einem neu erzeugten expliziten Intent-Objekt unter einem frei gewählten Namen gespeichert wird. Das Intent-Objekt wird der Methode startActivity übergeben, womit die neue Aktivität gestartet wird. Der Code der Methode clicked aus Listing 16.4 ändert sich wie folgt:

```
public void clicked(View view)
{
    EditText editText = (EditText) findViewById(R.id.input);
    String input = editText.getText().toString();
    Intent intent = new Intent(this, Activity2.class);
    intent.putExtra(MESSAGE, input);
    startActivity(intent);
}
```

MESSAGE ist eine öffentliche, frei gewählte String-Konstante, die zusätzlich in der Klasse Activity1 definiert wurde.

Die Klasse Activity2 ist in Listing 16.5 zu finden.

Listing 16.5 Klasse Activity2

```
package javacomp.android.activities;

import ...

public class Activity2 extends Activity
{
    protected void onCreate(Bundle savedInstanceState)
    {
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity2);
        Intent intent = getIntent();
        String message = intent.getStringExtra(Activity1.MESSAGE);
        TextView textView = (TextView) findViewById(R.id.msglabel);
        if(message != null)
        {
            textView.setText(R.string.receiveSuccessMessage);
            EditText edit = (EditText) findViewById(R.id.inoutput);
            edit.setText(message);
        }
        else
        {
            textView.setText(R.string.receiveFailureMessage);
        }
    }
}
```

Die Methode onCreate besorgt sich nach dem Aufbau der Oberfläche eine Referenz auf den startenden Intent durch Aufruf der geerbten Methode getIntent. Dann wird versucht, vom Intent-Objekt den unter einer bestimmten Kennung abgespeicherten String auszulesen. Wenn dies gelingt, wird eine Erfolgsmeldung ausgegeben, deren Text in strings.xml definiert ist, und der im Intent transportierte String in ein Textfeld des Typs EditText geschrieben, das die Kennung inoutput besitzt, wobei die Bezeichnung später erst verständlich werden wird. Falls der Intent die Nachricht nicht enthält, wird eine ebenfalls in strings.xml festgelegte Fehlernachricht ausgegeben. Dies kann passieren, wenn man ermöglicht, die Activity2 auch auf andere Weise zu starten.

Als letzte Änderung muss die neue Activity auch noch in AndroidManifest.xml eingetragen werden:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <application ...>
        ...
        <activity android:name=".Activity2"></activity>
    </application>
</manifest ...>
```

16.5.2 Start einer Activity mit implizitem Intent

Statt eines expliziten Intents kann in der Methode clicked auch ein impliziter Intent erzeugt werden. Wenn die Zeile

```
Intent intent = new Intent(this, Activity2.class);
```

durch

```
Intent intent = new Intent(ACTION);
```

ersetzt wird, wird ein anderer Intent-Konstruktor verwendet, der einen impliziten Intent erzeugt, wobei ACTION eine in der Klasse Activity1 definierte String-Konstante ist. Damit

die Ausführung wieder erfolgreich ausführbar ist, muss dieser ACTION-String als Intent-Filter in der AndroidManifest.xml-Datei spezifiziert werden. Die Zeile

```
<activity android:name=".Activity2"></activity>
```

sollte durch

```
<activity android:name=".Activity2">
    <intent-filter>
        <action android:name="javacomp.android.activities.ACTION"/>
        <category android:name="android.intent.category.DEFAULT"/>
    </intent-filter>
</activity>
```

ersetzt werden. Der in <action> angegebene Name muss der String sein, der in der Klasse Activity1 als Konstante ACTION definiert wird. Nehmen Sie bitte einfach hin, dass das Element <category> notwendig ist und die hier angegebene Variante funktioniert. Lesen Sie bei weiterem Interesse in der Android-Dokumentation die Bedeutung der Kategorien nach. Übrigens kann ein <activity>-Element beliebig viele Elemente der Art <intent-filter> enthalten (ein Intent-Filter auch mehrfach <action> und <category>). Eine Activity kann somit als eine Art Hauptklasse oder Einstiegsklasse der gesamten Anwendung definiert werden wie Activity1 und zusätzlich können weitere Actions angegeben werden, mit der die Activity gestartet werden kann.

Wenn der in einem impliziten Intent angegebene Action-String in keiner Anwendung als Intent-Filter definiert ist, wirft der Aufruf der Methode startActivity eine Ausnahme, so dass die Anwendung zwangsweise beendet wird. Wenn es hingegen mehr als eine Activity gibt, in deren Intent-Filter der Action-String vorkommt, zeigt Android einen Dialog an, über den der Benutzer die entsprechende Activity auswählen kann. Dabei kann er zusätzlich angeben, dass seine Wahl für spätere Fälle gemerkt und später ohne Zutun des Benutzers wiederholt wird.

16.5.3 Activity mit Resultat

Eine typische Anwendung ist die Aktivierung einer neuen Activity A2 durch eine Activity A1, um etwas in A2 auszuwählen oder einzugeben, was dann in A1 benutzt wird. In einem solchen Fall muss eine Activity mit Resultat eingesetzt werden. Zur Demonstration verändern wir unsere Anwendung so, dass in Activity2 der aus Activity1 übernommene String, der ja vorsorglich schon in einem Eingabefeld angezeigt wird, durch die Benutzerin verändert oder ergänzt werden kann. Wenn die Benutzerin dann auf einen Button der Activity2 drückt, um diese Activity zu beenden, soll die dann wieder auf dem Bildschirm erscheinende Activity1 den in Activity2 veränderten String in ihrem Ausgabefeld (in Bild 16.7 das Feld unter dem Absenden-Button) anzeigen. Zur Realisierung sind Änderungen in beiden Klassen Activity1 und Activity2 nötig.

In der Methode clicked der Klasse Activity1 wird zuerst die Zeile

```
startActivity(intent);
```

durch

```
startActivityForResult(intent, REQUEST_CODE);
```

ersetzt. Als Intent können wir übrigens einen expliziten oder einen impliziten Intent verwenden. Die Methode startActivityForResult hat neben dem Intent einen weiteren Parameter des Typs int. Wir benutzen die in der Klasse Activity1 definierte Konstante REQUEST_CODE, deren Wert frei gewählt wurde. Dieser Code hat nur dann eine Bedeutung, wenn von einer Activity aus mehrere andere Activities mit Resultat gestartet werden können. Wenn man in der startenden Activity auf die Rückkehr aus einer der anderen Activities reagiert (s. unten), dann steht der beim Start angegebene Code zur Verfügung. Wenn man für unterschiedliche Starts jeweils einen unterschiedlichen Code angibt, kann man später anhand des Codes unterscheiden, zu welchem Activity-Start die Rückkehr behandelt wird.

In Activity2 fügen wir einen Button mit der Beschriftung „Zurück“ hinzu. Als Reaktion auf das Klicken auf diesen Button soll eine Methode namens clicked aufgerufen werden. Die Klasse Activity2 (s. Listing 16.5) hatte bisher noch keine solche Methode. Sie muss ergänzt werden:

```
public void clicked(View view)
{
    EditText editText = (EditText) findViewById(R.id.inoutput);
    String input = editText.getText().toString();
    Intent intent = getIntent();
    intent.putExtra(Activity1.MESSAGE, input);
    setResult(RESULT_OK, intent);
    finish();
}
```

In der Methode clicked wird zuerst der Text, der aus Activity1 übernommen und eventuell von einer Benutzerin verändert wurde, ausgelesen und in einem Intent abgespeichert. Danach wird dieses Intent-Objekt zusammen mit einem Rückgabe-Code (in diesem Fall der Code für den Erfolgsfall) als Ergebnis gesetzt. Schließlich beendet sich Activity2 selbst, so dass durch das Klicken auf den Zurück-Button tatsächlich wieder Activity1 erscheint.

In der Klasse Activity1 muss auf die Rückkehr von Activity2 reagiert werden. Dazu muss die Methode onActivityResult überschrieben werden. Diese Methode wird bei der Rückkehr mit dem in startActivityForResult angegebenen Code, einem Ergebniscode und einem Intent aufgerufen. In unserem Fall prüfen wir, ob die Rückkehr erfolgreich ist und zu unserem beim Start angegebenen Code passt. Wenn dies der Fall ist, werden die Daten aus dem Intent gelesen und zusammen mit einer Erfolgsmeldung in das Ausgabefeld geschrieben. Andernfalls erfolgt eine Fehlerausgabe.

```
protected void onActivityResult(int requestCode, int resultCode,
                                Intent intent)
{
    TextView textView = (TextView) findViewById(R.id.output);
    if(resultCode == RESULT_OK && requestCode == REQUEST_CODE)
    {
        String message = intent.getStringExtra(MESSAGE);
        String prefix = getString(R.string.returnSuccessMessage);
        textView.setText(prefix + " " + message);
    }
    else
    {
        textView.setText(R.string.returnFailureMessage);
    }
}
```

Sollte jetzt durch die zahlreichen Änderungen etwas Verwirrung entstanden sein, so verweise ich nochmals auf den Quellcode, der von der Web-Seite zu diesem Buch bezogen werden kann.

16.5.4 Variationen

Wie man sich mit Hilfe von Log-Ausgaben (s. Abschnitt 16.6) leicht überzeugen kann, laufen alle Methoden der Klassen Activity1 und Activity2 im selben Prozess und werden auch alle von demselben Thread ausgeführt. Dabei handelt es sich um den Main-Thread, der in Android die Rolle des aus Swing bekannten Event-Dispatching-Threads hat (also insbesondere auf die grafische Benutzeroberfläche zugreifen darf). Durch eine einfache Konfigurationsänderung kann man erreichen, dass sich das Activity1- und Activity2-Objekt in unterschiedlichen Prozessen befinden. Dazu wird in AndroidManifest.xml die Deklaration von Activity2

```
<activity android:name=".Activity2">
    <intent-filter> ... </intent-filter>
</activity>
```

zu

```
<activity android:name=".Activity2" android:process=":a2">
    <intent-filter> ... </intent-filter>
</activity>
```

verändert. Im Android-Debugger DDMS (Dalvik Debug Monitor Server) kann man erkennen, dass bei der Ausführung der Anwendung zunächst ein Prozess mit dem Namen javacomp.android.activities vorhanden ist. Klickt man auf den Button in Activity1, so wird ein neuer Prozess mit dem Namen javacomp.android.activities:a2 erzeugt. Aus Sicht eines Nutzers ist allerdings kein Unterschied erkennbar.

Die beiden Activities könnten auch auf zwei unterschiedliche Anwendungen aufgeteilt werden, wobei man dann einen impliziten Intent verwenden sollte. Auch in diesem Fall würden sich die Activities in unterschiedlichen Prozessen befinden. Und auch in diesem Fall ergibt sich aus Sicht einer Benutzerin keine erkennbare Änderung.

Es ist sogar möglich, dass man während der Anzeige von Activity2 über den DDMS-Debugger den Prozess für Activity1 zwangsbeendet. Wenn man in Activity2 den Zurück-Button drückt, sieht man selbst dann keinen Unterschied (mit der Ausnahme, dass es ein klein wenig länger dauert, bis Activity1 wieder erscheint). Sogar der zuvor in Activity1 eingetippte Text steht wieder im Eingabefeld. Dieses Verhalten wurde in Abschnitt 16.2 bereits beschrieben: Android kann aus Ressourcenknappheit jederzeit laufende Prozesse beenden, ohne dass dies den Nutzern auffällt. Das Beenden von Prozessen kann man probehalber auch mit Hilfe des Debuggers erreichen.

Das beschriebene Verhalten wird dadurch erreicht, dass in Android im Rahmen des Lebenszyklus der Activities (s. 16.6) der Zustand der Standard-Interaktionselemente gespeichert wird. Bei der Wiederherstellung wird der gespeicherte Zustand dann beim Neuaufbau der Oberfläche zur Initialisierung der Elemente verwendet. Wenn man eigene Interaktionselemente entwickelt, muss man selbst für das Abspeichern und Wiederherstellen sorgen.

Anhand dieser Erläuterungen wird eines der wesentlichen Prinzipien von Android deutlich: Zur Durchführung einer bestimmten Aufgabe werden mehrere Activities gestartet. Es ist dabei weitgehend unabhängig, zu welchen Anwendungen diese Activities gehören und in welchen Prozessen diese ausgeführt werden. Es ist sogar möglich, dass der Prozess, in dem eine Activity läuft, im Laufe der Zeit wechselt, ohne dass dies von den Nutzern wahrgenommen wird.

16.6 Lebenszyklus von Activities

Im Rahmen des Lebenszyklus einer Activity haben wir bisher nur die Methode onCreate benutzt, die bei der Erzeugung einer Activity aufgerufen wird. Tatsächlich gibt es aber für Activities ein relativ umfangreiches Zustandsdiagramm (s. Bild 16.8) zur Beschreibung des Lebenszyklus, in dessen Verlauf eine ganze Reihe von Methoden aufgerufen werden, die in der Basisklasse Activity vorhanden sind und die in der eigenen aus Activity abgeleiteten Klasse überschrieben werden können.

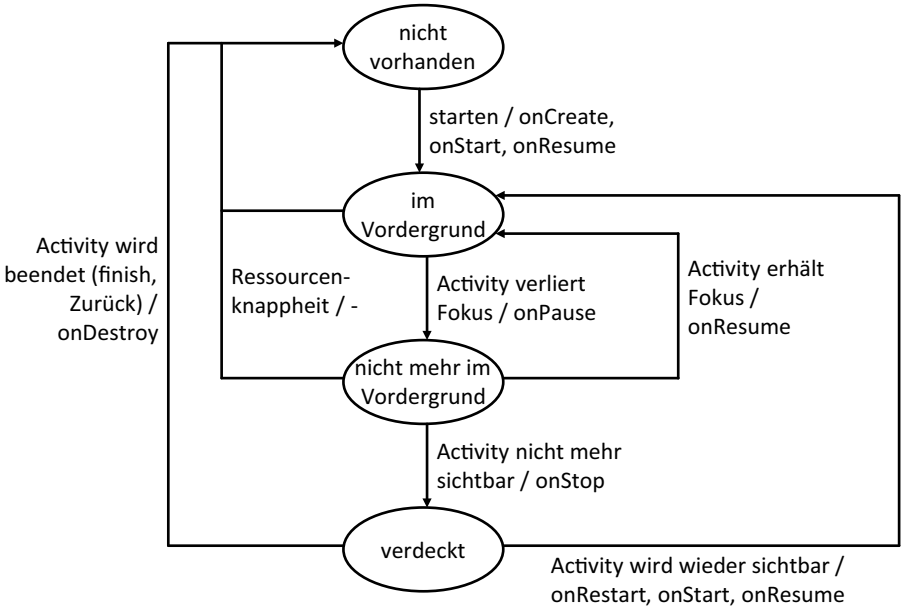

Bild 16.8 Lebenszyklus einer Activity

In Bild 16.8 sind die Zustände, die ein Activity-Objekt im Lauf seines Lebens durchläuft, als Ellipsen dargestellt. Zwischen den Zuständen gibt es Übergänge, die in Form von Pfeilen zu sehen sind. Die Beschriftung der Pfeile hat die Form „Ereignis/Reaktion“. Vor dem Schrägstrich steht das Ereignis, das den Zustandsübergang auslöst, während nach dem Schrägstrich die Methoden aufgeführt sind, die durch diesen Zustandswechsel ausgeführt werden. Zu Beginn wird nicht nur onCreate, sondern auch noch onStart und onResume ausgeführt.

Damit ist die Activity auf dem Bildschirm zu sehen und hat den Eingabefokus. Eine Activity kann danach zum Beispiel durch einen Dialog teilweise verdeckt sein. In diesem Fall ist sie im Zustand „nicht mehr im Vordergrund“. Wenn dagegen eine Activity eine neue Activity startet und die alte Activity dadurch komplett verdeckt wird, gelangt die alte Activity in den Zustand „verdeckt“. Die weitere Zustandsübergänge sollten ohne weitere Erläuterung verständlich sein.

Als Demonstration können wir alle in Bild 16.8 vorkommenden Methoden in unseren beiden Activity-Klassen überschreiben und in alle diese Methoden eine Log-Ausgabe einbauen. Für die Methode onCreate könnte das zum Beispiel so aussehen:

```
Log.d(getClass().getName(), "onCreate");
```

Beim Start der Anwendung erhalten wir dann im LogCat-Bereich von Eclipse die folgende Ausgabe (vereinfacht durch Entfernung der Zeitstempel und weiterer Informationen):

```
javacomp.android.activities.Activity1: onCreate
javacomp.android.activities.Activity1: onStart
javacomp.android.activities.Activity1: onResume
```

Dies ist die erwartete Ausgabe beim Start einer Activity. Wenn man in Activity1 auf den Absenden-Button drückt und dadurch Activity2 startet, sieht man Folgendes:

```
javacomp.android.activities.Activity1: onPause
javacomp.android.activities.Activity2: onCreate
javacomp.android.activities.Activity2: onStart
javacomp.android.activities.Activity2: onResume
javacomp.android.activities.Activity1: onStop
```

Man erkennt, dass auf Activity1 zuerst onPause angewendet wird. Dann wird Activity2 durch onCreate, onStart und onResume aufgebaut. Nachdem dieser Aufbau erfolgreich verlaufen ist und Activity1 dadurch vollständig verdeckt wird, wird auf Activity1 jetzt auch noch onStop angewendet.

Wenn man aus Activity2 zurückkehrt, erfolgt wieder ein Activity-Wechsel, allerdings in die andere Richtung:

```
javacomp.android.activities.Activity2: onPause
javacomp.android.activities.Activity1: onRestart
javacomp.android.activities.Activity1: onStart
javacomp.android.activities.Activity1: onResume
javacomp.android.activities.Activity2: onStop
javacomp.android.activities.Activity2: onDestroy
```

Zuerst wird onPause auf Activity2 angewendet, dann erfolgt der Zustandswechsel von „verdeckt“ zu „im Vordergrund“ für Activity1 mit Aufruf von onRestart, onStart und onResume. Nachdem Activity1 wieder zu sehen ist, wird onStop auf Activity2 angewendet. Da man zu einer Activity, die vom Activity-Keller entfernt wurde, nie mehr zurückkehren kann, wird auf Activity2 auch noch onDestroy aufgerufen. Das Activity2-Objekt kann anschließend durch die Garbage Collection entsorgt werden.

Mit Hilfe der Log-Ausgaben lässt sich auch untersuchen, was passiert, wenn das Android-Gerät um 90 Grad gedreht wird (also vom Hochformat zum Querformat oder umgekehrt):

```
javacomp.android.activities.Activity1: onStop
javacomp.android.activities.Activity1: onDestroy
javacomp.android.activities.Activity1: onCreate
javacomp.android.activities.Activity1: onStart
javacomp.android.activities.Activity1: onResume
```

Wie man sieht, wird die aktuelle Activity komplett beendet (onStop, onDestroy), ein neues Activity-Objekt erzeugt und dieses initialisiert (onCreate, onStart, onResume). Dass ein neues Activity-Objekt erzeugt wird, ist an der obigen Ausgabe nicht zu erkennen; man kann das aber durch eine weitere Log-Ausgabe im Konstruktor der Klasse Activity1 nachweisen. Durch eine Ausgabe im Konstruktor kann man auch sehr schön zeigen, dass es mehrere Objekte derselben Activity-Klasse zu einem Zeitpunkt auf dem Activity-Keller geben kann. Zu diesem Zweck habe ich unserer Activity2 einen weiteren Button hinzugefügt, mit der wie in Activity1 über einen Intent die Activity2 erneut gestartet wird (also eine Art rekursiver Aufruf von Activity2). Den Objekten habe ich eine laufende Nummer im Konstruktor zugewiesen und diese Nummer in die Log-Ausgaben geschrieben. Man kann dann sehr schön sehen, dass bei jedem Activity-Start ein neues Objekt erzeugt und auf den Activity-Keller gelegt wird. Beim Zurückgehen ist auch gut erkennbar, wie jedes dieser Objekte seinen eigenen Lebenszyklus hat.

16.7 Service und Activity im Vergleich

Ein Service hat viele Ähnlichkeiten mit einer Activity, aber auch einen wesentlichen Unterschied: Ein Service hat keine Benutzeroberfläche. Wie eine Activity wird ein Service über einen expliziten oder impliziten Intent gestartet. Ein Service kann damit von anderen Bausteinen derselben Anwendung oder aber auch einer anderen Anwendung benutzt werden. Ein Service muss wie eine Activity in der Konfigurationsdatei AndroidManifest.xml vereinbart werden. Auch in diesem Fall kann man bestimmen, dass der Service in einem eigenen Prozess laufen soll. Wenn er durch einen impliziten Intent gestartet werden soll, muss in der Konfigurationsdatei ein Intent-Filter für ihn angelegt werden. Auch ein Service muss von einer vorgegebenen Klasse (in diesem Fall von der Klasse Service) abgeleitet werden und einige Methoden ihrer Basisklasse überschreiben. Diese Methoden werden vom Framework im Rahmen des Lebenszyklus aufgerufen, den ein Service-Objekt durchläuft. Es gibt zwei Sorten von Services: ungebundene und gebundene Services. Die Bezeichnung ist nicht besonders aussagekräftig. Anhand der folgenden Beispiele wird man den Unterschied hoffentlich besser verstehen. Wir beginnen mit einem ungebundenen Service.

16.8 Anwendung mit einem ungebundenen Service

Sowohl ungebundene als auch gebundene Services müssen von derselben Basisklasse Service abgeleitet werden. Für ungebundene Services kann man (muss man aber nicht) die Methoden onCreate, onStartCommand und onDestroy überschreiben. Die Methode onBind muss in jedem Fall überschrieben werden. Diese Methode spielt nur bei gebundenen Services eine Rolle. Für einen ungebundenen Service sollte man sie mit „return null;“ realisieren, was wir im Folgenden auch tun werden. Da gebundene Services wesentlich interessanter sind als ungebundene, werden wir für einen ungebundenen Service nur ein sehr einfaches Beispiel betrachten (s. Listing 16.6).

Listing 16.6 Klasse MyUnboundService als Beispiel für einen ungebundenen Service

```
package javacomp.android.unboundservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;

public class MyUnboundService extends Service
{
    public IBinder onBind(Intent intent)
    {
        return null;
    }

    public void onCreate()
    {
        Log.d(this.getClass().getName(), "onCreate");
    }

    public void onDestroy()
    {
       Log.d(this.getClass().getName(), "onDestroy");
    }

    public int onStartCommand(Intent intent, int flags, int startid)
    {
        Log.d(this.getClass().getName(), "onStartCommand");
        new ServiceThread().start();
        return START_STICKY;
    }
}

class ServiceThread extends Thread
{
    public void run()
    {
        for (int i = 1; i <= 10; i++)
        {
            Log.d(this.getClass().getName(), "service with thread "
                                             + getName()
                                             + ", iteration no. "
```

```
                                                + i);
            try
            {
                Thread.sleep(2000);
            }
            catch(InterruptedException e)
            {
            }
        }
    }
}
```

Wie schon beschrieben muss die Methode bind der Klasse MyUnboundService überschrieben werden. Die Methoden onCreate und onDestroy erzeugen lediglich eine Log-Ausgabe. Die Methode onStartCommand erzeugt einen Thread, der im Abstand von 2 Sekunden zehn Mal etwas in den Log ausgibt.

Damit der Dienst genutzt werden kann, muss er in AndroidManifest.xml vereinbart werden:

```
<service android:name=".MyUnboundService" ></service>
```

Auch hier wäre es wieder möglich anzugeben, dass der Service in einem eigenen Prozess läuft. Auch könnten Intent-Filter angegeben werden, falls der Dienst über einen impliziten Intent gestartet werden soll.

In unserer Anwendung gibt es zusätzlich eine einfache Activity (s. Listing 16.7) mit zwei Buttons zum Starten und Anhalten des Dienstes. In der XML-Layout-Datei main.xml ist festgelegt, dass beim Drücken der Buttons die Methode clickedOnStart bzw. clickedOnStop aufgerufen werden soll.

Listing 16.7 Activity-Klasse zum Ausprobieren des ungebundenen Service

```
package javacomp.android.unboundservice;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ActivityForUnboundService extends Activity
{
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void clickedOnStart(View v)
    {
        startService(new Intent(this, MyUnboundService.class));
    }

    public void clickedOnStop(View v)
    {
        stopService(new Intent(this, MyUnboundService.class));
    }
}
```

In Listing 16.7 ist zu erkennen, dass zum Starten und zum Anhalten eines Dienstes jeweils ein Intent-Objekt erzeugt wird (in diesem Fall handelt es sich um einen expliziten Intent) und dieses als Argument den Methoden startService bzw. stopService übergeben wird. Der Aufruf von startService bewirkt die Erzeugung eines MyUnboundService-Objekts sowie die Anwendung der Methoden onCreate und onStartCommand auf diesem neuen Objekt. Alle Methoden (sowohl der Activity- als auch der Service-Klasse) werden vom Main-Thread des Anwendungsprozesses ausgeführt. Wie schon zuvor erwähnt wurde, ist dieser Thread vergleichbar mit dem Event-Dispatching-Thread von Swing. Wie bei Swing gilt, dass dieser Thread immer schnell auf Eingaben reagieren können soll. Somit sollten alle Methoden, die von diesem Thread ausgeführt werden, nur eine kurze Ausführungszeit benötigen. Wenn doch eine Aktion mit längerer Ausführungszeit durchgeführt werden muss, dann sollte diese in einen Thread ausgelagert werden. Dies wird in der Methode onStartCommand durch Erzeugung und Starten eines Threads angedeutet.

Wenn der Start-Button geklickt wird, wird ein neues MyUnboundService-Objekt erzeugt und auf dieses die Methoden onCreate und onStartCommand angewendet, wodurch ein neuer Thread erzeugt wird. Wenn der Start-Button dann öfter gedruckt wird, wird jedes Mal nur noch die Methode onStartCommand auf das schon vorhandene Service-Objekt angewendet. Wenn der Stopp-Button gedrückt wird, wird die Methode onDestroy auf dem Service-Objekt aufgerufen und das Service-Objekt steht für die Abfallsammlung (Garbage Collection) bereit. Wenn anschließend wieder auf Start geklickt wird, wird ein neues Service-Objekt erzeugt und auf dieses dann wieder onCreate und onStartCommand angewendet. Ganz wichtig ist dies: Wie man an der Ausgabe erkennen kann, bewirkt das Beenden eines Dienstes nicht, dass alle von ihm gestarteten Threads automatisch beendet werden. Wenn man also den Stopp-Button relativ schnell nach dem Start-Button drückt, sieht man, dass der Thread weiterläuft, auch wenn onDestroy ausgeführt wurde. Es liegt also in der Verantwortung der Anwendungsentwicklerin, dass eventuell noch laufende Threads bei onDestroy beendet werden, falls dies so sein soll. Im Beispielprogramm wurde absichtlich darauf verzichtet, um den soeben geschilderten Effekt demonstrieren zu können.

Zum Abschluss der Besprechung dieser Beispielanwendung noch eine Bemerkung zum Rückgabewert der Methode onStartCommand in Listing 16.6: Der Rückgabewert spielt dann eine Rolle, wenn Android sich entschließt, den Prozess, in dem der Service läuft, abzuschießen. Wenn Android später wieder freie Ressourcen hat, wird bei einer Rückgabe von START_STICKY automatisch ein neuer Prozess erzeugt, ein Service-Objekt erzeugt und darauf onCreate und onStartCommand mit einem Null-Intent aufgerufen. Das Intent-Objekt, welches den Aufruf von onStartCommand veranlasst hat, ist das erste Argument dieser Methode. Wie im Beispiel mit den zwei Activities in Abschnitt 16.5 kann man auch diesem Intent zusätzliche Daten übergeben, die in onStartCommand ausgewertet werden. Die beiden weiteren Parameter flags und startId von onStartCommand werden hier nicht erläutert. Es sei auf die Android-Dokumentation verwiesen.

16.9 Anwendung mit einem gebundenen Service

Das Konzept eines gebundenen Service ist äußerst mächtig. Ein gebundener Service stellt nämlich die Android-Version von RMI (Remote Method Invocation) dar, wobei in diesem Fall die Aufrufe nicht wirklich von einem anderen Rechner kommen (also nicht wirklich Remote sind), aber von einer anderen Anwendung bzw. von einem anderen Prozess und damit einem anderen Adressraum. In einer Service-Klasse kann sowohl ein ungebundener als auch ein gebundener Service realisiert werden. Wir betrachten die Dienstarten hier getrennt und konzentrieren uns nach der Besprechung eines ungebundenen Service im Folgenden ausschließlich auf einen gebundenen Service.

16.9.1 AIDL-Schnittstelle

Wie bei RMI benötigt man für einen gebundenen Service zunächst einmal eine Schnittstelle. In RMI definiert man diese als Schnittstelle in Java. In Android muss man diese Schnittstelle in AIDL (Android Interface Definition Language) in einer Datei mit der Endung .aidl angeben. Es wird höchste Zeit, dass wir unser Zählerbeispiel wieder aufgreifen. In Listing 16.8 findet sich eine Counter-Schnittstelle in AIDL.

Listing 16.8 AIDL-Schnittstelle Counter

```
package javacomp.android.boundservice;

interface Counter
{
    int increment();
    int reset();
}
```

Bis jetzt ist der Unterschied zu Java kaum zu erkennen. Diese AIDL-Schnittstelle ist auch eine gültige Java-Schnittstelle. Wenn man allerdings das Schlüsselwort public noch vor die Schnittstelle oder die Methoden schreibt, was in Java geht, dann erhält man einen Fehler in AIDL. Die Entwicklungsumgebung für Android erzeugt aus der AIDL-Datei automatisch Code in Form einer Klasse mit demselben Namen und in demselben Package wie die AIDL-Schnittstelle, in unserem Fall also die Klasse Counter in dem in Listing 16.8 angegebenen Package.

16.9.2 Implementierung einer AIDL-Schnittstelle

Zur Implementierung einer AIDL-Schnittstelle muss man eine Klasse aus der inneren Klasse Stub der automatisch erzeugten Klasse ableiten (in unserem Fall also aus Counter.Stub). Ähnlich wie bei RMI müssen die Methoden alle mit „throws RemoteException“ versehen sein, wobei die Klasse RemoteException eine andere Klasse ist als die Klasse gleichen

Namens von RMI, da sie aus einem anderen Package stammt. In Listing 16.9 ist die Klasse CounterImpl zu finden, wobei die beiden Methoden increment und reset vorsorglich schon einmal synchronized sind.

Listing 16.9 Implementierung der AIDL-Schnittstelle Counter

```
package javacomp.android.boundservice;

import android.os.RemoteException;

public class CounterImpl extends Counter.Stub
{
    private int counter;

    public synchronized int increment() throws RemoteException
    {
        counter++;
        return counter;
    }

    public synchronized int reset() throws RemoteException
    {
        counter = 0;
        return counter;
    }
}
```

16.9.3 Realisierung eines gebundenen Service

Wenn man eine solche Klasse wie CounterImpl hat, dann kann man einen gebundenen Service leicht realisieren. Man leitet eine eigene Klasse aus Service ab, überschreibt onBind und gibt dabei ein Objekt einer solchen Klasse wie CounterImpl zurück (s. Listing 16.10).

Listing 16.10 Klasse MyBoundService zur Realisierung eines gebundenen Service

```
package javacomp.android.boundservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class MyBoundService extends Service
{
    public IBinder onBind(Intent intent)
    {
        return new CounterImpl();
    }
}
```

Wie ein ungebundener Service muss auch ein gebundener Service in AndroidManifest.xml angegeben werden. In diesem Fall soll der Dienst in einem eigenen Prozess laufen und auch von außerhalb unserer Anwendung mit Hilfe eines impliziten Intents genutzt werden können:

```
<service android:name=".MyBoundService" android:process=":service">
    <intent-filter>
        <action
         android:name="javacomp.android.bindservice.BINDSERVICE"/>
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</service>
```

16.9.4 Nutzung eines gebundenen Service

Ähnlich wie bei einem ungebundenen Service muss man zur Nutzung eines gebundenen Service ein Intent-Objekt erzeugen und eine entsprechende Methode aufrufen, der dieses Intent-Objekt übergeben wird. Für einen gebundenen Service heißt die Methode bindService statt startService. Die Methode bindService liefert aber keine Referenz auf das eigentliche Dienstobjekt (in unserem Fall den Zähler) zurück, sondern man muss in bindService als zweites Argument nach dem Intent ein Objekt einer Klasse angeben, welche die Schnittstelle ServiceConnection mit den Methoden onServiceConnected und onServiceDisconnected implementiert. Im Folgenden gehen Sie bitte davon aus, dass serviceConn auf ein Objekt einer solchen SerivceConnection implementierenden Klasse zeigt. Dann können wir den Kontakt zu unserem gebundenen Service wie folgt herstellen:

```
Intent intent =
            new Intent("javacomp.android.bindservice.BINDSERVICE");
bindService(intent, serviceConn, Context.BIND_AUTO_CREATE);
```

Wie zu sehen ist, hat bindService noch ein drittes Argument. Mit diesem wird in unserem Fall angegeben, dass beim Binden der Dienst automatisch erzeugt werden soll, falls er noch nicht existiert.

Falls das Binden erfolgreich war, was durch den Rückgabewert true von bindService angezeigt wird, wird nach der Rückkehr aus bindService die Methode onServiceConnected auf dem als zweitem Argument angegebenen Objekt aufgerufen.

Zum besseren Verständnis unserer Beispielimplementierung von ServiceConnection sollte man wissen, dass es auch zu dieser Anwendung eine Activity-Klasse namens ServiceActivity gibt, die eine Methode setCounter besitzt (die Klasse ServiceActivity wird zum Platzsparen nicht abgedruckt, kann aber von der Web-Seite zum Buch heruntergeladen werden). Durch setCounter wird ein Counter-Objekt gesetzt, das beim Drücken auf den Button „Erhöhen“ bzw. „Zurücksetzen“ verwendet wird, um den Zähler zu erhöhen bzw. zurückzusetzen. Außerdem zeigt die Activity an, dass der Dienst gebunden ist und verwendet werden kann. Wird als Argument von setCounter null angegeben, dann wird eine Meldung angezeigt, dass der Dienst nicht zur Verfügung steht. Die Klasse MyServiceConnection (s. Listing 16.11) implementiert die Schnittstelle ServiceConnection. Die Methode onServiceConnected ist die entscheidende Methode. Hier wird das Dienstobjekt als zweiter Parameter übergeben. Durch Anwendung der statischen Methode asInterface der inneren Klasse Counter.Stub erhält man ein Objekt des Typs Counter zurück, mit dem man auf den Zähler zugreifen kann. Im einfachsten Fall, wenn nämlich der Dienst und die Activity, die den Dienst benutzt, im selben Prozess laufen, ist der Typ des Objekts, das von asInterface zurückgegeben wird, tatsächlich CounterImpl. Man erhält also eine echte Referenz auf das in der Methode onBind

erzeugte Objekt (s. Listing 16.10). Wenn dagegen der Dienst in einem anderen Prozess läuft, dann erhält man nur einen Stub, der dafür sorgt, dass beim Aufruf der Methoden increment und reset die Prozessgrenzen überwunden und die entsprechende Methode auf dem Objekt, das sich in einem anderen Prozess befindet, aufgerufen wird. In seiner Anwendung muss man aber die beiden Fälle nicht unterscheiden.

Listing 16.11 Implementierung der Schnittstelle ServiceConnection durch die Klasse MyServiceConnection

```
package javacomp.android.boundservice;

import android.content.ComponentName;
import android.content.ServiceConnection;
import android.os.IBinder;

public class MyServiceConnection implements ServiceConnection
{
    private ServiceActivity activity;

    public MyServiceConnection(ServiceActivity activity)
    {
       this.activity = activity;
    }

    public void onServiceConnected(ComponentName name,
                                   IBinder binder)
    {
       Counter c = Counter.Stub.asInterface(binder);
       activity.setCounter(c);
    }

    public void onServiceDisconnected(ComponentName name)
    {
       activity.setCounter(null);
    }
}
```

Wenn man den Dienst wieder abmelden möchte, ruft man die Methode unbindService auf. Hierzu muss man das ServiceConnection-Objekt, das man bei bindService übergeben hat, wieder angeben:

```
unbindService(serviceConn);
```

Als Reaktion auf unbindService wird allerdings die Methode onServiceDisconnected nicht aufgerufen. Die Methode onServiceDisconnected wird nur aufgerufen, wenn der Dienst außerplanmäßig nicht mehr zur Verfügung steht. Dies ist zum Beispiel dann der Fall, wenn er in einem eigenen Prozess läuft und Android diesen Prozess wegen Ressourcenknappheit beendet hat, was man über den Debugger durch Abschießen des Prozesses auch simulieren kann. Android erzeugt den Prozess neu, falls beim Beenden Bindungen vorhanden waren. Alle diese Bindungen werden nach dem Neustart des Prozesses von Android wiederhergestellt. Wenn man also den Prozess über seinen Debugger beendet, zeigt die Activity in ihrer Oberfläche an, dass der Dienst nicht mehr zur Verfügung steht, da onServiceDisconnected aufgerufen wurde. Kurze Zeit später steht der Dienst aber wieder bereit; die ServiceActivity zeigt dies an, ohne dass man dazu irgendetwas tun muss. Die Methode onServiceConnected

wird beim automatischen Neubinden wieder aufgerufen. Diese Robustheit ist ein besonderes Qualitätsmerkmal von Android: Ein zwangsweises Beenden eines Prozesses wird wieder aufgefangen. In unserem Fall ist natürlich der alte Zählerstand weg und die Zählung beginnt wieder bei 0. Durch Abspeichern des Zählers auf nicht-flüchtigem Speicher und Wiedereinlesen des abgespeicherten Werts beim Erzeugen des Zähler-Objekts könnte man dem aber entgegenwirken.

Die beiden Methoden onServiceConnected und onServiceDisconnected werden übrigens vom Main-Thread des Prozesses ausgeführt. Da dieser Thread (als Einziger) auf die grafische Benutzeroberfläche zugreifen darf, ergibt sich kein Problem, wenn man in der Methode setCounter, die von diesen Methoden aufgerufen wird, die Anzeige auf der Oberfläche entsprechend ändert (vgl. zu dieser Problematik Abschnitt 16.9.6).

Wenn die Activity, die den Service gebunden hat, endet ohne zuvor unbindService aufgerufen zu haben, wird eine Ausnahme von Android geworfen. Android erkennt diesen Fall also und hält auf diese Weise den Entwickler zu einem sauberen Programmierstil an. Man sollte sich also spätestens in der Activity-Methode onDestroy von allen noch gebundenen Diensten abkoppeln.

16.9.5 Parameterübergabe durch Call-By-Value-Result

Parameter einer der primitiven Java-Datentypen wie int oder Parameter des Typs String sind bei der Übergabe problemlos. Diese werden als Wert (Call-By-Value), d. h. in Form einer Kopie, übergeben. Will man aber als Parameter ein Objekt seiner eigenen Klasse übergeben, muss man dafür sorgen, dass dieses bei Bedarf auch zu einem anderen Prozess transportiert werden kann. Bei RMI muss man die Klassen solcher Objekte durch Implementierung der leeren Schnittstelle Serializable serialisierbar machen. Dies ist bei Android ähnlich, aber etwas aufwändiger. Man muss die Schnittstelle Parcelable implementieren, die aber nicht leer ist. Man muss darin wirklich die Serialisierung seiner Objekte ausprogrammieren. Als Beispiel nehmen wir an, dass unsere Counter-Schnittstelle eine zusätzliche Methode change bekommen soll, in der ein Objekt einer Klasse übergeben wird, die zwei Werte des Typs int enthält: einen, mit dem der Zähler zuerst multipliziert und einen, der anschließend zum Zähler addiert wird. Aus Demonstrationsgründen soll die Methode change den neuen Wert nicht als Rückgabewert zurückgeben, sondern in der Klasse Data befindet sich ein weiteres Attribut für den Wert des Zählers am Ende der Methode change. Listing 16.12 zeigt einen Ausschnitt der Klasse Data (weitere Methoden wie Konstruktoren, Getter- und Setter-Methoden werden weggelassen).

Listing 16.12 Klasse Data

```
package javacomp.android.boundservice;

import android.os.Parcel;
import android.os.Parcelable;

public class Data implements Parcelable
{
    private int factor;
    private int addend;
    private int result;
```

```
    ...

    public void writeToParcel(Parcel out, int flags)
    {
        out.writeInt(factor);
        out.writeInt(addend);
        out.writeInt(result);
    }

    public void readFromParcel(Parcel in)
    {
        factor = in.readInt();
        addend = in.readInt();
        result = in.readInt();
    }
}
```

Listing 16.12 lässt erahnen, was man tun muss. Die Attribute werden in einer bestimmten Reihenfolge geschrieben. Genau in dieser Reihenfolge kommen sie an und müssen den Attributen zugewiesen werden.

Bevor man die Klasse als Parameter in einer AIDL-Datei verwenden kann, muss für Data eine eigene AIDL-Datei angelegt werden (s. Listing 16.13).

Listing 16.13 AIDL-Datei für die Klasse Data

```
package javacomp.android.boundservice;
parcelable Data;
```

Damit können wir die AIDL-Schnittstelle Counter (s. Listing 16.8) um folgende Methode erweitern:

```
void change(inout Data data);
```

Hier sieht man jetzt eine Abweichung bei der AIDL-Syntax gegenüber Java: Man muss spezifizieren, ob es sich bei dem Parameter um einen Eingabeparameter handelt (in), also eine Datenübergabe vom Aufrufer zur aufgerufenen Methode, oder um einen Ausgabeparameter (out), also von der aufgerufenen Methode zurück zum Aufrufer, oder um beides (inout). Weil wir die Attribute factor und addend an die Methode übergeben und result von der Methode zurückbekommen wollen, brauchen wir inout. Die Implementierung dieser Methode ist sehr naheliegend; die Klasse CounterImpl aus Listing 16.9 wird dazu wie folgt ergänzt:

```
public synchronized void change(Data data) throws RemoteException
{
    counter = counter * data.getFactor() + data.getAddend();
    data.setResult(counter);
}
```

Die neue Methode change kann beispielsweise wie folgt benutzt werden (Counter sei eine Referenz auf den Zähler aus onServiceConnected):

```
Data data = new Data(10, 1);
counter.change(data);
int result = data.getResult();
```

Wenn man die Methode change ausprobiert, kann man feststellen, dass alles funktioniert wie erwartet. Wenn man in der AIDL-Datei statt inout aber nur in angibt, wird zwar der Zähler geändert, aber der neue Wert steht nach dem Aufruf nicht im übergebenen Datenobjekt. Gibt man nur out an, so kommen die übergebenen Attributwerte factor und addend beim Aufrufer nicht an. Oder besser gesagt: Sie kommen mit dem Wert 0 an, so dass der Zähler damit immer auf 0 gesetzt wird. Das Gesagte gilt allerdings nur dann, wenn der Service in einem eigenen Prozess läuft. Befinden sich dagegen der Aufrufer und der Dienst im selben Prozess, so funktioniert immer alles richtig, gleichgültig ob man in, out oder inout angibt. Dies liegt daran, dass in einem solchen Fall ein ganz normaler Methodenaufruf stattfindet. Der Data-Parameter wird wie bei Java üblich als Referenz übergeben, die Serialisierung wird nicht benutzt und in, out bzw. inout wird nicht berücksichtigt. Im Falle unterschiedlicher Prozesse wird das Data-Objekt vor dem Aufruf zum Dienstprozess kopiert, falls in oder inout spezifiziert wurde, und nach dem Aufruf wieder zum Aufrufer zurückkopiert, falls out oder inout in der AIDl-Datei angegeben wurde. Dieses Hin- und Herkopieren beim Methodenaufruf nennt man Call-By-Value-Result.

16.9.6 Parameterübergabe durch Call-By-Reference

Eine Parameterübergabe durch Call-By-Value bzw. Call-By-Value-Result für Objektparameter ist in Java unüblich. Bei RMI kann die allgemein übliche Referenzübergabe (Call-By-Reference) nachgeahmt werden, indem ein in einer Methode übergebenes Objekt ebenfalls ein RMI-Objekt ist. Der Aufrufer erzeugt also zuerst ein Objekt, das von der Ferne aus aufgerufen werden kann. Dieses Objekt wird übergeben, so dass der Aufgerufene auf das Objekt, das der Aufrufer übergeben hat, aus der Ferne sofort (noch während des Methodenaufrufs) oder später (wenn er die Referenz auf das RMI-Objekt speichert) zurückrufen kann. Man bezeichnet dies als Callback. In Android gibt es diese Möglichkeit in ganz analoger Weise.

Zu diesem Zweck muss für den Callback-Parameter ebenfalls eine AIDL-Schnittstelle definiert werden. Als Aufgabe wollen wir die Schnittstelle Counter um eine Methode multiIncrement erweitern, die den Zähler n Mal erhöht (n kann als Argument angegeben werden). Aus Demonstrationsgründen soll die Erhöhung in Einser-Schritten und relativ langsam (also mit Wartezeiten zwischen je zwei Erhöhungen) erfolgen. Jedes Mal, wenn der Zähler erhöht wurde, soll der Aufrufer durch einen Rückruf benachrichtigt werden, dass der Zähler erhöht wurde. Die Rückrufschnittstelle besitzt deshalb eine einzige Methode zum Melden eines neuen Zählerwerts (s. Listing 16.14).

Listing 16.14 AIDL-Rückrufschnittstelle

```
package javacomp.android.boundservice;

interface Callback
{
    void newCounterValue(int newValue);
}
```

Der Typ Callback kann nun als Parametertyp in der AIDL-Schnittstelle Counter verwendet werden. Wir ergänzen die AIDL-Schnittstelle Counter ein weiteres Mal um diese Methode:

```
void multiIncrement(int times, Callback cb);
```

Beim Aufruf dieser Methode soll der Zähler also times Mal erhöht werden und bei jeder Erhöhung soll das Objekt cb, das der Aufrufer bereitstellt, durch Aufruf der Methode newCounterValue benachrichtigt werden. Die Implementierung von multiIncrement, die wir hier nicht wiedergeben, erzeugt einen Thread, der times Mal die Methode increment aufruft und nach jeder Erhöhung cb zurückruft.

Nun fehlt nur noch eine Implementierung der Schnittstelle Callback. Wie zuvor für Counter wird zur Callback-Schnittstelle automatisch eine Java-Klasse namens Callback erzeugt. Zur Implementierung muss man eine eigene Klasse aus der inneren Klasse Stub von Callback ableiten. Die Methode newCounterValue soll den neuen Zählerwert in einem Textfeld auf der Oberfläche ausgeben. Unser erster naiver Realisierungsversuch sieht so aus:

```
package javacomp.android.boundservice;

import ...

public class CallbackImpl extends Callback.Stub
{
    private TextView tv;

    public CallbackImpl(TextView tv)
    {
        this.tv = tv;
    }

    public void newCounterValue(int newValue) throws RemoteException
    {
        tv.setText("neuer Wert: " + newValue);
    }
}
```

So funktioniert es aber nicht, denn der Rückruf wird von einem eigenen Thread ausgeführt. Aus Synchronisationsgründen darf in Android aber nur der Thread auf die Oberfläche zugreifen, der sie aufgebaut hat. Das ist in unserem Fall der Main-Thread. Die Situation ist sehr ähnlich wie bei Java Swing. Auch hier sollte nur der Event-Dispatcher auf die Oberfläche zugreifen. Wenn man die Regel nicht einhält, begeht man einen Synchronisationsfehler, der in Swing unter Umständen nicht auffällt. In Android ist die Situation insofern besser, als bei einem solchen Zugriff durch einen falschen Thread direkt eine Ausnahme geworfen wird. Der Programmiererin wird also sofort der Fehler aufgezeigt.

Die Lösung des Problems ist auch ganz ähnlich wie in Java Swing. Man muss einen Auftrag an den Thread senden, der auf die Oberfläche zugreifen darf. In Android braucht man dazu einen Handler. Über einen Handler können Aufträge an den Thread gesendet werden, der das Handler-Objekt erzeugt hat. In unserem Fall muss also der Main-Thread das Handler-Objekt erzeugen und als Parameter dem Konstruktor der Klasse CallbackImpl übergeben. Für das Senden eines Auftrags an einen Handler gibt es mehrere Varianten. Ich verwende im Folgenden diejenige, die der von Swing sehr ähnlich ist: Ein Auftrag ist ein Objekt einer

Klasse, welche die Schnittstelle Runnable implementiert. Die korrekte Implementierung der Klasse CallbackImpl ist in Listing 16.15 zu finden.

Listing 16.15 Implementierung der AIDL-Schnittstelle Callback

```
package javacomp.android.boundservice;

import android.os.Handler;
import android.os.RemoteException;
import android.widget.TextView;

public class CallbackImpl extends Callback.Stub
{
    private Handler handler;
    private TextView tv;

    public CallbackImpl(Handler handler, TextView tv)
    {
        this.handler = handler;
        this.tv = tv;
    }

    public void newCounterValue(int newValue) throws RemoteException
    {
        Runner r = new Runner(tv, newValue);
        handler.post(r);
    }
}

class Runner implements Runnable
{
    private TextView tv;
    private int newValue;

    public Runner(TextView tv, int newValue)
    {
        this.tv = tv;
        this.newValue = newValue;
    }

    public void run()
    {
        tv.setText("neuer Wert: " + newValue);
    }
}
```

Die folgenden Zeilen zeigen beispielhaft die Benutzung der Methode multiIncrement (counter sei wieder eine Referenz auf den Zähler aus onServiceConnected, tv eine Referenz auf ein TextField). Es wird davon ausgegangen, dasss dieser Code vom Main-Thread ausgeführt wird (wegen der Erzeugung des Handler-Objekts):

```
Handler handler = new Handler();
Callback cb = new CallbackImpl(handler, tv);
counter.multiIncrement(15, cb);
```

Die letzten zwei Beispiele für einen ungebundenen und einen gebundenen Service stellen jeweils nur eine Anwendung dar. Ein wichtiges Thema dieses Buches ist aber die Zusammenarbeit unterschiedlicher Anwendungen. Ich hoffe, dass aufgrund der vorhergehenden Äußerungen klar wurde, dass jede Anwendung ohne Änderung des Programmcodes auf jeweils zwei Anwendungen aufgeteilt werden kann: Die Activity wird jeweils Bestandteil der einen und der Service Bestandteil der anderen Anwendung. Für den Fall des gebundenen Service müssen die AIDL-Dateien sowie die Data-Klasse in beiden Anwendungen vorhanden sein.

Es gäbe bei Android noch viele weitere interessante Dinge zu besprechen. Für einen Einblick in die Prinzipien des Komponentenmodells von Android sollten die besprochenen Sachverhalte allerdings ausreichen.

16.10 Bewertung

Zur Bewertung von Android müssen wir den Begriff Komponente, den wir in diesem Kapitel so weit wie möglich vermieden haben, wieder verwenden. Eine Komponente ist im Kontext dieses Buches eine komplette Android-Anwendung (eine App), die aus mehreren Teilkomponenten oder Bausteinen besteht. Wie durch die Besprechung hoffentlich klar geworden ist, ist ein wesentliches Ziel des Android-Frameworks die einfache Kombinierbarkeit der Bestandteile unterschiedlicher Komponenten. Insofern entspricht das Entwurfsziel voll und ganz der Komponentenidee. Im Folgenden prüfen wir nun, ob auch die Umsetzung von Android zu den Vorgaben für ein Komponentensystem aus Kapitel 7 passt:

- Zu E1: Gemäß E1 soll eine Komponente eine klar identifizierbare Einheit sein, die konform zu einem Komponentenmodell ist. Dies ist für Android eindeutig gegeben, wobei – um es nochmals zu sagen – als Komponente eine Android-Anwendung verstanden wird. Eine solche Anwendung ist in einer APK-Datei verpackt, die eine klar vorgegebene Struktur hat. In der Konfigurationsdatei werden die einzelnen Bestandteile der Komponente wie Activities und Services definiert, die ebenfalls wieder klaren Vorgaben folgen.
- Zu E2: Gerade der Kopplungsmechanismus von Bauteilen unterschiedlicher Komponenten ist eines der herausragenden Merkmale von Android. Da dieser Mechanismus auch zur Kopplung der Bauteile innerhalb einer Anwendung genutzt wird, ist es nahezu gleichgültig, zu welcher Komponente ein benutzter Baustein gehört. Der Intent-Kopplungsmechanismus gehört damit zu den dominierenden Merkmalen von Android. Man kann bei Android ohne Übertreibung von einem modularen System sprechen. Der Kopplungsmechanismus über Intents erinnert an ein Ereignisbussystem, bei dem Nachrichten über einen Bus versendet werden, wobei diese Nachrichten an den zugestellt werden, der den passenden Intent-Filter definiert hat. Die Komponenten sind dadurch nur sehr lose gekoppelt und können zur Laufzeit einfach hinzugefügt, ausgetauscht und entfernt werden.
- Zu E3: Das Android-Framework realisiert zwar keinen Lebenszyklus für eine Komponente im Sinne einer App, aber es gibt ausgeprägte Lebenszyklen für die Teilkomponenten wie Activities und Services. Auch liegt die Erzeugung und das Löschen solcher Objekte voll und ganz in der Verantwortung des Komponenten-Frameworks. Im Gegensatz zu anderen

Frameworks wird ein Objekt einer Einstiegsklasse nicht nur beim Installieren der Komponente erzeugt und beim Deinstallieren gelöscht, sondern in Android gibt es einen ausgeklügelten Mechanismus bezüglich des Erzeugens und Löschens der Bausteinobjekte.

- Zu E4: Eine Angabe, was eine Komponente bereitstellt und benötigt, gibt es in indirekter Form in Gestalt der Rechte. Die Nutzung von Bauteilen anderer Komponenten kann durch Rechte abgesichert werden. Wenn eine Komponente ein bestimmtes Recht braucht, muss dieses in der Konfigurationsdatei AndroidManifest.xml angegeben werden und beim Installieren muss der Benutzer entscheiden, ob die Anwendung dieses Recht erhalten soll oder nicht. Die angeforderten Rechte könnte man als Entsprechung für den Bedarf einer Komponente interpretieren. Umgekehrt können die Rechte, die eine Komponente definiert, um selbst von anderen benutzt zu werden, als das gesehen werden, was die Komponente bereitstellt.

Wenn vielleicht auch die Erfüllung der Eigenschaft E4 nur mit einer gewissen Interpretation möglich war, so kann man doch feststellen, dass Android der Idee eines Komponentensystems voll und ganz entspricht. Da Android ein verhältnismäßig neues System ist, sind also Komponenten offensichtlich noch immer in Mode, wenn auch die Idee von Komponentensystemen schon relativ alt ist.

Literatur

Das Literaturverzeichnis ist nicht alphabetisch, sondern thematisch gegliedert. Zuerst werden Grundlagenbücher zum Thema Java empfohlen. Das sind einerseits Bücher, die das Basiswissen enthalten, das vor dem Lesen dieses Buches schon vorhanden sein sollte. Andererseits geht es auch um Bücher, mit denen man die Java-Grundlagen aus dem ersten Teil dieses Buches vertiefen kann. Danach folgen Bücher zum Thema Komponentenorientierung. Zuletzt folgen Bücher zu den einzelnen Frameworks wie OSGi, EJB und Android, die im dritten Teil dieses Buchs vorgestellt wurden.

Java-Grundlagen

Michael Inden: Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung. 1. Auflage, dpunkt, 2011.

Guido Krüger, Heiko Hansen: Handbuch der Java-Programmierung: Standard Edition 7. 7. Auflage, Addison-Wesley, 2011.

Rainer Oechsle: Parallele und verteilte Anwendungen in Java. 3. Auflage, Hanser, 2011.

Christian Ullenboom: Java ist auch eine Insel: Das umfassende Handbuch. 10. Auflage, Galileo Computing, 2011.

Komponentenorientierte Programmierung

Clemens Szyperski with Dominik Gruntz and Stephan Murer: Component Software – Beyond Object-Oriented Programming. 2nd Edition, Addison-Wesley, 2002.

Andy Ju An Wang, Kai Qian: Component-Oriented Programming. 1st Edition, John Wiley & Sons, 2005.

Java Beans

Robert Englander: Developing Java Beans. 1st Edition, O'Reilly Media, 1997.

OSGi

Alexandre De Castro Alves: OSGi in Depth. 1. Auflage, Manning, 2011.

Richard S. Hall, Karl Pauls, Stuart McCulloch, David Savage: OSGi in Action: Creating Modular Applications in Java. 1. Auflage, Manning, 2011.

Bernd Weber, Patrick Baumgartner, Oliver Braun: OSGi für Praktiker: Prinzipien, Werkzeuge und praktische Anleitungen auf dem Weg zur „kleinen SOA“. 1. Auflage, Hanser, 2010.

Gerd Wütherich, Nils Hartmann, Bernd Kolb, Matthias Lübken: Die OSGi Service Platform – Eine Einführung mit Eclipse Equinox. 1. Auflage, dpunkt, 2008.

Eclipse

Eric Clayberg, Dan Rubel: Eclipse – Building Commercial-Quality Plug-ins. 3rd Edition, Addison-Wesley Longman, 2008.

Jeff McAffer, Jean-Michel Lemieux, Chris Aniszczyk: Eclipse Rich Client Platform – Designing, Coding, and Packaging Java Applications. 2nd Edition, Addison-Wesley Longman, 2008.

Lars Vogel: Eclipse 4 Application Development – The Complete Guide to Eclipse 4 RCP Development. 1. Auflage, Lars Vogel, 2012.

Applets

Elizabeth S. Boese: An Introduction to Programming with Java Applets. 3rd Edition, Jones & Bartlett Publishers, 2009.

Servlets

Jason Hunter: Java Servlet Programming. 2nd Edition, O’Reilly Media, 2001.

Bruce W. Perry: Java Servlet and JSP Cookbook. 1st Edition, O’Reilly Media, 2004.

EJB

Werner Eberling, Jan Leßner: Enterprise JavaBeans 3.1: Das EJB-Praxisbuch für Ein- und Umsteiger. 2. Auflage, Hanser, 2011.

Oliver Ihns, Stefan M. Heldt, Holger Koschek, Joachim Ehm, Carsten Sahling, Roman Schlömmer: EJB 3.1 professionell: Grundlagen- und Expertenwissen zu Enterprise JavaBeans 3.1 – inkl. JPA 2.0. 2. Auflage, dpunkt, 2011.

Andrew Lee Rubinger, Bill Burke: Enterprise JavaBeans 3.1. 6th Edition, O’Reilly Media, 2010.

Spring

Craig Walls: Spring in Action. 3rd Edition, Manning, 2011.

Eberhard Wolff: Spring 3: Framework für die Java-Entwicklung. 3. Auflage, dpunkt, 2010.

Android

Thomas Künneth: Android 4 – Apps entwickeln mit dem Android SDK. 2. Auflage, Galileo Computing, 2012.

Zigurd Mednieks, Laird Dornin, G. Blake Meike, Masumi Nakamura: Programming Android. 1st Edition, O’Reilly, 2011.

Heiko Mosemann, Matthias Kose: Android – Anwendungen für das Handy-Betriebssystem erfolgreich programmieren. 1. Auflage, Hanser, 2009.

Index

L

M

N

O

P

Q

R

S

T

U

V

W

Z

JAVA-KOMPONENTEN //

- Java-Grundlagen, die für die Komponentenprogrammierung wesentlich sind
- Grundprinzipien der Komponentenprogrammierung anhand eines selbstentwickelten Beispiels
- Konkrete Beispiele zu Eclipse, Enterprise Java Beans, Android, Servlets, OSGi
- Entwicklung eigener Komponenten und Frameworks

Immer mehr Softwareentwicklungen bauen heute auf dem Komponentenprinzip auf. Dieses Lehrbuch ermöglicht den Lesern, sich selbstständig in Komponenten-Frameworks einzuarbeiten bzw. eigene Frameworks zu entwickeln. Ziel ist es, eine umfassende Vorstellung darüber zu vermitteln, was Komponenten-Software im Java-Umfeld bedeutet.

Zuerst werden jene Java-Grundlagen, die für die Komponentenprogrammierung essentiell sind, vermittelt. Anhand eines selbstentwickelten Beispiels werden im weiteren Verlauf die Grundprinzipien von Komponentensystemen herausgearbeitet und erklärt. Der dritte Teil erläutert ausgewählte Java-Komponentensysteme. In diesem Zusammenhang stellt das Buch konkrete Beispiele zu Eclipse, Enterprise Java Beans, Android, Servlets sowie OSGi vor.

Am Ende des Buches sind die Leser in der Lage, sowohl Komponenten für die im Buch behandelten Frameworks als auch eigene Komponenten-Frameworks zu entwickeln. Das Lehrbuch richtet sich an Studierende der Informatik und verwandter Studiengänge sowie bereits im Berufsleben stehende Java-Software-Entwicklerinnen und Entwickler.

Prof. Dr. Rainer **OECHSLE** lehrt an der Hochschule Trier am Fachbereich Informatik und vertritt die Fachgebiete Rechnernetze und verteilte Systeme. Auf der Webseite **http://jk.hochschule-trier.de** finden Sie alle Programme des Buches.

AUS DEM INHALT //

- Java-Grundlagen:
  - Generics,
  - Reflection,
  - Annotationen,
  - dynamische Proxies,
  - ClassLoading und
  - Hot Deployment
- Prototypische Implementierung eines Komponentensystems
- Komponenten und Komponentensysteme
- Beispiele für Komponentensysteme: OSGi, Eclipse, Enterprise Java Beans, Android u.a.

UNSER BUCHTIPP FÜR SIE //

Oechsle, Parallele und verteilte Anwendungen in Java
2011. 416 Seiten. FlexCover. € 34,90.
ISBN 978-3-446-42459-3

HANSER

€ 29,99 [D] | € 30,90 [A]
ISBN 978-3-446-43176-8