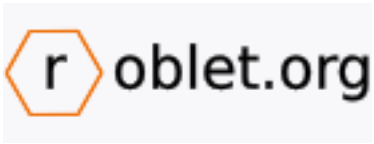

Roblet®

Einführung in die Technik

Version vom

2. August 2010

http://roblet.org/book

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung der Roblet®-Technik begann im Jahre 2001. Der Autor dieses Buches arbeitete daran, eine Steuerung für mobile Roboter-Plattformen des Fraunhofer IPA, Stuttgart, neu zu organisieren. Es sollten Teile der zum Betrieb notwendigen Software über ein Funknetz zur Laufzeit an den jeweiligen Roboter geschickt und dann dort ausgeführt werden können. Die Idee dabei war vom Schreibtisch aus den überwiegenden Teil der Software zur Steuerung des Roboters und dessen Einbindung in das Umfeld entwickeln zu können. Das neuartige Konzept, das in den folgenden Jahren aus dieser Arbeit hervorgegangen ist, ist die *Roblet®-Technik* .

Einige Jahre später zeigte sich, daß diese Technik ein unerwartet breites Anwendungsspektrum besitzt. Mit der Roblet®-Technik läßt sich generell in verteilten Systemen eine deutliche Vereinfachung der Entwicklung, Inbetriebnahme, Wartung und Weiterentwicklung beobachten. Wie dies möglich ist, zeigen die Beschreibungen in diesem Buch.

Die Roblet®-Technik profitiert entscheidend von der Tatsache, daß im ursprünglichen Anwendungsgebiet, der mobilen Robotik, besonders extreme Randbedingungen einzuhalten sind. So ist ein Abreißen der Verbindung zwischen zwei Anwendungen im Netz keine Ausnahmesituation. Oft ist die Verbindung schlecht, weshalb die Bandbreite schwankt oder stark reduziert ist. Weiterhin ist es häufig der Fall, daß nicht so leicht oder im Extremfall gar kein physischer Kontakt zum mobilen Roboter aufgenommen werden kann. Ebenso regelmäßig handelt es sich um heterogene Soft- und Hardwareumgebungen, in die Roboter eingebunden werden. Dann müssen verschiedenste Anwendungen miteinander kommunizieren, wobei die Rechentechnik (Prozessor, Betriebssystem etc.) sich teilweise drastisch unterscheidet.

Dieses Buch gibt dem Leser eine Einführung in die Roblet®-Technik. Dazu werden zunächst ausführlich grundlegende Prinzipien der Technik erläutert. Zusätzlich werden eigene Erfahrungen hinsichtlich der Entwicklung von verteilten Systemen vermittelt, die das Verständnis fördern, wie die Roblet®-Technik gewinnbringend eingesetzt wird. Vorausgesetzt werden grundlegende Kenntnisse in Java™. Die Beispiele setzen zur Ausführung eine Internetverbindung voraus.

Teil I des Buches befaßt sich mit der anwendungsseitigen Entwicklung und ist für Einsteiger und Anwendungsentwickler interessant. Kapitel 2 beschreibt die klassische "Hallo Welt!"-Anwendung, um dem Leser die Leichtigkeit der Roblet®-Technik vor Augen zu führen und außerdem gleich einen lauffähigen Ansatz für die weitere Anwendungsentwicklung zu haben. Dabei wird auf einfache Weise veranschaulicht, um

was es sich bei einem Roblet®, dem Kern der Roblet®-Technik, handelt. Aus dieser Anfangsanwendung werden in Kapitel 3 Beispiele abgeleitet, die schrittweise die Grundlagen erläutern. Kapitel 4 umreißt die Vielfalt an weitergehenden Möglichkeiten, die sich automatisch auf Grund der Implementierung in Java™ bieten. Mit der Idee der Einheiten beschreibt Kapitel 5 den Teil der Technik, der den Zugriff auf verteilte Ressourcen und Funktionalität kontrolliert ermöglicht. Kapitel 6 beleuchtet die Kommunikation zwischen einer Anwendung und den von ihr verteilt ausgeführten Roblets®.

Teil II des Buches beleuchtet die Server-Seite der Roblet®-Technik und ist für Administratoren, Entwickler von Server-Erweiterungen aber z.T. auch Entwickler von Anwendungen gedacht. Kapitel 7 beschreibt, wie ein einfacher Server ohne Module gestartet und betreut wird. Die hier gegebenen Hinweise sind für Administratoren von besonderer Bedeutung. Wie ein Server mit einem oder mehreren Modulen gestartet wird, erläutert Kapitel 8. Hier wird auch die Entwicklung eines Beispiel-Moduls aufgezeigt.

Der Teil III des Buches stellt den Anhang dar. Im Anhang A wird beschrieben, wie die Beispiele dieses Buches gebaut und ausgeführt werden.

1.1 Danksagung

Ohne den Anstoß durch meinen langjährigen Freund und Geschäftspartner Daniel Westhoff wäre dieses Buch nicht entstanden. Manche heute selbstverständliche Teile der Technik, wie z.B. die Möglichkeit, daß Roblets® Werte einfach zurückgeben können oder daß in einem Roblet®-Server mehrere Roblets® gleichzeitig laufen können, lassen sich auf ihn bzw. seine Arbeit mit seinen Studenten an der Universität Hamburg zurückführen. Ihm gilt mein besonderer Dank.

Herzlichen Dank empfinde ich auch für die Mühen, die Daniel Poodratchi investiert hat. Von ihm kommt auch das Logo für roblet®.org. Obwohl in diesem Buch sicher noch einige umständliche Formulierungen vorhanden sind, hat er für die Vereinfachung so manch anderer gesorgt.

Teil I

Anwendungen

2 Hallo Welt!

Als erstes eine kleine Roblet®-Anwendung. Sie soll einen Eindruck von der Leichtigkeit der Roblet®-Technik geben. Darüber hinaus kann Sie herangezogen werden, wenn eine Anwendung um Roblet®-Technik erweitert werden soll.

Die Anwendung gibt auf der im Internet befindlichen Webseite `http://roblet.org/de/technique.sample.html` den klassischen Text "Hello World!" aus. Die hier und da auftretenden neuen Begriffe werden im Rahmen dieses Buches nach und nach erklärt.

Der Quelltext 2.1 zeigt die Java™-Datei der "Hello World!"-Anwendung. Um sie zum Laufen zu bringen, legt man am besten zunächst in einem leeren Verzeichnis eine Datei an. Die Datei muß den Namen **Hello.java** haben.

Hat man die Datei angelegt, muß sie übersetzt und ausgeführt werden. Wie dieses und alle weiteren Beispiele dieses Buches übersetzt und ausgeführt werden, wird in Anhang A beschrieben.

Durch das Ausführen wird die Webseite `http://roblet.org/de/technique.sample.html` um eine Zeile mit dem Inhalt "Hello World!" erweitert. Der Text ist weiter unten auf der Seite innerhalb eines grauen Kastens zu finden. Vor dem Text ist eine Zeitangabe eingefügt, damit man sehen kann, daß man gerade die Hello-Anwendung ausgeführt hat. Hatte man schon ein Browser-Fenster mit der genannten Webseite geöffnet, bevor man das Beispiel laufen ließ, so muß es aktualisiert werden.

Beim Laufenlassen des Beispieles läuft folgendes ab:

- Zunächst wird von der JVM[1] die Methode `main(...)` aufgerufen (Zeile 8).
- Dann wird eine Instanz der Klasse `Hello` erzeugt (Zeile 10). Diese Klasse ist nicht nur die Klasse unserer Anwendung, sondern in diesem Fall auch die sogenannte Roblet®-Klasse, da `org.roblet.Roblet` implementiert wird (Zeile 7).
- Zeile 9 bedeutet, daß der ferne Server, der auf Host *roblet.org* läuft, ein Fach für das zukünftige Roblet® erzeugt und bereithält.
- Durch Aufruf von `run(...)` in Zeile 10 wird die Roblet®-Instanz zu diesem Fach transportiert und in ihm vom Roblet®-Server zur Ausführung gebracht. Für den Transport ist es nötig, daß die Roblet®-Klasse serialisierbar ist, was

[1] Java™ virtual machine - im jeweiligen Betriebssystem meist das Programm **java**.

```
1  import  book.unit.HelloUnit;
2  import  genRob.genControl.client.Client;
3  import  java.io.Serializable;
4  import  org.roblet.Roblet;
5  import  org.roblet.Robot;
6
7  public class  Hello implements Roblet, Serializable {
8      public static void  main (String[] args) throws Exception {
9          new Client (). getServer ("roblet.org"). getSlot ().
10                 run (new Hello ());
11     }
12     public Object  execute (Robot robot) {
13         HelloUnit  hu = (HelloUnit) robot. getUnit (HelloUnit.class);
14         hu. sayHello ();
15         return null;
16     }
17 }
```

Listing 2.1: Die "Hello World!"-Roblet®-Anwendung.

durch die Implementierung der Schnittstelle `java.io.Serializable` in Zeile 7 angezeigt wird.

- Die zum Roblet®-Server auf Host *roblet.org* übertragene Roblet®-Instanz wird dort im reservierten Fach durch Aufruf der Methode `execute(...)` in Zeile 12 zur Ausführung gebracht und damit zum Roblet®.
- Durch das Roblet® wird in der Methode `execute(...)` in Zeile 13 eine Instanz mit Namen `hu` einer Einheit mit Namen `book.unit.HelloUnit` vom Roblet®-Server erbeten.
- Von der Einheiten-Instanz `hu` wird in Zeile 14 die Methode `sayHello(...)` aufgerufen. Dieser Aufruf löst den Eintrag des Textes "Hello World!" in die Webseite `http://roblet.org/de/technique.sample.html` aus. Der Eintrag wird durch den Roblet®-Server auf Host *roblet.org* durchgeführt. Das Browser-Fenster muß aktualisiert werden, falls es mit der genannten Webseite schon geöffnet war, bevor der Server seinen Hinzufügung machte.
- Die Zeile 15 signalisiert lediglich, daß das Roblet® den Wert `null` an die Roblet®-Anwendung zurückgibt. In dem vorliegenden Fall wird dieser Wert von der Roblet®-Anwendung schlicht ignoriert.
- Nach Ende der Ausführung des Roblets® in Zeile 16 wird die Roblet®-Anwendung in Zeile 11 fortgeführt und endet damit.

Bitte beachten Sie, daß die Methode `execute(...)` auf dem Roblet®-Server auf Host *roblet.org* ausgeführt wird. Lokal wird nur eine Instanz der Klasse Hello erzeugt - `execute(...)` wird nicht lokal ausgeführt. Man spricht dann davon, daß `execute(...)` *fern*[2] ausgeführt wird.

Das Beispiel kann als eine der kleinsten Roblet®-Anwendungen aufgefaßt werden. Eine Roblet®-Anwendung ist dabei einfach eine Anwendung, die die Roblet®-Technik einsetzt, also mit Roblet®-Servern kommuniziert und Roblets® fern laufen läßt. Ein Roblet®-Server stellt sicher, daß man in einer bestimmten Weise kommunizieren kann und gleichzeitig die Roblets® eine gewisse Ablaufumgebung vorfinden.

Die Einheit `book.unit.HelloUnit` wird vom Autore dieses Buches definiert und beschrieben. Das Vorhandensein einer Implementierung der genannten Einheit auf einem beliebigen anderen Roblet®-Server ist natürlich nicht zwangsläufig bzw. eher unwahrscheinlich. Es wird im Zusammenhang mit diesem Buch aber sichergestellt, daß sie auf dem Roblet®-Server auf Host *roblet.org* vorhanden ist.

[2] Man spricht in diesem Zusammenhang oft auch von *entfernt*. Jedoch hat den Autor diese Bezeichnung allzuoft an *entfernen*, *wegnehmen*, *löschen* denken lassen, was hier ja gar nicht gemeint ist.

3 Grundlagen

Aus dem im vorangegangenen Kapitel präsentierten Beispiel werden nun weitere grundsätzliche Begriffe und Mechanismen der Roblet®-Technik abgeleitet.

3.1 Die Roblet®-Klasse

In Kapitel 2 wurde schon angedeutet, daß ein Teil einer Roblet®-Anwendung im Netz transportiert und auf einem fernen Roblet®-Server zur Ausführung gebracht wird. Der fern zur Ausführung gebrachte Teil muß in Form einer Klasse vorliegen. Diese Klasse wird zur Verdeutlichung und Unterscheidung *Roblet®-Klasse* genannt.

Eine Roblet®-Klasse ist ein solche, die die Schnittstelle `org.roblet.Roblet`[1] implementiert. Sie kann Gebrauch von (fast) beliebig weiteren Klassen machen.

```
class RobletClass implements org.roblet.Roblet
{
    public Object execute (Robot robot)
    {
        ...
    }
}
...
Roblet roblet = new RobletClass ();
```

Zu bemerken ist, daß die Roblet®-Klasse nicht gleich ein Roblet® ist (vgl. Kapitel 3.7), sondern eben nur die Ausführungsvorschrift eines solchen.

3.1.1 Serialisierbarkeit

In allen Beispielen dieses Buches implementiert die Roblet®-Klasse außerdem noch die Schnittstelle `java.io.Serializable`. Damit ist die Klasse dann *serialisierbar*. Dies geschieht, weil vor der Übertragung von Instanzen einer Klasse die Instanzdaten serialisiert werden müssen. Dabei erfordert der von Java™ dafür bereitgestellte und für die Roblet®-Technik genutzte Mechanismus der Serialisierung die Implementierung der oben genannten Schnittstelle[2].

[1] Vgl. [rdk, `http://roblet.org/rdk/1.2/doc/app/org/roblet/Roblet.html`]

[2] Vgl. [jdk, Object Serialization, `http://java.sun.com/j2se/1.5.0/docs/guide/serialization`]

Werden neben der Instanz der Roblet®-Klasse noch weitere Instanzen mit zum oder auch bei Rückkehr vom Roblet®-Server übertragen, so müssen die zugehörigen Klassen auch serialisierbar sein. Diese kompliziert anmutende Regel geht in der Praxis schnell in Fleisch und Blut über. Im Kapitel 4 wird darauf noch einmal eingegangen.

3.1.2 Statisch verschachtelte Roblet®-Klasse

Zur Verdeutlichung der Idee der Roblet®-Klasse überarbeiten wir die ursprünglichen Anwendung 2.1. Wir trennen als erstes stärker den Teil der Anwendung, der lokal auf unserem Rechner läuft, von dem Teil, der fern im Roblet®-Server auf Host *roblet.org* läuft.

Zur Auftrennung verwenden wir eine neue Klasse `HelloRoblet`, die wir statisch in die Klasse `Hello` verschachteln[3]. `HelloRoblet` ist nun die Roblet®-Klasse und `Hello` ist es nicht mehr. Dazu verlegen wir außerdem aus organisatorischen Gründen unsere Klasse `Hello` vom unbenannten[4] Java™-Paket in ein eigenes benanntes[5] (Java™-Paket `hello1`).

Der Quelltext 3.1 stellt die entstandene Anwendung dar. Er muß in das Unterverzeichnis **hello1** in einer Datei mit dem Namen **Hello.java** abgelegt werden. Die Kompilation und Ausführung ist in Anhang A beschrieben. Die Roblet®-Anwendung modifiziert wieder fern mit Hilfe der Einheit des Roblet®-Server die Webseite `http://roblet.org/de/technique.sample.html` und macht lokal keine Ausgaben.

Zeile 1 von Auflistung 3.1 ist die angekündigte Reorganisation in das benannte Java™-Paket `hello1`. Die Deklaration der Klasse `Hello` in Zeile 9 ist nun vereinfacht, da sie nicht mehr `org.roblet.Roblet` und `java.io.Serializable` implementiert. In Zeile 13 wird nun eine Instanz der Roblet®-Klasse `HelloRoblet` erzeugt. Die Zeilen 16 bis 22 deklarieren diese neue Klasse in statisch verschachtelter Form.

Die Roblet®-Klasse ist in Form einer statisch verschachtelten und nicht als innere[6], d.h. vereinfacht nicht-statischen, Klasse realisiert. Der Grund liegt darin, daß ansonsten für den Transport der Roblet®-Instanz über das Netz vom Java™-Serialisiserungsmechanismus zusätzlich auch die Instanz der umfassenden Klasse mit serialisiert wird. Der Vorgang ist zwar transparent und stört zunächst nicht, da meist nicht viel mehr Daten übertragen werden. Allerdings kann es in gewissen Situation beim Kompilieren oder auch erst zur Laufzeit zu Fehlern kommen, die allein aus Fragen der Serialisierung heraus entstehen und auch durch einen Fortgeschrittenen oft

[3] `HelloRoblet` ist eine *nested class* [jls, §8 Classes], aber keine *inner class* [jls, §8.1.2 Inner Classes and Enclosing Instances].

[4] Unbenannt heißt vereinfacht, daß die Klasse nicht in einem Unterverzeichnis liegt, sondern in dem Verzeichnis, in dem wir kompilieren und ausführen [jls, §7.4.2. Unnamed Packages].

[5] Für uns liegt die Klasse in dem *Unter*verzeichnis **hello1** [jls, §7.4.1. named Packages].

[6] [jls, §8.1.2 Inner Classes and Enclosing Instances]

```
1  package hello1;
2
3  import book.unit.HelloUnit;
4  import genRob.genControl.client.Client;
5  import java.io.Serializable;
6  import org.roblet.Roblet;
7  import org.roblet.Robot;
8
9  public class Hello {
10
11     public static void main(String[] args) throws Exception {
12         new Client().getServer("roblet.org").getSlot().
13                 run(new HelloRoblet());
14     }
15
16     private static class HelloRoblet implements Roblet, Serializable {
17         public Object execute(Robot robot) {
18             HelloUnit hu = (HelloUnit) robot.getUnit(HelloUnit.class);
19             hu.sayHello();
20             return null;
21         }
22     }
23
24 }
```

Listing 3.1: Die "Hallo Welt!"-Roblet®-Anwendung mit einer statisch verschachtelten Roblet®-Klasse.

```
1  package hello2;
2
3  import  book.unit.HelloUnit;
4  import  genRob.genControl.client.Client;
5  import  java.io.Serializable;
6  import  org.roblet.Roblet;
7  import  org.roblet.Robot;
8
9  public class  Hello {
10
11     public static void  main (String[] args) throws Exception {
12         new Client (). getServer ("roblet.org"). getSlot ().
13                 run (new HelloRoblet ());
14     }
15
16 }
17 class  HelloRoblet implements Roblet, Serializable {
18     public Object  execute (Robot robot) {
19         HelloUnit  hu = (HelloUnit) robot. getUnit (HelloUnit.class);
20         hu. sayHello ();
21         return null;
22     }
23 }
```

Listing 3.2: Die "Hallo Welt!"-Roblet®-Anwendung mit einer Roblet®-Klasse der obersten Deklarationsebene in der gleichen Datei.

nicht sofort zu durchschauen sind.

Beachtenswert ist auch das in Zeile 15 von Quelltext 3.1 eingeführte Schlüsselwort[7] `private`. Es gibt an, daß die Roblet®-Klasse `hello1.Hello.HelloRoblet` von außerhalb des Körpers der Klasse `hello1.Hello` nicht zugegriffen werden kann[8]. In unserem Beispiel stellen wir demnach sicher, daß nur die Klasse `hello1.Hello` die Roblet®-Klasse benutzt. Das bringt bei größeren Programmen einen Gewinn für die Lesbarkeit.

3.1.3 Roblet®-Klasse der obersten Deklarationsebene

Um weiter an Übersicht zu gewinnen, verschieben wir die Roblet®-Klasse `HelloRoblet` innerhalb der gleichen Datei auf die oberste Deklarationsebene[9].

Im Quelltext 3.2 ist die entstandene Anwendung dargestellt. Er muß diesmal in

[7] Vgl. [jls, §3.9 Keywords].
[8] Vgl. [jls, §6.6.1 Determining Accessibility].
[9] Vgl. [jls, §7.6 Top Level Type Declarations].

```
1  package hello3;
2
3  import genRob.genControl.client.Client;
4
5  public class Hello {
6
7      public static void main(String[] args) throws Exception {
8          new Client().getServer("roblet.org").getSlot().
9                  run(new HelloRoblet());
10     }
11
12 }
```

Listing 3.3: Die "Hallo Welt!"-Roblet®-Anwendung mit einer Roblet®-Klasse in einer separaten Datei.

das Unterverzeichnis **hello2** in einer Datei mit dem Namen **Hello.java** abgelegt werden. Die Kompilation und Ausführung ist in Anhang A beschrieben. Die Roblet®-Anwendung modifiziert wieder die Webseite `http://roblet.org/de/technique.sample.html` und macht lokal keine Ausgaben.

Die Zeilen 17 bis 23 im Quelltext 3.2 sind im wesentlichen die verschobenen Zeilen 16 bis 22 von Quelltext 3.1. Allein die Klassen-Modifikatoren[10] wurden weggelassen.

3.1.4 Roblet®-Klasse in einer separaten Datei

Eine weitere Aufspaltung geschieht nun durch das Verschieben der Roblet®-Klasse `HelloRoblet` in eine eigene Datei. Das Resultat ist dann die Roblet®-Anwendung von Quelltext 3.3 mit seiner separaten Roblet®-Klasse `HelloRoblet` von Quelltext 3.4.

Die abgebildeten Quelltexte müssen konsequenterweise in das Unterverzeichnis **hello3** in die Dateien **Hello.java** bzw. **HelloRoblet.java**. Die Kompilation und Ausführung ist in Anhang A beschrieben. Die Roblet®-Anwendung modifiziert wieder die Webseite `http://roblet.org/de/technique.sample.html` und macht lokal keine Ausgaben.

Im wesentlichen wurden die Zeilen 17 bis 23 von Quelltext 3.2 aus Quelltext 3.3 weggelassen und zu den Zeilen 8 bis 14 von Quelltext 3.4. Außerdem wurden 4 der Import-Deklarationen[11] mit verschoben.

[10] Vgl. [jls, §8.1.1 Class Modifiers].
[11] Vgl. [jls, §7.5 Import Declarations].

```
1  package hello3;
2
3  import  book.unit.HelloUnit;
4  import  java.io.Serializable;
5  import  org.roblet.Roblet;
6  import  org.roblet.Robot;
7
8  class  HelloRoblet implements Roblet, Serializable {
9      public Object  execute (Robot robot) {
10         HelloUnit  hu = (HelloUnit) robot. getUnit (HelloUnit.class);
11         hu. sayHello ();
12         return null;
13     }
14 }
```

Listing 3.4: Die Roblet®-Klasse zur "Hallo Welt!"-Roblet®-Anwendung mit der Roblet®-Klasse in einer separaten Datei.

3.2 Der Roblet®-Klient

Um mit einem Roblet®-Server zu kommunizieren, wird die Klasse `genRob.genControl.client.Client` bereitgestellt. Eine Instanz dieser Klasse wird *Roblet®-Klient* oder typischerweise einfach *Klient* genannt. Pro Roblet®-Anwendung muß nur ein Klient erzeugt werden. Ein Roblet®-Klient erlaubt es, Roblets® zu einem Roblet®-Server zu schicken und deren dortige Ausführung zu steuern.

Im Quelltext 3.5 wurde zur Verdeutlichung des Klienten eine eigene Variable `client` für dessen Instanz als Zeile 8 eingefügt. Passend wurde auch die (neue) Zeile 9 modifiziert, um diese Variable zu nutzen. Die Roblet®-Klasse ist nicht als Quelltext abgebildet, ist aber die gleiche wie von Quelltext 3.4 mit dem Unterschied, daß in Zeile 1 das Java™-Paket `hello4` stehen muß.

3.3 Der Name des Servers

Der *Name* eines Roblet®-Server ist der DNS[12]-Name oder die IP[13]-Adresse des Wirtsrechners (*host*) auf dem er läuft, erweitert um einen Doppelpunkt und die TCP-Port-Nummer, an dem er lauscht. Lauscht ein Roblet®-Server am Standard-TCP-Port *2001*[14], so ist kann sein Name auch verkürzt ohne Doppelpunkt und Port-Nummer geschrieben werden.

[12] *domain name system*
[13] *internet protocol*
[14] Demgegenüber lauscht z.B. ein Web-Server normalerweise an TCP-Port *80*.

```
1  package hello4;
2
3  import genRob.genControl.client.Client;
4
5  public class Hello {
6
7      public static void main (String[] args) throws Exception {
8          Client client = new Client ();
9          client.getServer ("roblet.org").getSlot ().
10                 run (new HelloRoblet ());
11     }
12
13 }
```

Listing 3.5: Die "Hallo Welt!"-Roblet®-Anwendung mit separater Variable für den Klienten.

Ein Server kann viele Namen haben. Er kann mehrere Namen im DNS haben und lauscht in der Regel auch an mehreren Netzwerkschnittstellen des Wirtsrechners, so daß er dann auch mehrere IP-Adressen hat.

In unserem Beispiel ist der Name des Roblet®-Servers *roblet.org*. Dieser Roblet®-Server läuft auf Standard-TCP-Port *2001*. Man könnte stattdessen deshalb auch *roblet.org:2001* angeben.

3.4 Der Repräsentant des Servers

Damit man in der Roblet®-Anwendung einen Server benutzen kann, kann man sich vom Roblet®-Klienten einen *Server-Repräsentanten* holen. Es handelt sich dabei um eine Instanz vom Typ `genRob.genControl.client.Server`. Wo eine Verwechslung ausgeschlossen ist, werden wir in diesem Buch diese Instanz meist einfach *Server* nennen.

Im Quelltext 3.6 wurde zur Verdeutlichung des Server-Repräsentanten eine eigene Variable `server` in Zeile 10 eingeführt. Diese Variable wird nun in Zeile 11 benutzt. Die Klasse der Variable wurde in Zeile 4 deklariert. Die Roblet®-Klasse ist nicht als Quelltext abgebildet, ist aber die gleiche wie von Quelltext 3.4 mit dem Unterschied, daß in Zeile 1 das Java™-Paket `hello5` stehen muß.

Die in Zeile 10 benutzte Klienten-Methode `getServer(...)` gibt einen Server-Repräsentanten zurück. Zur Erzeugung eines Repräsentanten baut der Klient eine Verbindung zum Server auf. Kennt er den Server schon, so gibt er den schon existierenden Repräsentanten an den Aufrufer zurück. Ist der Server noch nicht bekannt,

```
1  package hello5;
2
3  import  genRob.genControl.client.Client;
4  import  genRob.genControl.client.Server;
5
6  public class  Hello {
7
8      public static void  main (String[] args) throws Exception {
9          Client  client = new Client ();
10         Server  server = client. getServer ("roblet.org");
11         server. getSlot ().
12                 run (new HelloRoblet ());
13     }
14
15 }
```

Listing 3.6: Die "Hallo Welt!"-Roblet®-Anwendung mit separater Variable für den Server-Repräsentanten.

so wird ein Repräsentant erzeugt und zurückgegeben.[15]

Das Holen eines Server-Repräsentanten ist technisch demnach mit einem Netzwerkzugriff verbunden und kann dementsprechend Zeit in Anspruch nehmen. Da die Roblet®-Technik keine *timeout*-Strategie verwendet, kann ein solcher Zugriff theoretisch unendlich lange dauern. Der Roblet®-Klient wird solange versuchen den Server zu kontaktieren, bis er ein Signal bekommt, daß er es nicht mehr tun soll. Dieses Signal würde er dadurch erhalten, daß der die Klienten-Methode aufrufende, "wartende" Thread unterbrochen wird.[16]

Ein Server ist als laufende Instanz eines Programmes definiert. Beendet man ein solches Programm und startet es danach neu - möglicherweise am gleichen Port operierend - so entsteht ein neuer Server und damit für den Fall einer Anfrage am Klienten auch ein neuer Server-Repräsentant!

3.5 Ein Fach eines Servers

Um ein Roblet® in einem Roblet®-Server laufenlassen zu können, benötigt man im Server ein *Fach*. Es ist eine Laufzeitumgebung für Roblets®. Ein Fach könnte als Roblet®-Container aufgefaßt werden.

[15] Zu beachten ist jedoch, daß für den hier nicht weiter behandelten Fall von mehreren Klienten innerhalb einer Anwendung, es auch zu entsprechend vielen Server-Repräsentanten kommen wird, weil die Roblet®-Klienten (gewollt) nichts voneinander wissen.

[16] Hier nicht gezeigt.

```
1  package hello6;
2
3  import  genRob.genControl.client.Client;
4  import  genRob.genControl.client.Server;
5  import  genRob.genControl.client.Slot;
6
7  public class  Hello {
8
9      public static void  main (String[] args) throws Exception {
10         Client  client = new Client ();
11         Server  server = client. getServer ("roblet.org");
12         Slot    slot   = server. getSlot ();
13         slot.
14                 run (new HelloRoblet ());
15     }
16
17 }
```

Listing 3.7: Die "Hallo Welt!"-Roblet®-Anwendung mit separater Variable für ein Fach.

Auf Seiten der Roblet®-Anwendung wird ein Fach durch eine Instanz vom Typ `genRob.genControl.client.Slot` repräsentiert und wie beim Begriff Server spricht man der Einfachheit halber auf Anwendungsseite vom *Fach* und eigentlich nie vom *Fach-Repräsentanten*, obwohl das korrekter wäre.

Im Quelltext 3.7 wurde zur Verdeutlichung des Faches eine eigene Variable `slot` in Zeile 12 eingeführt. Diese Variable wird nun in Zeile 13 benutzt. Die Klasse wurde in Zeile 5 deklariert. Die Roblet®-Klasse ist nicht als Quelltext abgebildet, ist aber die gleiche wie von Quelltext 3.4 mit dem Unterschied, daß in Zeile 1 das Java™-Paket `hello6` stehen muß.

Eine Anwendung kann sich beliebig viele Fächer bereitstellen lassen und verwalten.[17] Pro Fach kann zu einem Zeitpunkt nur ein Roblet® laufen. Ein Fach kann auch *leer*, d.h. ohne laufendes Roblet®, sein. Direkt nach der Bereitstellung ist ein Fach immer leer.

Ein Fach kann beliebig oft wiederverwendet werden. Es kann mehrfach die gleiche Roblet®-Instanz (vgl. 3.6) hineingeschickt werden, aber auch das Schicken von Roblet®-Instanzen verschiedener Klassen ist möglich. Zwischendurch kann ein Fach leer sein - muß aber nicht.

Läuft in einem Fach schon ein Roblet® (vgl. 3.7), so wird dieses durch Schicken einer anderen Instanz, egal welcher Klasse, abrupt beendet. Das Beenden wird vom Roblet®-Server durchgeführt und gibt dem Roblet® keine Möglichkeit, noch weite-

[17] theoretisch - praktisch gibt es natürlich Grenzen durch endliche Ressourcen

re Aktionen durchzuführen. Statt einer Instanz kann auch `null` geschickt werden, wonach das Fach leer ist.

Ausgeführt bzw. beendet werden Roblets® über die Methode `run(...)` des Fach-Repräsentanten. Diese Methode liefert eine Instanz vom Typ `java.lang.Object` oder `null` zurück[18]. Die zurückgegebene Instanz ist genau die, die vom serverseitig laufenden Roblet® zurückgegeben wurde. Sie wurde vom Roblet®-Server serialisiert, zum Klienten übertragen, von diesem deserialisiert und dann als Resultat der Methode `run(...)` zurückgegeben. In unserem Beispiel ist der Rückgabewert `null` und dieser Wert wird anwendungsseitig ignoriert.

Die Tatsache, daß eine Instanz vom Typ `java.lang.Object` zurückgegeben wird, mag zunächst wie eine Einschränkung aussehen - ist es aber nicht. Jeder primitive Typ[19] hat in der Java™-Bibliothek (mindestens) einen zugehörigen Referenztyp[20] und jeder Referenztyp ist von `java.lang.Object` abgeleitet. Dazu gehören auch Felder[21]. Daher sind beliebe Daten als Rückgabewert möglich - nur serialisierbar müssen sie verständlicherweise sein.

3.6 Die Roblet®-Instanz

Die Instanz[22] einer Roblet®-Klasse wird *Roblet®-Instanz* genannt.

Die besondere Beleuchtung des Begriffs ist erfahrungsgemäß deshalb sinnvoll, weil die Übertragung über das Netz ein besonderes Phänomen hervorbringt. Die mehrfache Anwendung von `run(...)` des Fach-Repräsentanten auf die gleiche lokale Instanz hat nämlich zur Folge, daß eben die gleiche Instanz mehrfach verschickt wird. Und trotzdem wird jedesmal eine *neue* Instanz im fernen Fach des Roblet®-Servers erzeugt.

Es handelt sich dabei natürlich nicht wirklich um ein Phänomen, sondern ist eine Entscheidung, die beim Entwurf der Roblet®-Technik getroffen wurde. Der Grund liegt darin, daß in der Praxis die geschickten Instanzen meist selbst ihre Daten ändern und so nach kurzer Zeit nicht mehr in der ursprünglichen Form auf dem Server vorliegen. Auch kann jede lokale Roblet®-Instanz nach dem Verschicken modifiziert werden und ein erneutes Verschicken ändert dann möglicherweise deren Verhalten als Roblet®. Aus eben diesem Grund wird auch zusätzlich eine begriffliche Unterscheidung zwischen Roblet®-Instanz und Roblet® (vgl. Abschnitt 3.7) getroffen.

Im Quelltext 3.8 wurde zunächst zur Verdeutlichung der Roblet®-Instanz eine eigene Variable `roblet` in Zeile 14 eingeführt. Diese Variable wird nun in Zeile 16

[18] Vgl. [jls, §4.5.2 Variables of Reference Type].
[19] Vgl. [jls, §4.2 Primitive Types and Values].
[20] Vgl. [jls, §4.3 Reference Types and Values].
[21] Vgl. [jls, §10 Arrays].
[22] In Java™ ist hierfür eigentlich der Begriff *object* vorgesehen. Allerdings heißt die zentrale Basisklasse auch `Object`, was häufig zu Verwirrungen bei Einsteigern führt.

```
1  package hello7;
2
3  import  genRob.genControl.client.Client;
4  import  genRob.genControl.client.Server;
5  import  genRob.genControl.client.Slot;
6  import  org.roblet.Roblet;
7
8  public class  Hello {
9
10     public static void  main (String[] args) throws Exception {
11         Client  client = new Client ();
12         Server  server = client. getServer ("roblet.org");
13         Slot    slot   = server. getSlot ();
14         Roblet  roblet = new HelloRoblet ();
15         slot.
16                 run (roblet);
17     }
18
19 }
```

Listing 3.8: Die "Hallo Welt!"-Roblet®-Anwendung mit separater Variable für eine Roblet®-Instanz.

benutzt. Die Klasse wurde in Zeile 6 deklariert[23]. Die Roblet®-Klasse ist nicht als Quelltext abgebildet, ist aber die gleiche wie von Quelltext 3.4 mit dem Unterschied, daß in Zeile 1 das Java™-Paket `hello7` stehen muß.

3.7 Das Roblet®

Eine Roblet®-Instanz, die zu einem Roblet®-Server übertragen wurde und dort unter dessen Kontrolle in einem Fach läuft, wird als *Roblet®* bezeichnet.[24]

Das Dasein eines Roblets® ist an die Tatsache geknüpft, daß es in einem Roblet®-Server "läuft" oder "lebt". In der Praxis spricht man deshalb auch häufig vom "laufenden" und seltener auch "lebenden" Roblet®, wobei dann trotzdem einfach nur ein Roblet® gemeint ist.

Wie schon in Abschnitt 3.6 ausgeführt, führt eine mehrfach geschickte, gleiche Roblet®-Instanz trotzdem jedes Mal zu einem neuen Roblet®. Und wie in Kapitel 4 weiter ausgeführt werden wird, ist ein Roblet® auch vergleichbar mit einer laufenden Java™-Anwendung - nur, daß gewisse, eingrenzende Randbedingungen herrschen.

[23] Natürlich hätte auch eine Variable vom Typ `hello7.HelloRoblet` erzeugt werden können.

[24] Technisch gesehen ist es eine Roblet®-Instanz-Kopie (vgl. Kapitel 3.6). Das ist aber eine unpraktische Bezeichnung und für Einsteiger eher verwirrend.

3.8 Der Robot

Die Schnittstelle `org.roblet.Roblet` (vgl. Kapitel 3.1) definiert nur die Methode `execute`. In ihrer Deklaration[25] findet sich nur ein formaler Parameter[26] vom Typ `org.roblet.Robot`. Dieser Parameter wird einfach *Robot* genannt. Er kann als Server-Kontext aufgefaßt werden. Der Robot erlaubt den Zugriff auf die über den Roblet®-Server bereitgestellten Ressourcen.[27]

Welche Ressourcen jeweils bereitgestellt werden, hängt einerseits davon ab, was der Roblet®-Server anbieten kann (verwaltet). Und andererseits auch davon, was er dem jeweiligen Roblet® anbieten will. Trotzdem wird niemals `null` als Robot übergeben[28].

In der Roblet®-Klasse 3.4 ist in Zeile 9 der formale Parameter mit Namen `robot` der Methode `execute` zu sehen. `robot` ist der Robot.

3.9 Einheiten

Die von einem Roblet®-Server bereitgestellten Ressourcen sind aus Sicht eines Roblets® in *Einheiten* aufgeteilt - genauer, in Instanzen vom Typ `org.roblet.Unit`.

Sie werden mit Hilfe der Robot-Methode `getUnit(...)` gewonnen. Übergeben wird dabei der Methode der Typ der betreffenden Einheit (`.class`) und Resultat ist eine Instanz dieses Typs oder `null`. Letzteres wird zurückgegeben, wenn der Server die Einheit nicht zur Verfügung hat oder nicht bereitstellen will.

In Quelltext 3.9 wurde gegenüber Quelltext 3.4 u.a. die Zeile 11 eingefügt. Diese Einfügung dient allein der Verdeutlichung, daß über die Methode zunächst nur eine Instanz vom Typ `org.roblet.Unit` zurückgeben wird. Zeile 12 zeigt dann den "cast"[29], der nötig ist, um auf den richtigen Typ zu wandeln. Die Wandlung ist in der angegebenen Form immer möglich, wenn Parameter-Typ und `cast`-Typ übereinstimmen[30] - hier `book.unit.HelloUnit`.

Der Begriff der Einheit ist dreigeteilt. Zunächst gibt es die Einheiten-*Definition*, die im Beispiel aus der Java™-Schnittstelle (Java™-`interface`) `book.unit.HelloUnit` mit zugehöriger Dokumentation `http://roblet.org/`

[25] [jls, §8.4 Method Declarations]
[26] [jls, §8.4.1 Formal Parameters]
[27] In der Anfangszeit der Roblet®-Technik waren es tatsächlich nur Roboter, auf denen die Roblet®-Server liefen. Und damit bekam ein Roblet® über den *Robot* tatsächlich Zugang zum Roboter - seinen Motoren, Sensoren, Prozessen.
[28] Alles andere wäre ein Fehler in der Implementierung des Roblet®-Server und kann vernachlässigt bzw. mit einer Standard-Fehlerbehandlung auf Seiten der Anwendung versehen werden. Ein solcher Roblet®-Server sollte von der Anwendung schlichtweg ignoriert werden.
[29] [jls, §5.5 Casting Conversion]
[30] Alles andere wäre wieder ein Fehler in der Implementierung des Roblet®-Server und bräuchte nur über eine generelle Standard-Fehlerbehandlung bearbeitet werden.

```
1  package hello8;
2
3  import  book.unit.HelloUnit;
4  import  java.io.Serializable;
5  import  org.roblet.Roblet;
6  import  org.roblet.Robot;
7  import  org.roblet.Unit;
8
9  class  HelloRoblet implements Roblet, Serializable {
10     public Object  execute (Robot robot) {
11         Unit  unit = robot. getUnit (HelloUnit.class);
12         HelloUnit  hu = (HelloUnit) unit;
13         hu. sayHello ();
14         return null;
15     }
16 }
```

Listing 3.9: Die Roblet®-Klasse zur "Hallo Welt!"-Roblet®-Anwendung mit separater Variable `unit`.

`book/api/book/unit/HelloUnit.html` besteht. Dazu wurde eine Einheiten-*Implementierung* in Form einer Java™-Klasse (Java™-`class`) erstellt[31], die dem Roblet®-Server *roblet.org* bekannt gemacht wurde. Und schließlich erzeugt dieser Roblet®-Server *roblet.org* zur Laufzeit auf Anfrage durch das Roblet® per `robot.getUnit(...)` eine Einheiten-*Instanz* der Implementierung bzw. gibt eine vorhandene zurück.

Meist geht aus dem Kontext hervor, ob man eine Definition, Implementierung oder Instanz einer Einheit meint. Man spricht daher meist nur von der "Einheit". Mehr zu diesem Thema ist im Kapitel 5 zu finden.

3.10 Zusammenfassung

Die Roblet®-Technik erlaubt die Erstellung von Java™-Anwendungen, die zur Laufzeit Teile ihrer selbst im Netz verschicken und anderswo zum Laufen bringen. Derartige Anwendungen werden Roblet®-Anwendung genannt. Die Teile, die anderswo laufen, heißen Roblets®. Die Programme, in denen die Roblets® ausgeführt werden, heißen Roblet®-Server.

Zum Versenden, Laufenlassen und Verwalten von Roblets® steht der Roblet®-Klient zur Verfügung. Mit ihm kann Zugriff auf beliebige Roblet®-Server (gleichzeitig) genommen werden, wobei Server-Repräsentanten erzeugt werden. Mit diesen können

[31] Hier nicht gezeigt.

Fächer auf dem jeweiligen Roblet®-Server eingenommen werden. Jedes Fach erlaubt die (parallele) Ausführung eines Roblets®.

In Roblet®-Anwendungen müssen zunächst Roblet®-Klassen vorhanden sein. Von diesen Roblet®-Klassen können Roblet®-Instanzen erzeugt werden. Diese Roblet®-Instanzen können zu Roblet®-Servern geschickt werden und bilden dann dort die Roblets®.

Ein Roblet® benutzt den Robot um an die vom Roblet®-Server verwalteten Ressourcen zu kommen. Diese werden in Form von Einheiten zur Verfügung gestellt.

4 Natürliche Fähigkeiten

Dieses Kapitel wird jeder Leser vermutlich nur einmal durcharbeiten müssen. Denn seine eigentliche Aussage ist, daß ein Roblet® mit leichten Unterschieden ein kleines Java™-Programm darstellt. Daher kann eine Roblet®-Anwendung zusammen mit ihren Roblets® wie *ein* Programm aufgefaßt werden, daß auf *mehreren* Rechnern gleichzeitig läuft. Es wird lediglich dadurch eingeschränkt, daß Ressourcen vom Roblet®-Server verwaltet werden und daher nur durch ihn vermittelt zugegriffen werden.

Verschiedene Java™-Techniken werden im folgenden eingesetzt und so gezeigt, daß dieselben nun auch *jenseits* von Netzwerkgrenzen im Rahmen des jeweiligen Programmes benutzt werden können. Auf diese Weise erweitert sich deshalb objektorientierte Programmierung fast magisch von der lokalen Anwendung auf die verteilte.

Die hier gewonnenen Fertigkeiten lassen sich für einfache wie auch äußerst anspruchsvolle Anwendungen einsetzen. Die Beispiele folgen dem Prinzip eines Baukastensystem.

Der in Java™ erfahrene Leser wird nur wenige wirklich neue Dinge bemerken, sollte aber nicht voreilig den Schluß ziehen, mit diesem Wissen gleich größte Anwendungen erstellen zu können. Erfahrungsgemäß haben es überraschenderweise eher in der Programmierung unerfahrene Leser einfacher, sich die neue Denkweise anzueignen, die mit der Roblet®-Technik einhergeht.

4.1 Die kleinste Roblet®-Klasse

Nicht nur der Vollständigkeit halber soll an dieser Stelle die kleinste Roblet®-Klasse gezeigt werden. Oft beginnt das Verteilen von Anwendungsfunktionalität banaler Weise mit solch einem Typ, denn so lassen sich schon die Mechanismen des Versendens einer Roblet®-Instanz inkl. Fehlerbehandlung testen.

Quelltext 4.1 stellen die Roblet®-Klasse dar. Das einzige, was ein Roblet® dieser Klasse tun wird, ist in Zeile 9 dargestellt - es macht sozusagen nichts und gibt auch nichts zurück.

Korrekterweise muß bemerkt werden, daß natürlich `null` zurückgegeben wird. Auch ein anwendungsseitig aufgerufenes `run(...)` gibt entsprechend dieses `null` zurück. Es wird nur nicht ausgewertet.

```
1  package natural00;
2
3  import java.io.Serializable;
4  import org.roblet.Roblet;
5  import org.roblet.Robot;
6
7  class NaturalRoblet implements Roblet, Serializable {
8      public Object execute (Robot robot) {
9          return null;
10     }
11 }
```

Listing 4.1: Die kleinste Roblet®-Klasse.

4.2 Vordefinierte Typen

Eine Roblet® kann sämtliche Typen einsetzen, die in der Sprache Java™ und der Java™-Bibliothek definiert sind.

Quelltext 4.2 ist eine Roblet®-Klasse, in dem vordefinierte Typen benutzt werden. Gegenüber Quelltext 4.1 sind die Zeilen 9 und 10 hinzugekommen. Zeile 9 definiert einen primitiven Typ[1]. Zeile 10 definiert einen Referenztyp[2].

4.3 Selbstdefinierte Typen

Ein Roblet® kann auch selbstdefinierte Typen einsetzen. Derartige Typen sind dann Klassen[3] oder Schnittstellen[4]. Sie können in beliebigen Java™-Paketen[5] residieren. Es gibt keine Einschränkungen hinsichtlich Schachtelung[6], Lokaldefinition[7] oder Anonymität[8].

Die resultierenden Namen[9] unterliegen den gleichen Zugriffsregeln[10] wie Typen einer normalen Java™-Anwendung. Der *byte-code*[11] der Typen wird, wie die Roblet®-Klassen selbst, vom Roblet®-Klienten bei Bedarf automatisch zum Roblet®-Server

[1] [jls, §4.2 Primitive Types and Values].
[2] [jls, §4.3 Reference Types and Values].
[3] [jls, §8 Classes].
[4] [jls, §9 Interfaces].
[5] [jls, §7 Packages].
[6] Klassen, die nicht *top level* sind nach [jls, §8 Classes].
[7] [jls, §14.3 Local Class Declarations].
[8] [jls, §15.9.5 Anonymous Class Declarations].
[9] [jls, §6 Names].
[10] [jls, §6.6 Access Control].
[11] Bildlich die **.class**-Dateien

```
1  package natural01;
2
3  import java.io.Serializable;
4  import org.roblet.Roblet;
5  import org.roblet.Robot;
6
7  class NaturalRoblet implements Roblet, Serializable {
8      public Object execute (Robot robot) {
9          int i = 42;
10         String s = "42";
11         return null;
12     }
13 }
```

Listing 4.2: Nutzung von vordefinierten Typen in einem Roblet®.

übertragen. Insgesamt bieten sich so grundlegende bekannte Möglichkeiten von Java™ zur Strukturierung einer Software auch auf Seiten eines Roblets®.

Quelltext 4.3 ist eine Roblet®-Klasse, in der der selbstdefinierte Typ `natural02.NaturalRoblet.OwnClass` benutzt wird. Zeilen 12 bis 14 definieren den eigenen Typ. Die Definition als innere Klasse[12] ist nicht zwingend, sondern hier beispielhaft. In Zeile 9 wird die Klasse eingesetzt.

4.4 Ausdrücke, Anweisungen, Blöcke

Auch hinsichtlich der Formulierung von Ausdrücken[13], von Anweisungen und der Strukturierung in Blöcken[14] gibt es keinerlei Einschränkungen.

Quelltext 4.4 ist eine Roblet®-Klasse, in der Ausdrücke, Anweisungen und Blöcke zum Einsatz kommen. Zeilen 9 bis 14 geben hierfür (triviale) Beispiele. Zeile 9 ist ein Ausdruck zur Deklaration der lokalen Variable[15] `i`. Zeile 10 ist eine Ausdrucksanweisung[16], die einen Zuweisungsausdruck[17] umfaßt. Die Zeile 11 bis 14 beschreiben eine `for`-Anweisung[18], die über einen Block[19] iteriert. Dieser Anweisungsblock erstreckt sich von Zeile 12 bis 14.

[12] [jls, §8.1.2 Inner Classes and Enclosing Instances].
[13] [jls, §15 Expressions].
[14] [jls, §14 Blocks and Statements].
[15] [jls, §14.4 Local Variable Declaration Statements].
[16] [jls, §14.8 Expression Statements].
[17] [jls, §15.27 Expression].
[18] [jls, §14.13 The `for` Statement].
[19] [jls, §14.2 Blocks].

```
1  package natural02;
2
3  import java.io.Serializable;
4  import org.roblet.Roblet;
5  import org.roblet.Robot;
6
7  class NaturalRoblet implements Roblet, Serializable {
8      public Object execute (Robot robot) {
9          OwnClass oc = new OwnClass ();
10         return null;
11     }
12     private class OwnClass {
13         public int i = 42;
14     }
15 }
```

Listing 4.3: Nutzung eines selbstdefinierten Typs in einem Roblet®.

```
1  package natural03;
2
3  import java.io.Serializable;
4  import org.roblet.Roblet;
5  import org.roblet.Robot;
6
7  class NaturalRoblet implements Roblet, Serializable {
8      public Object execute (Robot robot) {
9          int i;
10         i = 3;
11         for (int j = 0; j < i; ++j)
12         {
13             // ...
14         }
15         return null;
16     }
17 }
```

Listing 4.4: Ausdrücke, Anweisungen, Blöcke in einem Roblet®.

```
1  package natural04;
2
3  import java.io.Serializable;
4  import org.roblet.Roblet;
5  import org.roblet.Robot;
6
7  class NaturalRoblet implements Roblet, Serializable {
8      public Object execute(Robot robot) throws Exception {
9          Thread t = new SecondThread();
10         t.start();
11         t.join();
12         int i = result;
13         return null;
14     }
15     int result;
16     class SecondThread extends Thread {
17         public void run() {
18             result = 42;
19         }
20     }
21 }
```

Listing 4.5: Nebenläufigkeit in einem Roblet®.

4.5 Nebenläufigkeit (Threads)

Einem Roblet® wird vom Roblet®-Server immer mindestens ein Thread[20] zugeordnet. Dieser eine Thread wird *Hauptthread* genannt. Er ruft die Methode `execute(...)` des Roblets® einmal auf und führt daher die darin enthaltenen Anweisungen aus.

Threads innerhalb eines Roblets® können genausogut jedoch auch weitere Threads starten. Diese laufen dann parallel zu den startenden Threads. Jegliche Inter-Thread-Kommunikation kann genutzt werden.

Endet der Hauptthread, so endet die Roblet®-sendende `run(...)`-Methode aber nicht unbedingt das Roblet®. Das Roblet® endet erst, wenn kein Thread im Roblet®, also auch kein selbstgestarteter, mehr läuft.

Quelltext 4.5 ist eine Roblet®-Klasse, in welcher die Nutzung eines zweiten Threads demonstriert wird.[21] Zeilen 16 bis 20 definieren die Klasse des zweiten Threads. In Zeile 9 wird die Instanz erzeugt und in Zeile 10 wird der zweite Thread zum Leben

[20] [jls, §17 Threads and Locks].

[21] Das Beispiel ist nicht wirklich brauchbar, da es besser in einem Thread funktionieren würde. Aber sinnvollere Beispiele sind leider gleich wieder zu komplex, um sie hier zu präsentieren.

```
1  package natural05;
2
3  import java.io.Serializable;
4  import org.roblet.Roblet;
5  import org.roblet.Robot;
6
7  class NaturalRoblet implements Roblet, Serializable {
8      public Object execute (Robot robot) {
9          try {
10             throw new SomeException ();
11         }
12         catch (SomeException e) {
13             // ...
14         }
15         return null;
16     }
17     class SomeException extends Exception {
18     }
19 }
```

Listing 4.6: Abfangen einer Ausnahme in einem Roblet®.

erweckt.

Zeile 11 und Zeile 18 demonstrieren zwei Formen der Inter-Thread-Kommunikation. In Zeile 11 wartet der erste Thread (Hauptthread) auf das Ende des zweiten Threads. In Zeile 18 weist der zweite Thread der Roblet®-Instanz-Variablen `result` einen Wert zu, der in Zeile 12 vom ersten Thread ausgelesen wird.

4.6 Ausnahmen und ihre Behandlung

Ausnahmen funktionieren wie bei einem normalen Java™-Programm. *Allerdings gibt es einen wesentlichen Unterschied hinsichtlich unbehandelter Ausnahmen.*

Unbehandelte Ausnahmen sind solche, die weder gefangen noch durchgereicht werden. Die Laufzeitumgebung einer Java™-Anwendung ist direkt die JVM und behandelt ein Thread hier eine Ausnahme nicht, so endet nur der Thread und der Stacktrace der Ausnahme wird auf der Konsole der JVM ausgegeben. Die Laufzeitumgebung eines Roblets® ist der Roblet®-Server auf der JVM und im Falle einer Nichtbehandlung einer Ausnahme durch einen beliebigen Thread *beendet der Roblet®-Server das Roblet®* . Läuft in diesem Moment noch der Hauptthread oder ist dieser der Ursprung der Ausnahme, so wird diese Ausnahme zur Anwendung zurückgeschickt und vom Roblet®-Klient in der `run(...)`-Methode geworfen.

Quelltext 4.6 ist ein Roblet®, in dem eine selbstdefinierte Ausnahme geworfen und

```
1  package natural06;
2
3  import  java.io.Serializable;
4  import  org.roblet.Roblet;
5  import  org.roblet.Robot;
6
7  class  NaturalRoblet implements Roblet, Serializable {
8      public Object  execute (Robot robot) throws Exception {
9          throw new SomeException ();
10     }
11     class  SomeException extends Exception {
12     }
13 }
```

Listing 4.7: Durchreichen einer Ausnahme in einem Roblet®.

abgefangen wird. Zeilen 17 und 18 definieren den eigenen Ausnahmetyp. In Zeile 10 wird eine Instanz eines solchen Typs geworfen. Zeilen 12 bis 14 stellen die Ausnahmebehandlung dar.

Quelltext 4.7 ist ein Roblet®, in dem eine selbstdefinierte Ausnahme geworfen, aber nicht abgefangen, sondern durchgereicht wird. In Zeile 9 wird eine Instanz geworfen. Zeile 8 wurde dahingehend erweitert, daß das Weiterreichen per `throws Exception` deklariert wird. Auf Seiten der Anwendung wird diese Ausnahme aus `run(...)` heraus geworfen und kann abgefangen werden.

4.7 Parameter für Roblets®

Einem Roblet® Parameter für die Ausführung mitzugeben, wird über eine passende Initialisierung[22] der Roblet®-Instanz geregelt. Die Roblet®-Klasse muß dazu Instanz-Variable[23] haben, die auch nach Wunsch initialisiert werden müssen. Diese Variable werden automatisch als Teil der Roblet®-Instanz mit zum Roblet®-Server verschickt und stehen dem Roblet® dort zur Verfügung. Es gelten die generellen Regeln der Java™-Serialisierung[24].

Quelltext 4.8 ist ein Roblet®-Anwendung, die einem Roblet® Parameter mitgibt. Die Roblet®-Klasse ist in den Zeilen 14 bis 23 definiert. Zeile 15 deklariert dabei eine Instanz-Variable, welche einen Parameter tragen wird. Die Zeilen 16 bis 18 definieren einen Konstruktur, welcher die Variable initialsiert. In Zeile 20 wurde der Parameter

[22] [jls, §8.8 Constructor Declarations].
[23] [jls, §8.3 Field Declarations].
[24] [jdk, Object Serialization, http://java.sun.com/j2se/1.5.0/docs/guide/serialization]

```
1  package natural07;
2
3  import genRob.genControl.client.Client;
4  import java.io.Serializable;
5  import org.roblet.Roblet;
6  import org.roblet.Robot;
7
8  public class Natural {
9      public static void main(String[] args) throws Exception {
10         Roblet ri = new NaturalRoblet(42);
11         new Client().getServer("roblet.org").getSlot().run(ri);
12     }
13 }
14 class NaturalRoblet implements Roblet, Serializable {
15     int j;
16     NaturalRoblet(int i) {
17         j = i;
18     }
19     public Object execute(Robot robot) {
20         int k = j;
21         return null;
22     }
23 }
```

Listing 4.8: Roblet®-Anwendung mit einem parametrisierten Roblet®.

benutzt. Zeile 10 zeigt, wie eine Roblet®-Instanz mit dem Parameter 42 erzeugt wird.

Dieses einfache Beispiel läßt sich ohne weiteres um weitere Parameter erweitern. Parameter können beliebigen Typs sein, solange diese serialisierbar sind.

4.8 Rückgabewerte eines Roblets®

Der Hauptthread eines Roblets® kann auch Werte zurückgeben. Das geschieht durch Nutzung von `return` in der Roblet®-Methode `execute(...)`. Als Rückgabetyp ist hier `java.lang.Object` definiert. Um eine Übertragung über ein Netz zu ermöglichen, wird als weitere Einschränkung die Serialisierbarkeit[25] gefordert.

Auf Grund dieser Definition kann ein Wert generell ein serialisierbarer Referenztyp[26] oder `null`[27] sein. Primitive Typen[28] können mit Hilfe der "Hüllen"-Klassen[29] in Referenztypen umgewandelt werden. Felder fallen in die Kategorie der serialisierbaren Referenztypen, solange der Basistyp des Feldes ein serialisierbarer Referenztyp oder ein primitiver Typ ist.

Quelltext 4.9 ist eine Roblet®-Anwendung, die ein Roblet® verschickt, welches mit Hilfe einer Instanz einer Hüllenklasse den ganzzahligen Wert 42 zurückgibt. Zeile 20 zeigt, wie der Wert zunächst in die Instanz eines passenden Hüllentyps gewandelt wird. In Zeile 21 wird die Instanz an `return` übergeben. In Zeilen 10 bis 12 wird das Roblet® laufengelassen, wobei der Rückgabewert in der Variable `o` vermerkt wird. Da bekannt ist, daß der Rückgabewert vom Typ `java.lang.Integer` ist, kann in Zeile 13 bedenkenlos gewandelt werden[30]. In Zeile 14 wird der ganzzahlige Wert extrahiert. Zeile 15 gibt den Wert auf dem Terminal aus.

4.9 Zusammenfassung

In diesem Kapitel wurde gezeigt, daß ein Roblet® im wesentlichen ein kleines Java™-Programm darstellt. Die kleinste gezeigte Roblet®-Klasse kann stets als Ausgangspunkt für eigene Entwicklungen benutzt werden.

In einem Roblet® können vordefinierte Typen genauso zum Einsatz kommen, wie selbstdefinierte. Sämtliche Formen von Ausdrücken, Anweisungen und jegliche Strukturierung in Blöcke ist frei möglich.

Programmabläufe können mit Hilfe der bekannten Klasse `java.lang.Thread` parallelisiert werden. Ausnahmen können selbst definiert, geworfen oder abgehandelt

[25] [jdk, Object Serialization, `http://java.sun.com/j2se/1.5.0/docs/guide/serialization`]
[26] [jls, §4.3 Reference Types and Values].
[27] *null type* [jls, §4.1 The Kinds of Types and Values].
[28] [jls, §4.2 Primitive Types and Values].
[29] [jpl, §11.1 Wrapper Classes]
[30] `cast` - vgl. [jls, §5.5 Casting Conversion].

```
1  package natural08;
2
3  import genRob.genControl.client.Client;
4  import java.io.Serializable;
5  import org.roblet.Roblet;
6  import org.roblet.Robot;
7
8  public class Natural {
9      public static void main (String[] args) throws Exception {
10         Object o =
11             new Client (). getServer ("roblet.org"). getSlot ().
12                     run (new NaturalRoblet ());
13         Integer integer = (Integer) o;
14         int ret = integer. intValue ();
15         System.out.println (ret);
16     }
17     private static class NaturalRoblet implements Roblet, Serializable {
18         public Object execute (Robot robot) {
19             int ret = 42;
20             Integer integer = new Integer (ret);
21             return integer;
22         }
23     }
24 }
```

Listing 4.9: Eine Roblet®-Anwendung mit einem Roblet®, welches den Wert 42 zurückgibt.

werden. Auch das Weiterreichen von Ausnahmen ist möglich und in diesem Falle kann auf Seiten der ursprünglichen Anwendung eine Ausnahmebehandlung durchgeführt werden. Anders als in einem normalen Java™-Programm führen unbehandelte Ausnahmen jedoch zum Abbruch eines Roblets®.

Roblets® können mit Parameter in Form von Instanz-Variable versehen werden. Jedes Roblet® kann fast beliebige Daten zurückgeben. Für Parameter und Rückgabewerte muß eine Serialisierbarkeit vorliegen.

5 Einheiten

Die vorangegangenen Kapitel zeigten, wie Abläufe programmiert und verteilt werden können. Dabei werden Daten erzeugt, verschickt und verarbeitet. Daten sind Parameter, dann Ergebnisse und dann ev. wieder Parameter und so fort.

Ein verteiltes System ist jedoch noch mehr als das. Es besteht aus *unterschiedlichen* Ressourcen, die sich oft eben auf *verschiedenen* Rechnern befinden. Solche Ressourcen können Hardware sein, wie z.B. das Fahrgestell eines Roboters, aber auch Software, wie z.B. ein Navigationsalgorithmus oder eine Datenbank. Zum Verständnis der Problematik ist von Bedeutung, daß es dabei grundsätzlich mindestens zwei Blickrichtungen gibt. Einerseits gibt es die Nutzer und andererseits die Anbieter.

Die *Nutzer* möchten eine möglichst umfassend dokumentierte Schnittstelle vorfinden, die Ihnen eine effiziente Nutzung einer Ressource sichert. Desweiteren werden die meisten Rechnersysteme in der Praxis häufig rekonfiguriert, d.h. es kommen Ressourcen hinzu oder es fallen welche weg. Auf solche Änderungen will ein Nutzer zur Laufzeit reagieren können. Er will also herausfinden können, ob ein bestimmter Rechner auch tatsächlich eine bestimmte Ressource bereitstellt oder ob eine hinzugekommen ist.

Nutzer möchten ihre Programme möglichst bei Wechsel des Anbieters der Ressource nicht ändern müssen. Sie wollen außerdem die Möglichkeit haben, ähnliche Ressourcen verschiedener Anbieter in den gleichen Programmen unterstützen zu können. Und schließlich wollen sie auch in der Lage sein, verschiedene Varianten oder Versionen einer Ressource eines Anbieters in den gleichen Programmen nutzen zu können.

Anbieter einer Ressource demgegenüber möchten die Nutzung ermöglichen und stellen deshalb eine Dokumentation und passende Software für den Zugriff zur Verfügung. Die dazu notwenigen Schnittstellendefinitionen werden entweder vom Anbieter selbst vorgenommen oder z.B. von einer unabhängigen Kommission entwickelt. Auch andere Anbieter können dann die so oder so definierten Schnittstellen verwenden.

Anbieter möchten in der Lage sein, in gewissen Grenzen verschiedene Ressource-Versionen oder -Generationen zur Verfügung stellen zu können, ohne daß die Programme der Nutzer dafür jedesmal angepaßt werden müssen. Selten sind die Nutzer von Ressourcen gleich deren Anbieter.

Dieses Kapitel beschreibt nun, wie das Konzept der Einheiten diese Fragestellungen auffängt. Dabei wird zunächst die Idee beschrieben und dann die einzelnen Begriffe derart erläutert, daß ein Nutzer damit arbeiten kann.

Diejenigen, die eine Schnittstelle definieren, werden im folgenden *Gremium* genannt. Auf die tiefere Sicht von Gremien und Anbietern wird im Teil II des Buches genauer eingegangen.

5.1 Idee

Die Roblet®-Technik stellt Mechanismen bereit, mit Hilfe derer sich Anbieter und Nutzer von Ressourcen zusammenfinden können. Das Konzept fußt dabei einfach auf bekannten, grundlegenden Möglichkeiten von Java™. Diese sind die Erweiterung[1] von Schnittstellen[2] (`interface`), die Implementierung[3] von Schnittstellen und die Instanziierung[4] der Implementierungen.

Ein *Gremium*, d.h. dasjenige, das eine Schnittstelle zu einer Ressource *definiert*, erstellt dafür eine oder auch mehrere sog. Einheiten*definitionen* und ev. zusätzliche Hilfsklassen. Dazu kommt noch eine Beschreibung, welche mindestens die Nutzung und bei Bedarf auch Fehler- und Sonderfälle umfaßt. Eine Einheitendefinition erhält man, indem eine Java™-Schnittstelle erstellt und diese die Schnittstelle `org.roblet.Unit` erweitern läßt. Die Schnittstelle wird dann z.B. per **javadoc**[5] dokumentiert und kann an die Nutzer einer Komponente gegeben werden.

Der *Anbieter* einer Ressource liefert eine sog. Einheiten*implementierung* in Form der Java™-Klasse[6] (`class`), die die Java™-Schnittstelle der Einheitendefinition implementiert. Sie können dabei die Eigenheiten ihrer Komponente berücksichtigen und müssen sich nur an die Definition der Schnittstelle halten.[7]

Die *Nutzer* lesen die Dokumentation und wissen nun, was die Einheiten der jeweiligen Ressource bieten, wie sie eingesetzt werden müssen und können - und auch, was im Fehlerfalle passiert etc. Die Java™-Schnittstellen und mögliche Hilfsklassen der Einheitendefinition werden zur Kompilation benötigt und in den Roblets® verwendet. Sie müssen auch mit der Anwendung ausgeliefert werden, da sie zur Laufzeit benötigt werden.

Ein Roblet® erhält auf Anfrage zur Laufzeit vom Roblet®-Server eine sog. Einheiten*instanz*. Diese Instanzen führen letzendlich aus, was vom Roblet® gefordert wird.

---

[1] [jls, §9.1.2 Superinterfaces and Subinterfaces].
[2] [jls, §9 Interfaces].
[3] [jls, §8.1.4 Superinterfaces].
[4] [jls, §12.5 Creation of New Class Instances].
[5] [jdk, Tools and Utilities, `http://java.sun.com/j2se/1.5.0/docs/tooldocs`]
[6] [jls, §8 Classes]
[7] Dazu liefern die Anbieter meist noch die Implementierung eines Moduls, der eine Einbettung in einen Roblet®-Server durch einen Administrator ermöglich (vgl. Module im Teil II des Buches). Mit diesem Modul kann ein Roblet®-Server zur Laufzeit den Roblets® Instanzen von den Klassen des Anbieters zur Verfügung stellen.

Die Unterscheidung in Einheitendefinition, -implementierung und -instanz ist in der Praxis sprachlich nicht von Bedeutung. Der Kontext gibt meist sehr genau an, was gemeint ist. In der Regel spricht man von “der Einheit” und man “holt sich” Einheiten vom Roblet®-Server. Die Begrifflichkeiten wurden hier eingeführt, damit sich das Thema korrekt erläutern läßt.

5.2 Definitionen

Nutzer definieren Einheiten in den meisten Fällen nicht[8]. Stattdessen entwickeln sie Anwendungen mit Roblets®, in denen diese genutzt werden. Ein Nutzer ließt die Dokumentation einer Einheit. Daraus leitet er ab, was er mit ihr machen kann und wie er sie einsetzt.

Um jedoch den Einsatz von Einheiten besser verstehen zu können, wird an dieser Stelle der stark vereinfachte Definitionsprozeß erklärt: Eine Einheit wird definiert, indem eine Java™-Schnittstelle (`interface`) erstellt wird, die `org.roblet.Unit` erweitert und die gewünschten Methoden und Felder enthält. Dazu wird ein Java™-Quelltext erstellt. Dieser Quelltext wird dann kompiliert um den *byte-code*, d.h. die **class**-Datei[9], zu erzeugen. Meist hat man mehrere Einheiten definiert und/oder Hilfsklassen bereitgestellt. Diese packt man zur Vereinfachung für den Nutzer in ein Java™-Archiv, d.h. eine **jar**-Datei. Schließlich erstellt man noch eine Dokumentation, wobei z.B. **javadoc** zum Einsatz kommen kann. Danach kann man die Dokumentation veröffentlichen und das Java™-Archiv herausgeben. Die Nutzer verwenden die Java™-Archive nicht nur zur Kompilation, sondern liefern sie auch mit als Teil der Anwendung aus.

Quelltext 5.1 zeigt die Definition der in diesem Buch schon häufig genutzten Einheit `book.unit.HelloUnit`. Der *byte-code* dieser Einheitendefinition steht im Archiv **book.unit.jar** bereit (vgl. Anhang A.1.2). Die Dokumentation findet sich im Internet unter `http://roblet.org/book/unit/book/unit/HelloUnit.html`.

5.3 Implementierung

Nutzer implementieren normalerweise auch keine Einheiten. Das machen die Anbieter. Nutzer kümmern sich oft auch nicht um die Roblet®-Server, die bestimmte Einheiten für Roblets® anbieten. Das machen Administratoren. Nutzer brauchen nur zu wissen, welches die Roblet®-Server sind, die sie benutzen können.

Der in diesem Buch häufig genutzte Roblet®-Server mit Namen *roblet.org* befindet sich im Internet auf dem Wirtsrechner (*host*) *roblet.org* und wartet dort an Standard-

[8] Natürlich gibt es auch viele Fälle, in denen z.B. Anbieter, Gremium und Nutzer ein und dieselbe Person oder Gruppe sind.

[9] Auch ein `interface` hat eine **class**-Datei nach der Kompilation als Ergebnis.

```
1  package book.unit;
2
3  import  org.roblet.Unit;
4
5  /**
6   * <B>Einfachste Beispieleinheit</B> zur Ausgabe des klassischen
7   * <I>Hello World</I> auf der
8   * <A href="http://roblet.org/de/technique.sample.html" target="_top"
9   *     >Beispielseite</A> von
10  * <A href="http://roblet.org/de" target="_top">roblet®.org</A>.
11  */
12 public interface  HelloUnit extends Unit {
13
14     /**
15      * <B>Ausgabe von <I>Hello World</I></B> auf der
16      * <A href="http://roblet.org/de/technique.sample.html"
17      *     target="_top">Beispielseite</A>
18      * von
19      * <A href="http://roblet.org/de" target="_top">roblet®.org</A>.
20      */
21     public void  sayHello ();
22
23 }
```

Listing 5.1: Die Definition der Einheit `book.unit.HelloUnit`.

TCP-Port-Nummer (*port*) *2001* auf Anwendungen (vgl. Kapitel 3.3). Er enthält die Implementierung der Einheitendefinition aus Kapitel 5.2.

5.4 Instanzen

Nutzer arbeiten stets mit Einheiten*instanzen*. Ein Roblet® erhält vom Roblet®-Server auf Wunsch eine Einheiteninstanz mit Hilfe der Robot[10]-Methode `getUnit(...)`[11]. Zu diesem Zweck gibt es die passende Einheitendefinition an. Auch wenn es sich dabei um ein `interface` handelt, wird es im Quelltext als `.class` dargestellt. Die Instanzen werden vom Roblet®-Server von der zugehörigen Einheitenimplementierung erzeugt. Dabei steht es dem Server frei, bei jeder Anfrage eine neue Instanz zu erzeugen oder beliebige Optimierungen vorzunehmen. Hat ein Roblet® eine Instanz zu einer Definition erhalten, so kann diese über die gesamte Lebenzeit des Roblets® benutzt werden.

Hat der Roblet®-Server zur angegebenen Klasse gar keine Einheitenimplementierung zur Verfügung, so gibt mit Hilfe des Robot `null` zurück. Dieser Fall muß immer bedacht werden, auch wenn auf einem bestimmten Server schon einmal eine passende Instanz zurückgegeben wurde.

Quelltext 5.2 ist eine Roblet®-Anwendung, deren Roblet® prüft, ob der Roblet®-Server, auf dem es läuft, die gewünschte Einheiteninstanz vom Typ `book.unit.HelloUnit` auch zurückgibt. Die Zeile 19 ist dabei die Prüfung und Zeile 20 die Behandlung des Falles, daß keine Instanz zurückgegeben wurde. Dazu wird bei entsprechendem Prüfergebnis eine Ausnahme geworfen, die wegen `throws Exception` von Zeile 17 zum lokalen Anwendungsteil weitergereicht wird. Dort kommt sie bei Zeile 13 heraus und wird aber wegen `throws Exception` von Zeile 11 einfach an die lokale JVM weitergereicht. Diese wird den Stacktrace der Ausnahme ausgeben und die Anwendung beenden.

5.5 Versionierung

Nicht alles läuft bei einer Definition bzw. Implementierung von Ressourcen immer so glatt. Deshalb entstehen Versionen von Definitionen und Implementierungen. Im bisher erläuterten Konzept scheint kein Versionierungsmechanismus enthalten zu sein. Es ist aber doch ein solcher *implizit* vorhanden.

Gesteuert wird die Versionierung durch Gremium, Anbieter und Nutzer jeweils unabhängig nach Bedarf. Dabei sind mehrere Fälle zu unterscheiden:

- Eine Einheitendefinition ist fehlerhaft,

[10] vgl. Kapitel 3.8
[11] vgl. Kapitel 3.9

```
1  package units00;
2
3  import  book.unit.HelloUnit;
4  import  genRob.genControl.client.Client;
5  import  java.io.Serializable;
6  import  org.roblet.Roblet;
7  import  org.roblet.Robot;
8
9  public class  Units {
10
11     public static void  main (String[] args) throws Exception {
12         new Client (). getServer ("roblet.org"). getSlot ().
13             run (new UnitsRoblet ());
14     }
15
16     private static class  UnitsRoblet implements Roblet, Serializable {
17         public Object  execute (Robot robot) throws Exception {
18             HelloUnit  hu = (HelloUnit) robot. getUnit (HelloUnit.class);
19             if (hu == null)
20                 throw new Exception ("HelloUnit missing");
21             hu. sayHello ();
22             return null;
23         }
24     }
25 }
```

Listing 5.2: Eine Roblet®-Anwendung, deren Roblet® den Fall gesondert behandelt, daß die gewünschte Einheitenimplementierung nicht auf dem Roblet®-Server vorhanden ist.

- eine Einheitendefinition ist nicht ausreichend und
- eine Einheitendefinition wird auf absehbare Zeit nicht mehr als Implementierung angeboten werden.

In allen genannten Fällen wird vermutlich eine *Ersatzdefinition* angeboten werden. Diese Ersatzdefinition ist aber nichts weiter als eine Einheitendefinition. Gebräuchlich ist es, den gleichen Klassennamen verändert um eine Nummer zu verwenden. Diese Ersatzdefinition bekommt genauso vom Anbieter eine Implementierung.

Für den Fall, daß die ursprüngliche Definition schlicht *fehlerhaft* ist, wird man sich dafür entscheiden, die dazugehörige Implementierung nicht mehr auf den Roblet®-Servern anzubieten. Die Roblets® können dann dieses fehlerhafte Modell nicht mehr benutzen - es entstehen keine Fehler mehr. Natürlich laufen ältere Anwendungen mit ihren alten Roblets® auch nicht mehr mit den neuen Servern, denn die alten Roblets® erhalten vom Robot `null` auf ihre Anfrage nach einer Einheiteninstanz zurück und müssen dann möglicherweise enden.[12]

War die Definition nur *nicht ausreichend*, so kann man z.B. auf den jeweiligen Roblet®-Servern beide Implementierungen gleichzeitig anbieten. Dadurch, daß die neue Definition einen anderen Namen hat, gibt es keine Probleme von Seiten des Roblet®-Servers. Alte Anwendungen wissen nichts von der neuen Einheit und ihre Roblets® arbeiten einfach wie bisher. Neue Anwendungen können so entwickelt werden, daß entweder ihre Roblets® die alte Definition oder die neue Definition oder auch beide gleichzeitig benutzten.

Der Fall, daß eine Definition *in absehbarer Zeit nicht mehr angeboten wird*, ist ein Spezialfall des vorigen. Hier ist die Lösung der Java™-Mechanismus des Dokumentierens mit Hilfe der **javadoc**-Marke `@deprecated`. Bei der Kompilation wird diese Marke erkannt und entsprechend gemeldet.

Setzt man daher eine bestimmte Einheitendefinition `@deprecated` durch einfaches Hinzufügen zur Dokumentation der Klasse, und verteilt die neu kompilierten Schnittstellen neu, so wissen alle zukünftigen Nutzer davon. Sie sehen es, wenn sie ihre Anwendungen mit den Roblets® rekompilieren und können dann reagieren oder auch die Sache belassen.

Quelltext 5.3 ist eine Roblet®-Anwendung, deren Roblet® die zweite Form der Versionierung einsetzt. In Zeilen 19 und 20 wird eine Einheit vom Typ `book.unit.HelloUnit2` geholt. Ist die Prüfung in Zeile 21, ob die Einheit vorhanden ist, erfolgreich, so wird in Zeile 22 die Methode `say(...)` genutzt, um den Text *Hello Universe!* auf `http://roblet.org/de/technique.sample.html` auszugeben. In diesem Fall endet danach das Roblet®. Im anderen Fall wird eine Ausgabe mit der alten Einheit versucht (Zeilen 25 bis 28).

[12] Allein schon deswegen sollte man in Roblets® den Fall `null` berücksichtigen.

```
1  package units01;
2
3  import  book.unit.HelloUnit;
4  import  book.unit.HelloUnit2;
5  import  genRob.genControl.client.Client;
6  import  java.io.Serializable;
7  import  org.roblet.Roblet;
8  import  org.roblet.Robot;
9
10 public class  Units {
11
12     public static void  main (String[] args) throws Exception {
13         new Client (). getServer ("roblet.org"). getSlot ().
14             run (new UnitsRoblet ());
15     }
16
17     private static class  UnitsRoblet implements Roblet, Serializable {
18         public Object  execute (Robot robot) throws Exception {
19             HelloUnit2  hu2 = (HelloUnit2) robot. getUnit (
20                                                 HelloUnit2.class);
21             if (hu2 != null) {
22                 hu2. say ("Hello Universe!");
23                 return null;
24             }
25             HelloUnit  hu = (HelloUnit) robot. getUnit (HelloUnit.class);
26             if (hu == null)
27                 throw new Exception ("HelloUnit missing");
28             hu. sayHello ();
29             return null;
30         }
31     }
32 }
```

Listing 5.3: Eine Roblet®-Anwendung, deren Roblet® Versionierung nutzt.

5.6 Zusammenfassung

Einheiten ermöglichen einer Roblet®-Anwendung, mit Hilfe eigener Roblets® die durch Roblet®-Server bereitgestellte Ressource zu nutzen. Die nutzbaren Ressourcen werden von einem Gremium in Einheiten aufgeteilt. Einheitendefinitionen enthalten Dokumentation und kompilierte Java™-Schnittstellen. Anbieter liefern Einheitenimplementierungen und halten sich dabei an die Definition. Mit den Implementierungen werden Roblet®-Server ausgestattet.

Ein Nutzer verwendet die Einheitendefinitionen um damit seine Roblet®-Klassen zu schreiben. Im Roblet® wird der Roblet®-Server mit Hilfe des Robot nach Einheitendefinitionen gefragt, woraufhin dieser mit Hilfe der Einheitenimplementierungen Einheiteninstanzen erzeugt und an das Roblet® gibt. Hat der Roblet®-Server keine Einheitenimplementierung zur erfragten Einheitendefinition vorliegen, gibt er `null` zurück.

Durch Bereitstellung neuer Einheiten kann ein Gremium eine Versionierungsmöglichkeit bieten. Die Nutzung alter Einheiten kann verhindert werden. Neue Einheiten können gleichzeitig oder ersetzend bereitgestellt werden. Einheiten können als `@deprecated` gekennzeichnet werden, damit Nutzer bei der nächsten Rekompilation auf eine Veraltung hingewiesen werden.

6 Ferne Instanzen

Erstellt man Anwendungen unter Verwendung der Roblet®-Technik kommt man nach relativ kurzer Zeit in die Situation, daß die Anwendung mit ihren Roblets® kommunizieren soll. Die Anwendung soll also nicht nur einfach Roblets® erzeugen, diese arbeiten lassen und dann die Rückgabewerte auswerten. Stattdessen will man Roblets® befragen, diese während ihres Laufes mit weiteren Informationen versorgen oder diese sollen Fragen stellen oder Informationen schicken können.

Die wohl einfachste Möglichkeit, das zu erreichen, ist die Nutzung der Technik der *fernen Instanzen*. Dabei ist die Idee, daß einer der Kommunikatoren, d.h. Anwendung oder Roblet®, dem anderen die Methoden[1] einer bestimmten Instanz zur Verfügung stellt. Diese eben genannte Instanz wird dann *ferne Instanz*[2] genannte.

Demjenigen Kommunikator, der eine ferne Instanz nutzen möchte, wird die Möglichkeit geboten, auf seiner Seite eine Instanz zu erhalten, deren Methoden in ihrer Definition den Methoden der fernen Instanz identisch sind. Diese Instanz wird dann *vertretende Instanz* genannt. Die Methoden der vertretenden Instanz entsprechen nach Namen, Parametern und Rückgabewerten denen der fernen Instanz, jedoch steckt jeweils hinter ihnen nur der Code für einen Aufruf über das Netzwerk hin zur korrespondierenden Methode der fernen Instanz.

Damit beide Kommunikatoren über Methodennamen, Parameter und Rückgabetypen einig sind, wird eine Schnittstelle (Java™-`interface`) zur Absprache eingesetzt. Diese Schnittstelle ist in der Roblet®-Technik auf natürliche Weise automatisch auf beiden Seiten bekannt.[3] Der Kommunikator, der eine ferne Instanz bereitstellt, implementiert diese Schnittstelle. Der andere benutzt sie, um eine vertretende Instanz zu erhalten.

Die vertretende Instanz wird dabei von einer dynamisch erstellten Klasse erzeugt, die die gemeinsame Schnittstelle implementiert. Aus diesem Grund ist auch, wie in den Beispielen zu sehen, stets eine Wandlung[4] der vertretenden Instanz auf den Typ der Schnittstelle möglich. Dieser Vorgang läuft ohne weiteres Zutun automatisch ab.

---

[1] ... und eben nicht die Attribute - die Instanz wird dabei nicht verschickt

[2] Im englischen wird der Begriff *remote* gebraucht, was ins deutsche dann mit *entfernt* übersetzt wird. Dem Autor hat das den Einstieg stark erschwert, da er immer an *entfernen* gedacht hatte. Aus diesem Grund wird auch hier der Begriff *fern* gebraucht.

[3] Anders als bei RPC, Corba, RMI, Hessian, REST o.ä. muß man sich deshalb keine Gedanken über einen Verteilung machen.

[4] sog. `cast` - vgl. [jls, §5.5 Casting Conversion].

Ruft man also die Methode einer vertretenden Instanz auf, so wird letztendlich die gleiche Methode der fernen Instanz aufgerufen. Die vertretende Instanz fungiert als Stellvertreter für die ferne Instanz und bei normalen Netzwerkbedingungen bemerkt man gar nicht, daß bei solch einem Aufruf eine weitergehende Kommunikation stattfindet. Und genau das macht das Konzept so einfach - eigentlich zur einfachsten Art der Kommunikation überhaupt.

Auch wenn die nachfolgenden Beispiel kompliziert erscheinen, sei der Leser darauf hingewiesen, daß der Aufwand noch für kompliziertere Kommunikation danach kaum noch größer wird. Entscheidend ist nur, daß der unten angedeutete Rahmen einmal vorhanden ist. Man definiert dann einfach im vorhandenen Rahmen weitere Methoden. Pro Anwendung-Roblet®-Kombination ist im wesentlichen nur eine Schnittstelle und all das weitere nötig. Oft läßt sich die Nutzung gar auf weitere Kombinationen ausdehnen.

6.1 Hallo Anwendung!

Zur Illustration des eben beschriebenen nachfolgend eine kleine Anwendung. Sie zeigt eine Kommunikation, bei der die Roblet®-sendende Seite[5] einer Anwendung eine ferne Instanz bereitstellt und das Roblet® diese dann nutzt.

Quelltext 6.1 stellt die Roblet®-Anwendung inklusive Roblet® dar. In den Zeilen 19 bis 21 wird die für beide Seiten wichtige Schnittstelle `AnInterface` definiert. Die Zeilen 22 bis 26 implementieren diese Schnittstelle als Klasse `AnImplementation`.

In Zeile 15 stellt die Anwendung eine Instanz der Implementierung zur nachfolgenden Verwendbarkeit bereit. Das wird die ferne Instanz werden. Zeile 15 zeigt auch, daß auf Roblet®-sendender Seite der Anwendung die Arbeit mit fernen Instanzen *Fach-basiert* ist.

Das Roblet® holt sich in Zeile 30 zunächst die Einheit `genRob.genControl.unit.Proxies`. Ihre Aufgabe ist die Verwaltung von vertretenden Instanzen auf Seiten des Roblets®. Mit Hilfe dieser Einheit wird dann in Zeilen 31 und 32 eine vertretende Instanz `ai` geholt. Der hier nötige `cast` ist ungefährlich, da die Einheit sicherstellt, daß der zurückgegebene Typ dem übergebenen entspricht.

In Zeile 33 ruft das Roblet® nun die Methode `say()` der vertretenden Instanz `ai` auf. Dieser Aufruf wird an die gleichnamige Methode der fernen Instanz vermittelt. Dadurch wird letztendlich auf Roblet®-sendender Seite der Anwendung die Methode `say()` der Implementierung in den Zeilen 23 bis 25 aufgerufen. Es wird **Hello Application!** im Terminal der Roblet®-sendenden Seite der Anwendung ausgegeben.

Das Roblet® endet danach, worauf die Roblet®-sendende Seite der Anwendung in Zeile 17 fortsetzt und daraufhin ebenso endet.

[5] An anderer Stelle würde man von *lokal* sprechen, aber das ist hier für den Einsteiger sicher zu verwirrend.

```
1  package remote00;
2
3  import genRob.genControl.client.Client;
4  import genRob.genControl.client.Slot;
5  import genRob.genControl.unit.Proxies;
6  import java.io.Serializable;
7  import org.roblet.Roblet;
8  import org.roblet.Robot;
9
10 public class Remote {
11
12     public static void main (String[] args) throws Exception {
13         Slot slot = new Client (). getServer ("roblet.org")
14                                                     . getSlot ();
15         slot. offerRemote (new AnImplementation ());
16         slot. run (new ARoblet ());
17     }
18
19     private static interface AnInterface {
20         public void say (String text);
21     }
22     public static class AnImplementation implements AnInterface {
23         public void say (String text) {
24             System.out.println (text);
25         }
26     }
27
28     private static class ARoblet implements Roblet, Serializable {
29         public Object execute (Robot robot) throws Exception {
30             Proxies proxies = (Proxies) robot. getUnit (Proxies.class);
31             AnInterface ai = (AnInterface) proxies. obtain (
32                                                 AnInterface.class);
33             ai. say ("Hello Application!");
34             return null;
35         }
36     }
37 }
```

Listing 6.1: Eine Roblet®-Anwendung inkl. Roblet®, bei der die Anwendung eine ferne Instanz einsetzt.

6.2 Hallo roblet®.org!

Zur Verdeutlichung der Gegenrichtung dient folgendes Beispiel. Es zeigt eine Kommunikation, bei der ein Roblet® eine ferne Instanz bereitstellt, welche von der Roblet®-sendenden Seite der Anwendung genutzt wird.

Quelltext 6.2 stellt die Roblet®-Anwendung inklusive Roblet® dar. In den Zeilen 22 bis 24 wird die für beide Seiten wichtige Schnittstelle `AnInterface` definiert. Die Zeilen 25 bis 31 implementieren diese Schnittstelle als Klasse `AnImplementation`. Diesmal wird eine Einheiteninstanz im Konstruktor in Zeile 26 übergeben.

Das Roblet® holt sich in den Zeilen 35 bis 37 zunächst die Einheiten `book.unit.HelloUnit2` und `genRob.genControl.unit.Remotes`. Die zweite Einheit schafft die Möglichkeit, ferne Instanzen auf Seiten des Roblets® bereitstellen zu können. Mit Hilfe dieser Einheit wird dann in Zeile 38 eine ferne Instanz auf Basis der weiter oben beschriebenen Implementierung bereitgestellt. Die Zeilen 39 bis 43 beschreiben den Start eines Threads, der 10 Sekunden wartet um danach zu enden. Dadurch bleibt das Roblet® mindestens noch weitere 10 Sekunden aktiv und damit auch die ferne Instanz. Währenddessen endet der Hauptthread des Roblets® in Zeile 44. Danach kehrt die Ausführung zur Anwendung nach Zeile 16 zurück.

Während das Roblet® noch 10 Sekunden weiterlebt, läuft die Roblet®-sendende Seite der Anwendung in Zeile 17 weiter. Über eine Methode des Fachs wird nun in den Zeilen 17 und 18 eine vertretende Instanz `ai` geholt. Der hier nötige `cast` ist ungefährlich, da die Methode sicherstellt, daß der zurückgegebene Typ dem übergebenen entspricht.

In Zeile 19 ruft die Roblet®-sendende Seite der Anwendung nun die Methode `say()` der vertretenden Instanz `ai` auf. Dieser Aufruf wird an die gleichnamige Methode der fernen Instanz vermittelt. Dadurch wird letztendlich auf Seiten des Roblets® die Methode `say()` der Implementierung der Schnittstelle in den Zeilen 28 bis 30 aufgerufen. Es wird **Hello Roblet®.org!** auf der im Internet befindlichen Webseite `http://roblet.org/de/technique.sample.html` ausgegeben.

Die Roblet®-sendende Seite der Anwendung endet sofort danach. Das Roblet® nach dem Ende der 10 Sekunden Wartezeit. Damit ist dann die Roblet®-Anwendung beendet.

6.3 Bereitstellungszeitraum

Durch Aufruf der Methode `offerRemote(...)` von `genRob.genControl.client.Slot` bzw. `offer(...)` der Einheit `genRob.genControl.unit.Remotes` wird eine ferne Instanz bereitgestellt. Damit beginnt sozusagen ein Bereitstellungszeitraum.

Eine Bereitstellung kostet interne Ressourcen. In manchen Fällen ist es deshalb

```
1  package remote01;
2
3  import  book.unit.HelloUnit2;
4  import  genRob.genControl.client.Client;
5  import  genRob.genControl.client.Slot;
6  import  genRob.genControl.unit.Remotes;
7  import  java.io.Serializable;
8  import  org.roblet.Roblet;
9  import  org.roblet.Robot;
10
11 public class  Remote {
12
13     public static void  main (String[] args) throws Exception {
14         Slot  slot = new Client (). getServer ("roblet.org")
15                                                     . getSlot ();
16         slot. run (new ARoblet ());
17         AnInterface  ai = (AnInterface) slot. obtainProxy (
18                                                 AnInterface.class);
19         ai. say ("Hello roblet&reg;.org!");
20     }
21
22     private static interface  AnInterface {
23         public void  say (String text);
24     }
25     public static class  AnImplementation implements AnInterface {
26         AnImplementation (HelloUnit2 hu2) { this. hu2 = hu2; }
27         private final HelloUnit2  hu2;
28         public void  say (String text) {
29             hu2. say (text);
30         }
31     }
32
33     private static class  ARoblet implements Roblet, Serializable {
34         public Object  execute (Robot robot) throws Exception {
35             HelloUnit2  hu2 = (HelloUnit2) robot. getUnit (
36                                                 HelloUnit2.class);
37             Remotes  remotes = (Remotes) robot. getUnit (Remotes.class);
38             remotes. offer (new AnImplementation (hu2));
39             new Thread () {
40                 public void  run () {
41                     try { sleep (10000);} catch (Exception e) {}
42                 }
43             }. start ();
44             return null;
45         }
46     }
47 }
```

Listing 6.2: Eine Roblet®-Anwendung inkl. Roblet®, bei der das Roblet® eine ferne Instanz einsetzt.

sinnvoll, eine solche Bereitstellung wieder rückgängig zu machen. Dafür steht auf Roblet®-sendender Anwendungsseite die Methode `revokeRemote(...)` von `genRob.genControl.client.Slot` zur Verfügung. Auf Roblet®-Seite ist ein Aufruf der Methode `revoke(...)` der Einheit `genRob.genControl.unit.Remotes` auszuführen.

Wurde eine Bereitstellung wieder rückgängig gemacht, erzeugt der Aufruf einer Methode einer noch existierenden vertretenden Instanz eine Ausnahme - genau so, als sei die Bereitstellung nie erfolgt.

Endet ein Roblet®, so endet auch eine mögliche Bereitstellung einer fernen Instanz auf Roblet®-Seite. Eine Bereitstellung auf Roblet®-sendender Seite der Anwendung für ein bestimmtes Fach bleibt für dieses Fach jedoch erhalten. Das kann man für nachfolgende Roblets® im gleichen Fach verwenden.

6.4 Netzwerkverhalten

Der Aufruf einer Methode einer vertretenden Instanz wird über das Netzwerk zur korrespondierenden fernen Instanz vermittelt. Netzwerkprobleme machen sich beim Aufruf *nur* durch eine zeitliche Verzögerung bemerkbar. Unterbrochene Verbindungen werden automatisch wieder aufgebaut.[6]

Stattdessen kann der die Methode aufrufende Thread, dessen Methodenaufruf vermutlich nicht mehr zum Erfolg führen wird, von einem anderen Thread jederzeit unterbrochen werden. Der unterbrochene Thread erzeugt daraufhin eine ungeprüfte[7] Ausnahme abgeleitet vom Typ `java.lang.RuntimeException`[8] mit verketteter `java.lang.InterruptedException`[9], sofern er noch wartete. Wartete er nicht mehr, so wird sein Unterbrechungsanzeiger[10] gesetzt.

6.5 Zugriffssteuerung

Die Zugriffssteuerung[11] für Bezeichner[12] in Java™ schränkt die Möglichkeiten für die nötigen Schnittstellen ein, da sich die ferne und die vertretende Instanze in verschiedenen JVM's befinden. Diese Schnittstellen müssen in der Regel als öffentlich[13]

[6] Es wird also nicht mit einem *timeout* gearbeitet.
[7] engl. *unchecked* - vgl. [jls, §11.2 Compile-Time Checking of Exceptions]
[8] [jdk, API Specification, `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/RuntimeException.html`]
[9] [jdk, API Specification, `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/InterruptedException.html`]
[10] engl. *interrupt flag* - vgl. `java.lang.Thread` - [jdk, API Specification, `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html`]
[11] [jls, §6.6 Access Control]
[12] [jls, §6 Names]
[13] [jls, §6.6.4 Example: Access to `public` and Non-`public` Classes]

deklariert werden.

In Quelltext 6.1 und Quelltext 6.2 ist das in Zeile 22 bzw. 25 zu sehen.

6.6 Zusammenfassung

Mit Hilfe der fernen Instanzen läßt sich auf einfache Weise eine Kommunikation zwischen einer Anwendung und seinen Roblets® bereitstellen.

Dazu muß eine Schnittstelle definiert werden, die von einem der Kommunikatoren implementiert wird. Eine Instanz dieser Implementierung wird dann veröffentlicht. Die Instanz heißt dann ferne Instanz.

Der andere Kommunikator holt sich eine sog. vertretende Instanz. Diese ist vom Typ der Schnittstelle. Ein Aufruf einer Methode der vertretenden Instanz führt zu einem Aufruf der korrespondierenden Methode der fernen Instanz.

Netzwerkstörungen machen sich bei einem solchen Aufruf für den Aufrufer außer über eine Zeitverzögerung nicht bemerkbar. Der Thread, der eine Methode einer vertretenden Instanz aufruft, kann jedoch jederzeit unterbrochen werden.

Die Implementierung muß nach den Regeln der Java™-Zugriffsteuerung öffentlich (`public`) sein.

Teil II

Server

7 Einfacher Server

In der Roblet®-Technik wird nicht nur eine anwendungsseitig benötigter Roblet®-Klient bereitgestellt (vgl. Teil I), sondern auch die Gegenseite, d.h. der Roblet®-Server. Da beide Seiten in Java™ implementiert sind und somit überall zur Ausführung gebracht werden kann, ist es keine Einschränkung, daß kein Protokoll zwischen beiden Seiten (öffentlich) definiert wird. Man benötigt auf jeder Seite nur das Roblet®-Development-Kit [rdk].

Weiterhin ist der Server derart konstruiert, daß er für die jeweilig zu verwaltenden Ressourcen passend erweitert werden kann. Diese erweiternden Bibliotheken sind die sog. Module.

Der einfache Server, d.h. ohne Module, verwaltet auch schon Ressourcen. Aber es handelt sich dabei nur um Dinge, die den Server und die von ihm betreuten Elemente betreffen. Zu diesen betreuten Elementen gehören das Logging, die Roblets® u.a.m.

In diesem Kapitel wird zunächst geschildert, wo ein Roblet®-Development-Kit (RDK) heruntergeladen werden kann und was die Annahmen für die weiteren Abschnitte sind. Danach wird gezeigt, wie ein einfacher Server zum Laufen gebracht wird. Mit Hilfe des mit dem RDK mitgelieferten (Server-)Werkzeuges wird der laufende Server getestet. Und schließlich wird erläutert, wie der Servers dazu gebracht wird, Log-Einträge auf der Standard-Ausgabe auszugeben.

7.1 Herunterladen

Der Roblet®-Server ist Teil des Roblet®-Development-Kit. Ein aktuelles Paket mit *Arbeitsdateien* kann von der folgenden Web-Seite heruntergeladen werden:

`http://roblet.org/rdk`

Nach dem Herunterladen muß das Transport-Archiv ausgepackt werden.

7.2 Annahmen

Der Roblet®-Server ist eine Java™-Anwendung ohne grafische Oberfläche. Aus diesem Grund wird im folgenden die Ausführung im Windows-Befehlsinterpreter bzw. einer Unix-Shell beschrieben. Das Fenster, in dem der Befehlsinterpreter bzw. die Unix-Shell läuft, wird im folgenden *Terminal-Fenster* oder kurz *Terminal* genannt

werden.[1]

Zur weiteren Vereinfachung wird in diesem und dem nächsten Kapitel stets vorausgesetzt, daß das jeweilige Arbeitsverzeichnis die ausgepackten Java™-Archive enthält:

```
(Arbeitsverzeichnis)
    | org.roblet.jar
    |  ...
    - ...
    |   - ...
   ...
```

Es wird weiterhin vorausgesetzt, daß eine Java™-Laufzeitumgebung (JRE oder JDK) installiert und zur Vereinfachung die Umgebungsvariable **PATH** derart gesetzt ist, daß die Programme **java** bzw. **java.exe** darüber erreichbar sind.

7.3 Ausführen

Für alles folgende bringen wir den Roblet®-Server mit Standard-Einstellungen zum Laufen. Er läuft dadurch an TCP-Port 2001 und die Sicherheitsmechanismen sind aktiv.

Zur Ausführung eines einfachen Servers muß nun zusammen mit den Annahmen von Abschnitt 7.2 nur folgendes eingegeben werden:

```
java  -jar org.roblet.jar  server
```

Der Server meldet sich mit Rechte- und Versionsangaben. Danach ist er bereit und wartet an Standard-TCP-Port *2001* auf Roblets®. Weiterhin wartet er auf die Eingabe von Befehlen durch den Administrator auf der Eingabezeile des Terminals. Für Roblet®-Anwendung auf dem gleichen Rechner wird dieser Roblet®-Server immer mindestens den Namen *localhost* haben - weitere Namen sind *localhost:2001*, *127.0.0.1* usw.

Um den Server wieder zu beenden, muß auf der Eingabezeile nur ein **e** eingegeben und danach die Eingabetaste (**ENTER**) betätigt werden. Der Server gibt daraufhin noch etwas im Terminal aus, um danach jedoch sofort zu enden. Roblets®, die in dem Moment liefen, werden “unfreundlich” beendet.

7.4 Werkzeug

Mit dem *(Server-)Werkzeug* des Roblet®-Development-Kit (RDK) kann der Server teilweise administriert werden. Die Dokumentation des Server-Teils des RDK gibt

[1] Unter Windows ist das die *Eingabeaufforderung*. Unter X-Windows ist es z.B. ein *X-Terminal*.

aktuelle Auskunft darüber, was mit dem Werkzeug gemacht werden kann. An dieser Stelle soll das Werkzeug genutzt werden, um herauszufinden, ob der von uns ausgeführte Server läuft.

Als Test, ob der Server läuft, holen wir mit Hilfe des Werkzeuges die Versionsnummer des Servers. Die erhalten wir nur, wenn der Server korrekt läuft, da zu diesem Zweck ein Roblet® verschickt wird - das Werkzeug ist einfach eine Roblet®-Anwendung.

Unter den Annahmen von Abschnitt 7.2 ist dann in einem zusätzlichen Terminal zu schreiben:

```
java  -jar org.roblet.jar  tool  localhost  Version  get
```

Als Ergebnis erhält man die Server-Version.

7.5 Logging

Während der Test-Phase einer Anwendung kann es sinnvoll sein, das Logging des Servers zu aktivieren. Der Server läßt hier Einstellungen mit Hilfe der Java™-Eigenschaft **genRob.genControl.log** zu.

Damit der Server z.B. die Logbuch-Eintragungen auf der Kommandozeile ausgibt, muß die o.g. Java™-Eigenschaft mit dem Wert **terminal** belegt werden (in eine Zeile):

```
java  -DgenRob.genControl.log=terminal
        -jar org.roblet.jar  server
```

Macht jetzt z.B. ein Roblet® eines Ausgabe in das Logbuch, so erscheint diese zusätzlich nun auch als Ausgabe im Terminal.

Setzt man die Java™-Eigenschaft **genRob.genControl.log** auf den Wert **terminal,long**, so wird eine Langform der Datums- und Zeitausgabe verwendet. Das ist z.B. interessant, wenn man langlaufende Tests macht (in eine Zeile):

```
java  -DgenRob.genControl.log=terminal,long
        -jar org.roblet.jar  server
```

Zu bemerken ist, daß es sich nur um eine verändert Ausgabe handelt. Die Speicherung der Datums- und Zeitangabe eines der Logbuch-Eintrages ist immer vollständig. Weitere Werte für die o.g. Java™-Eigenschaft, welche umfangreicheres Logging zur Folge haben, kann der Dokumentation des Servers entnommen werden. Generell sollte mit dem Loggen des Servers sparsam umgegangen werden, da sich die allermeisten Fehler in Roblet®-Anwendungen über die entstehenden Ausnahmen und deren Stack klären lassen.

```
1  package server00;
2
3  import  genRob.genControl.client.Client;
4  import  genRob.genControl.unit.log.Logger2;
5  import  java.io.Serializable;
6  import  org.roblet.Roblet;
7  import  org.roblet.Robot;
8
9  public class  ServerUnits {
10
11     public static void  main (String[] args) throws Exception {
12         new Client (). getServer ("localhost"). getSlot ().
13             run (new ServerRoblet ());
14     }
15
16     private static class  ServerRoblet implements Roblet, Serializable {
17         public Object  execute (Robot robot) throws Exception {
18             Logger2  logger = (Logger2) robot. getUnit (Logger2.class);
19             logger. log ("Hello World!");
20             return null;
21         }
22     }
23 }
```

Listing 7.1: Eine Roblet®-Anwendung, deren Roblet® einen Log-Eintrag im Server-Logbuch macht.

7.6 Server-Einheiten

Die vom Server verwalteten Ressourcen können teilweise auch von einem Roblet® aus benutzt werden. Für ein Roblet® sind natürlich Einheiten nötig. Die Einheiten, die ein Server auch ohne Module bereitstellen kann, sind im Roblet®-Development-Kit definiert und dokumentiert. Sie werden *Server-Einheiten* genannt.

Quelltext 7.1 ist eine Roblet®-Anwendung, deren Roblet® einen Log-Eintrag im Logbuch des Roblet®-Server, auf dem es läuft, macht. In Zeile 4 wird die Server-Einheit `genRob.genControl.unit.log.Logger2` importiert, in Zeile 18 vom Roblet® geholt und in Zeile 19 genutzt.

7.7 Server-Name und -Instanzen

Der Server bietet, wie schon benutzt, eine Netzwerkschnittstelle an, um Anwendungen den Zugang zu ermöglichen. Diese Netzwerkschnittstelle wird im unterliegenden Wirtsrechner (engl. *host*) bzw. dessen Betriebssystem an ein *TCP-Port* gebunden.

Zusammen mit dem im Netzwerk verwendeten Namen des Wirtsrechners führt das zum Namen des Servers. Eine Zeichenkette bestehen aus *host* und *port* getrennt durch einen Doppelpunkt, wie z.B. **roblet.org:2001**, wird *Server-Name* genannt. Da 2001 das Standard-TCP-Port ist kann man für den Fall eben auch **roblet.org** beim Werkzeug schreiben.

Jeden auf einem Rechner laufenden Server nennt man *Server-Instanz* zur Unterscheidung vom "toten, nicht laufenden Software-Paket" Roblet®-Development-Kit. Von einem Rechner eines Netzwerkes aus gesehen, kann zu einem jeden Zeitpunkt hinter einem Server-Namen nur ein laufender Server, d.h. Server-Instanz, stehen. Aus Netzwerk-topologischen Gründen sind jedoch gleichzeitig mehrere Namen pro Instanz möglich.

7.8 Roblet®-Development-Kit

Mit Hilfe des Roblet®-Klient können Anwendungen Roblets® in einer Server-Instanz laufenlassen. Aber auch serverseitig kommt ein Roblet®-Klienten zum Einsatz. Trotzdem müssen Roblet®-Anwendung und Roblet®-Server nicht dem gleichen Roblet®-Development-Kit entstammen. Die verschiedenen Roblet®-Development-Kit stellen sicher, daß die Anwendungen und Server miteinander kommunizieren können, auch wenn ihre Versionen verschieden sind.

7.9 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie ein Roblet®-Server ohne Module heruntergeladen und zum Laufen gebracht werden kann und welche Möglichkeiten der Einrichtung und des Tests bestehen. Weiterhin wurden die Server-Einheiten und das Logging angerissen. Mehr dazu ist der Dokumentation der Roblet®-Development-Kit zu entnehmen. Ein Server hat einen Namen und laufende Server werden auch Server-Instanzen genannt. Roblet®-Anwendung und Roblet®-Server müssen nicht mit dem gleichen Roblet®-Development-Kit arbeiten.

Diese Hinweise sind für Server mit Modulen (vgl. nachfolgendes Kapitel 8) auch anwendbar.

8 Server mit Modulen

Ein Server mit Modulen ist prinzipiell ein einfacher Server (Kapitel 7) mit Modulen. Die dort getroffenen Aussagen gelten also auch im Folgenden. Insbesondere was das Herunterladen, das Werkzeug und das Logging betrifft. Der Unterschied zum einfachen Server ist eben nur, daß der Server noch Module zu laden und verwenden hat.

Roblet®-Server bieten den Roblets® Ressourcen in Form von Einheiten an. Mit Hilfe der Module kann man beliebige Ressourcen zur Verfügung stellen. Ein Modul ist dabei eine in Java™ geschriebenen Erweiterung, die einige wenige Bedingungen erfüllen muß. Eine solche Erweiterung ist ansonsten tatsächlich weitgehend frei und kann z.B. auch andere Java™-Bibliotheken, wie Datenbank- oder Netzwerkschnittstellen, nutzen oder über JNI[1] auf Bibliotheken zugreifen, die in C/C++ geschrieben wurden. Auf diese Weise eröffnen sich alle erdenklichen Möglichkeiten.

Nach einem Abschnitt über grundlegende Annahmen für dieses Kapitel, folgt ein Abschnitt, in dem angenommen wird, daß ein Modul bereits vorliegt. Es wird beschrieben, wie dann der Server entsprechend eingerichtet wird. Der nächste Abschnitt beschreibt die Entwicklung eines Moduls für eine beispielhafte Ressource. Diese Ausführungen lassen sich gut als Basis für eigene Entwicklungen nutzen. Im darauffolgenden Abschnitt wird beschrieben, wie bei einem Server mit weiteren Modulen verfahren wird.

8.1 Annahmen

Die Annahmen von Abschnitt 7.2 sollen auch hier gelten.

Dazu kommt noch, daß für die Kompilation auch die Programme **javac** bzw. **javac.exe** über die Umgebungsvariable **PATH** erreichbar sind. Ebenso für die Erzeugung von Java™-Archiven die Programme **jar** bzw. **jar.exe** und für Java™-Dokumentation die Programme **javadoc** bzw. **javadoc.exe**.

8.2 Server mit einem Modul

Bevor tatsächlich auf die Arbeit mit einem Modul eingegangen wird, als erstes noch einmal der Server-Lauf ohne Modul in anderer Schreibweise. Daraus kann dann leicht

[1] Java™ native interface

der Start mit Modul abgeleitet werden.

8.2.1 Ohne Modul in anderer Schreibweise

Nach Abschnitt 7.3 und den Annahmen von Abschnitt 8.1 ist zur Ausführung eines einfachen Servers bei Wahl von Port-Nummer 2010 folgendes anzugeben (in eine Zeile):

```
java  -jar org.roblet.jar  server
```

Das gleiche läßt sich auch folgendermaßen schreiben (in eine Zeile):

```
java  -cp org.roblet.jar  org.roblet.Main  server
```

Die Information für die JVM, daß die Klasse `org.roblet.Main` als Einstieg zu wählen ist, wird nun direkt angegeben, statt die JVM diese Information aus dem Java™-Archiv **org.roblet.jar** zu entnehmen zu lassen. Aus **-jar org.roblet.jar** wurde demnach einfach **-cp org.roblet.jar org.roblet.Main**. Es handelt sich um eine andere Schreibweise für das Starten des Servers, ohne daß sich der laufende Server dann anders verhalten würde.

8.2.2 Mit Modul

Angenommen es liegt nun ein Modul vor, d.h. z.B. der Anbieter X einer Hardware hat auch ein Modul mitgeliefert. Das Modul sei als einfachster Fall in einem Java™-Archiv mit Namen **x.module.jar** untergebracht und die Modul-Klasse (wird später in Abschnitt 8.3.1 erklärt) habe den Namen **x.module.ModuleImpl**. Die Datei liege im gleichen Verzeichnis, wie die Dateien des RDK und es gelten weiterhin die Annahmen von Abschnitt 8.1.

Dann läßt sich der Server mit Modul folgendermaßen starten (in eine Zeile; Unix, Linux, MacOSX):

```
java  -cp org.roblet.jar:x.module.jar
        org.roblet.Main  server
        x.module.ModuleImpl
```

Unter Windows muß statt des Doppelpunktes im Klassenpfad ein Semikolon geschrieben werden (in eine Zeile; Windows):

```
java  -cp org.roblet.jar;x.module.jar
        org.roblet.Main  server
        x.module.ModuleImpl
```

Wie muß man die Zeilen verstehen? Über den Klassenpfad mit den Einträgen **org.roblet.jar** und **x.module.jar** wird der JVM mitgeteilt, wo zur Laufzeit des Programms geforderte Klassen zu suchen sind. Das Programm beginnt seinen Lauf mit der Klasse **org.roblet.Main**, der Start-Klasse des Roblet®-Development-Kit. Die JVM gibt dem Programm die Argumente **server** und **x.module.ModuleImpl** mit. Das Programm interpretiert das erste Argument und weiß, daß es einen Server starten soll. Das zweite Argument und alle weiteren werden daher als Namen von Modul-Klassen interpretiert und diese daher geladen. Die JVM findet die Start-Klasse des Servers, weil **org.roblet.jar** im Klassenpfad ist; das Programm findet die Modul-Klasse, weil **x.module.jar** im Klassenpfad ist.

Obwohl die Unterscheidung nicht ganz sinnvoll ist, kann man als Erläuterung davon sprechen, daß die Start-Klasse schon beim Start geladen wird, während die Modul-Klasse erst zur Laufzeit geladen wird. Diese Tatsache wird durch die Angabe der Modul-Klasse als Parameter beleuchtet.

8.3 Entwicklung eines Modul

8.3.1 Modul-Klasse

Basis für die Möglichkeit, den Roblet®-Server zu erweitern, d.h. Module zu schreiben, ist eine Anzahl von Java™-Schnittstellen (`interface`). Unter diesen ist die Schnittstelle `genRob.genControl.modules.Module` von zentraler Bedeutung. Ein Modul muß sie implementieren, um als solches vom Roblet®-Server erkannt zu werden. Eine Implementierung wird *Modul-Klasse* genannt. Der Name dieser Klasse muß beim Start des Roblet®-Servers als Argument in der oben beschriebenen Weise angegeben werden.

8.3.2 Modul ohne Einheiten

Bei der Entwicklung eines Moduls beginnt man am besten mit dem einfachsten Fall, d.h. ohne Einheiten, und bringt das erst einmal zum Laufen.

Quelltext 8.1 zeigt die einfachst mögliche *Modul-Klasse*. Sie besteht im wesentlichen aus den 4 Methoden `moduleInit(...)`, `moduleDone()`, `getUnit4Slot(...)` und `resetUnit4Slot(...)`.

Die Methoden `getUnit(...)` und `getRegistry()` müssen nur in der angegebenen Form beigefügt werden. Sie werden von Servern der neuesten Generation nicht mehr verwendet.

Bevor hier mit dieser Modul-Klasse weitergearbeitet wird, ist es sinnvoll, noch das Logging des Servers zu nutzen, um die spätere Aktivität beobachten zu können.

Quelltext 8.2 zeigt die einfachst mögliche Modul-Klasse - nun aber um Logging erweitert. Dazu wird bei der Initialisierung des Moduls als erstes eine Instanz vom

```java
package module00;

import  org.roblet.Unit;
import  genRob.genControl.modules.Module;
import  genRob.genControl.modules.Registry;
import  genRob.genControl.modules.Slot;
import  genRob.genControl.modules.Supervisor;
import  genRob.genControl.modules.Use;

public class  ModuleImpl implements Module {

    public void  moduleInit (Supervisor sv, Use use) throws Exception {
    }

    public void  moduleDone () {
    }

    public Unit  getUnit4Slot (Class clazz, Use use, Slot slot) {
        return null;    // No implementation found
    }

    public boolean  resetUnit4Slot (Unit unit) {
        return false;   // Not found
    }

    // not used anymore by the server
    public Unit  getUnit (Class clazz)  { return null; }
    public Registry  getRegistry ()     { return null; }

}
```

Listing 8.1: Ein Modul ohne Einheiten.

```java
package module01;

import  org.roblet.Unit;
import  genRob.genControl.modules.Log;
import  genRob.genControl.modules.Module;
import  genRob.genControl.modules.Registry;
import  genRob.genControl.modules.Slot;
import  genRob.genControl.modules.Supervisor;
import  genRob.genControl.modules.Use;

public class  ModuleImpl implements Module {

    public void  moduleInit (Supervisor sv, Use use) throws Exception {
        log = sv. getLog ();
        log. out (null, null, "init");
    }
    private Log  log;

    public void  moduleDone () {
        log. out (null, null, "done");
    }

    public Unit  getUnit4Slot (Class clazz, Use use, Slot slot) {
        return null;    // No implementation found
    }

    public boolean  resetUnit4Slot (Unit unit) {
        return false;   // Not found
    }

    // not used anymore by the server
    public Unit  getUnit (Class clazz)  { return null; }
    public Registry  getRegistry ()     { return null; }

}
```

Listing 8.2: Ein Modul ohne Einheiten, aber mit Logging.

Typ `genRob.genControl.modules.Log` vermerkt. Die Instanz erhält man von der `genRob.genControl.modules.Supervisor`-Instanz, einer Art Kontext des Servers für ein Modul, die bei der Initialisierung mit übergeben wird. Für die weitere Verwendung während der "Lebenszeit" des Moduls wird in Zeile 14 die `Log`-Instanz vermerkt. In den Zeilen 15 und 20 werden Einträge ins Logbuch des Servers erzeugt.

Das Paketverzeichnis (**module01**) vom Beispiel sei ein direktes Unterverzeichnis, dann erfolgt eine Kompilation unter den Annahmen von Abschnitt 8.1 nun folgendermaßen (Unix, Linux und MacOSX):

```
javac  -cp org.roblet.jar  module01/ModuleImpl.java
```

Bzw. (Windows):

```
javac  -cp org.roblet.jar  module01\ModuleImpl.java
```

Um die entstandene Modul-Klasse in ein Java™-Archiv einzubetten, kann nun folgendes ausgeführt werden (Unix, Linux und MacOSX):

```
jar  cf module01.jar  module01/ModuleImpl.class
```

Bzw. (Windows):

```
jar  cf module01.jar  module01\ModuleImpl.class
```

Einen Roblet®-Server mit dem Modul und Logging-Ausgaben in das Terminal bringt man nun folgendermaßen zum laufen (in eine Zeile; Unix, Linux, MacOSX):

```
java  -DgenRob.genControl.log=terminal
        -cp  org.roblet.jar:module01.jar
        org.roblet.Main  server
        module01.ModuleImpl
```

Bzw. (in eine Zeile; Windows):

```
java  -DgenRob.genControl.log=terminal
        -classpath  org.roblet.jar;module01.jar
        org.roblet.Main  server
        module01.ModuleImpl
```

Wenn der Server läuft, so werden im Terminal Ausgaben erscheinen. Man beendet den Server dann durch die Eingabe von **e** am Zeilenanfang und nachfolgendem Betätigen der Eingabe-Taste **ENTER**. Nach dem Start erscheint die Ausgabe **init** vom Modul und nach der Aufforderung zum Beenden erscheint **done**.

```
1  package module02;
2
3  import  org.roblet.Unit;
4
5  /** Einheiten, welche Textnachrichten speichern und abrufen läßt. */
6  public interface MessagesUnit extends Unit {
7
8      /** Speichert eine Nachricht.
9       * @param message Nachricht
10      * @throws MessagesException  wenn <TT>null</TT> übergeben wurde
11      */
12     public void  store (String message) throws MessagesException;
13
14     /** Holt die älteste Nachricht ab und löscht sie danach aus der
15      * Speicherung.
16      * @return Nachricht oder <TT>null</TT>, wenn keine (weitere) vorliegt
17      */
18     public String  retrieve ();
19 }
```

Listing 8.3: Java™-Schnittstelle einer Einheiten zur Speicherung und Abholung von Nachrichten.

8.3.3 Einheiten-Definition

Ein Modul hat normalwerweise nur dann einen Sinn, wenn eine Ressource bereitgestellt wird. Eine Ressource wird Roblets® über Einheiten verfügbar gemacht. Der Entwickler eines Roblets® greift dazu auf die jeweiligen Einheiten-Definitionen zurück. Eine solche Einheiten-Definition wird nachfolgende beispielhaft entwickelt.

Als Beispiel-Ressource sei die Möglichkeit der Speicherung und des Abrufens von Nachrichten genommen. Das Ziel ist demnach, ein Modul bzw. an dieser Stelle erst einmal nur die Einheiten-Definition zu erstellen, welche die Speicherung und das Abrufen von Nachrichten ermöglichen.

Eine Einheiten-Definition besteht aus drei Teilen: erstens der Erstellung einer Java™-Schnittstelle, welche `org.roblet.Unit` erweitert, und ihrer Hilfsklassen, zweitens der Erzeugung eines Java™-Archivs für die entstandenen Klassen und drittens der Erzeugung von Java™-Dokumentation für die Java™-Schnittstelle der Einheit. Die letzten beiden Teile werden dann an Entwickler von Roblets® gegeben.

Quelltext 8.3 zeigt die beispielhafte Java™-Schnittstelle. Für manche ungewohnt ist die Erweiterung der Schnittstelle `org.roblet.Unit` in Zeile 6 mit Hilfe des Java™-Schlüsselwortes `extends`. Prinzipiell handelt es sich aber um eine normale Java™-Schnittstelle.

```java
package module02;

/**
 * Wird erzeugt, wenn in {@link MessagesUnit} eine <TT>null</TT>-Nachricht
 * übergeben wird.
 */
public class MessagesException extends Exception {

}
```

Listing 8.4: Java™-Klasse eines Ausnahmetyps zur Verwendung im Zusammenhang mit der Nutzung der Einheit.

Quelltext 8.4 zeigt als Hilfsklasse zu der Einheiten-Schnittstelle einen eigenen Ausnahmetyp.

Das Paketverzeichnis (**module02**) vom Beispiel sei ein direktes Unterverzeichnis, dann erfolgt eine Kompilation unter den Annahmen von Abschnitt 8.1 nun folgendermaßen (in eine Zeile; Unix, Linux und MacOSX):

```
javac  -cp org.roblet.jar
        module02/MessagesUnit.java
        module02/MessagesException.java
```

Bzw. (in eine Zeile; Windows):

```
javac  -cp org.roblet.jar
        module02\MessagesUnit.java
        module02\MessagesException.java
```

Daraus wird das Java™-Archiv **module02-units.jar** für Entwickler von Roblets® folgendermaßen erstellt (in eine Zeile; Unix, Linux und MacOSX):

```
jar  cf module02-units.jar
        module02/MessagesUnit.class
        module02/MessagesException.class
```

Bzw. (in eine Zeile; Windows):

```
jar  cf module02-units.jar
        module02\MessagesUnit.class
        module02\MessagesException.class
```

Der Einsteiger sei noch darauf hingewiesen, daß bei der Kompilation einer **java**-Datei auch mehr als nur eine **class**-Datei entstehen kann. Das muß bei der Erstellung eines Java™-Archiv immer berücksichtigt werden.

Die Java™-Dokumentation für Entwickler von Roblets® entsteht im Unterverzeichnis **module02-units-javadoc** durch (in eine Zeile; Unix, Linux und MacOSX):

```
javadoc  -classpath org.roblet.jar
        -d module02-units-javadoc
        module02/MessagesUnit.java
        module02/MessagesException.java
```

Bzw. (in eine Zeile; Windows):

```
javadoc  -classpath org.roblet.jar
        -d module02-units-javadoc
        module02\MessagesUnit.java
        module02\MessagesException.java
```

Die Java™-Dokumentation berücksichtigt die Kommentare in den Java™-Dateien **module02/MessagesUnit.java** und **module02/MessagesException.java**. Die Einstiegsdatei der Dokumentation ist **module02-units-javadoc/index.html** und kann mit einem beliebigen Browser geöffnet werden.

Will man nun noch eine ZIP-Datei **module02-units.zip** erstellen, die das Java™-Archiv und die Java™-Dokumentation enthält und sich leicht an Entwickler von Roblets® schicken läßt, so erreicht man das nun einfach durch (in eine Zeile; alle Betriebssysteme):

```
jar  cMf module02-units.zip
        module02-units.jar  module02-units-javadoc
```

Der Entwickler der Roblets® verwendet die Java™-Dokumentation, um zu verstehen, was die im Java™-Archiv enthaltenen Einheiten-Schnittstellen und mögliche weitere Klassen bieten - im obigen Fall nur eine Einheiten-Schnittstelle. Das Java™-Archiv wird zur Kompilation und während des Laufes benötigt und muß an den Nutzer der entstandenen Anwendung mit ausgeliefert werden.

8.3.4 Einheiten-Implementierung

Der nächste Schritt bei der Entwicklung des Beispiel-Moduls, ist die *Implementierung* der Einheit. Das schließt in unserem Fall noch die Betreuung der Ressource mit ein. Die Ressource ist im Beispiel ein Nachrichtenverwalter (und nicht die Nachrichten oder der Speicher der Nachrichten). Das Logging wird auch gleich berücksichtigt, damit wieder beobachtet werden kann, was zur Laufzeit passiert. Liegt die Einheiten-Implementierung vor, so kann auch die Modul-Implementierung ergänzt werden.

8.3.4.1 Nachrichtenverwalter

Quelltext 8.5 ist die Klasse des Nachrichtenverwalters. Im Konstruktor wird eine Instanz vom Logbuch vermerkt, um später Informationen schreiben zu können. Es wird auch eine Liste erzeugt, in der später Nachrichten gespeichert werden.

Beachtenswert ist die Verwendung des Schlüsselwortes `synchronized` für die Methoden `store(...)` und `retrieve()`. Sie stellt sicher, daß Schreiben und Lesen in die bzw. von der Liste nicht von zwei Threads gleichzeitig geschehen kann. Das ist unabhängig davon, ob es sich um Threads aus dem gleichen oder zwei verschiedenen Roblets® handelt. Beim Bereitstellen von Ressourcen muß stets von solchen Parallelitäten ausgegangen werden.

8.3.4.2 Einheiten-Implementierung

Quelltext 8.6 ist die Implementierung der Einheit. Dem Konstruktor wird eine Instanz vom Typ `genRob.genControl.modules.Use`, eines Nutzungszählers, und der Nachrichtenverwalter übergeben. Der Nachrichtenverwalter ist sicher nicht überraschend. Die Einheit hat ja zum Ziel, die Verwaltung von Nachrichten zu ermöglichen.

Der Nutzungszähler hingegen wird benötigt, um dem Roblet®-Server Hilfestellung zu geben. Sein Einsatz ermöglicht ihm, mitzuzählen, wieviele Threads eines Roblets® noch innerhalb von Modulen sind.

Hintergrund ist eine Problemstellung, die auftritt, wenn ein Roblet® von außen beendet werden muß, ohne daß es etwas dagegen unternehmen kann. In diesem Fall muß der Roblet®-Server sicherstellen, daß kein Thread des Roblets® sich noch innerhalb irgendeines Moduls befindet. Zu diesem Zweck stellt der Roblet®-Server zu jedem Roblet® genau einen Nutzungszähler zur Verfügung. Er übergibt diesen zu einem bestimmten Zeitpunkt an die Modul-Implementierung, die ihn dann bei Erzeugung einer Instanz der Einheiten-Implementierung mit angibt.

Das bedeutet natürlich auch, daß pro Roblet® mindestens eine Instanz der Einheiten-Implementierung erzeugt werden muß. Demgegenüber ist unsere Nachrichtenverwaltung nur in einer Instanz vorhanden.

Weiterhin sollte die Verweilzeit eines Threads eines Roblets® im Modul möglichst kurz sein. Zur Not ist vom Modul vor Eintritt in das Modul ein eigener Thread zu starten, der längere Arbeiten vornimmt. In diesem Fall muß der Thread des Roblets® auf die Abarbeitung warten - das Roblet® bzw. dieser Thread kann aber trotzdem jederzeit beendet werden. Ein Modul wird über das Ende eines Roblets® informiert und kann seinerseits dann weitere Aktivitäten für das Roblet® einstellen.

Der Nutzungszähler ist stets so einzusetzen, wie im gegebenen Beispiel der durch das Attribut `use` dargestellte:

```
use. raise ();
try {
```

```java
package module02;

import genRob.genControl.modules.Log;
import java.util.LinkedList;

public class MessagesManager {

    MessagesManager (Log log) {
        this. log = log;
        list = new LinkedList ();
    }
    private final Log  log;
    private final LinkedList  list;

    public synchronized void  store (String message)
            throws MessagesException {
        log. out (null, null, "store(" + message + ")");
        if (message == null)
            throw new MessagesException ();
        list. addLast (message);
    }

    public synchronized String  retrieve () {
        String  message;
        if (list. isEmpty ())
            message = null;
        else
            message = (String) list. removeFirst ();
        log. out (null, null, "retrieve() = " + message);
        return message;
    }
}
```

Listing 8.5: Java™-Klasse zur Verwaltung von Nachrichten.

```
1  package module02;
2
3  import  genRob.genControl.modules.Use;
4
5  public class MessagesUnitImpl implements MessagesUnit {
6
7      MessagesUnitImpl (Use use, MessagesManager messages) {
8          this. use = use;
9          this. messages = messages;
10     }
11     private final Use  use;
12     private final MessagesManager  messages;
13
14     public void  store (String message) throws MessagesException {
15         use. raise ();
16         try {
17             messages. store (message);
18         } finally {
19             use. lower ();
20         }
21     }
22
23     public String  retrieve () {
24         use. raise ();
25         try {
26             return messages. retrieve ();
27         } finally {
28             use. lower ();
29         }
30     }
31 }
```

Listing 8.6: Java™-Klasse eines Ausnahmetyps zur Verwendung im Zusammenhang mit der Nutzung der Einheit.

```
    // ... hier die Arbeit mit der Ressource
} finally {
    use. lower ();
}
```

Dadurch wird sichergestellt, daß erstens `raise()` stets genau einmal genau am Anfang aufgerufen wird, d.h. der Zähler wird erhöht. Zweitens wird `lower()` stets genau einmal am Ende aufgerufen, d.h. der Zähler wird erniedrigt. Der Einsatz von `try ...finally` bewirkt, daß `lower()` auch aufgerufen wird, wenn bei der Arbeit mit der Ressource eine Ausnahme entsteht - gewollt oder ungewollt. Die Zählung ist damit für den Roblet®-Server immer korrekt.

8.3.4.3 Modul-Implementierung

Die Implementierung des Moduls berücksichtigt für das Beispiel natürlich nun auch die Einheit.

Quelltext 8.7 ist die Implementierung des Moduls. Gegenüber den vorigen Modulen wird hier die Initialisierung in Zeile 16 noch um die Erzeugung einer (einzigen) Nachrichtenverwaltung erweitert. Die Methoden `getUnit4Slot(...)` und `resetUnit4Slot(...)` wurden dahingehend erweitert, daß die Einheiten-Implementierung nun berücksichtigt wurde.

Die Methode `getUnit4Slot(...)` wird vom Roblet®-Server aufgerufen, wenn ein Roblet® mit Hilfe seines Kontexts vom Typ `org.roblet.Robot` per `getUnit(module02.MessagesUnit.class)` eine Instanz der Einheiten-Implementierung anfordert. Die vorliegende Zeile 27 der Modul-Implementierung erzeugt der Einfachheit halber pro Aufruf eine neue Instanz. Meist wird ein Roblet® nur einmal einen entsprechenden Aufruf machen. Bei der Instanzerzeugung wird nun Zählerinstanz und die Nachrichtenverwaltung übergeben. Die Methode muß `null` zurückgeben, wenn keine Instanz vom Modul zurückgegeben werden kann, damit der Server im nächsten Modul weitersucht.

Die Methode `resetUnit4Slot(...)` wird vom Roblet®-Server aufgerufen, wenn ein Roblet® endete. D.h. es läuft dann schon nicht mehr bzw. kein Thread des Roblets® läuft noch. Der Aufruf durch den Roblet®-Server erfolgt pro per `getUnit4Slot(...)` zurückgegebener Instanz. Das Modul kann auf diese Weise Aufräumarbeiten erledigen. Im Beispiel wird einfach nur `true` zurückgeben, um zu signalisieren, daß die Instanz aus diesem Modul stammte. Stammt die Instanz nicht aus dem Modul, muß `false` zurückgegeben werden, um dem Roblet®-Server zu signalisieren, daß die Aufräumarbeiten in einem anderen Modul stattfinden.

```
1  package module02;
2
3  import  org.roblet.Unit;
4  import  genRob.genControl.modules.Log;
5  import  genRob.genControl.modules.Module;
6  import  genRob.genControl.modules.Registry;
7  import  genRob.genControl.modules.Slot;
8  import  genRob.genControl.modules.Supervisor;
9  import  genRob.genControl.modules.Use;
10
11 public class  ModuleImpl implements Module {
12
13     public void  moduleInit (Supervisor sv, Use use) throws Exception {
14         log = sv. getLog ();
15         log. out (null, null, "init");
16         messages = new MessagesManager (log);
17     }
18     private Log  log;
19     private MessagesManager  messages;
20
21     public void  moduleDone () {
22         log. out (null, null, "done");
23     }
24
25     public Unit  getUnit4Slot (Class clazz, Use use, Slot slot) {
26         if (clazz == MessagesUnit.class)
27             return new MessagesUnitImpl (use, messages);
28         return null;     // No implementation found
29     }
30
31     public boolean  resetUnit4Slot (Unit unit) {
32         if (unit instanceof MessagesUnitImpl)
33             return true;     // Found (and no special reset needed)
34         return false;    // Not found
35     }
36
37     // not used anymore by the server
38     public Unit  getUnit (Class clazz)  { return null; }
39     public Registry  getRegistry ()     { return null; }
40
41 }
```

Listing 8.7: Java™-Klasse der Implementierung des Moduls unter Berücksichtigung der Einheit.

8.3.4.4 Kompilation und Verpackung

Das Paketverzeichnis (**module02**) vom Beispiel sei ein direktes Unterverzeichnis und die Einheiten-Definition und die Ausnahme sind bereits kompilert und in das Java™-Archiv verpackt, dann erfolgt eine Kompilation unter den Annahmen von Abschnitt 8.1 nun folgendermaßen (in eine Zeile; Unix, Linux und MacOSX):

```
javac  -cp org.roblet.jar:module02-units.jar
        module02/MessagesManager.java
        module02/MessagesUnitImpl.java
        module02/ModuleImpl.java
```

Bzw. (in eine Zeile; Windows):

```
javac  -cp org.roblet.jar:module02-units.jar
        module02\MessagesManager.java
        module02\MessagesUnitImpl.java
        module02\ModuleImpl.java
```

Daraus wird das Java™-Archiv **module02-module.jar** für Administratoren von Roblet®-Servern folgendermaßen (in eine Zeile; Unix, Linux und MacOSX):

```
jar  cf module02-module.jar
        module02/MessagesUnit.class
        module02/MessagesException.class
        module02/MessagesManager.class
        module02/MessagesUnitImpl.class
        module02/ModuleImpl.class
```

Bzw. (in eine Zeile; Windows):

```
jar  cf module02-module.jar
        module02\MessagesUnit.class
        module02\MessagesException.class
        module02\MessagesManager.class
        module02\MessagesUnitImpl.class
        module02\ModuleImpl.class
```

Wieder sei der Einsteiger darauf hingewiesen, daß bei der Kompilation einer **java**-Datei auch mehr als nur eine **class**-Datei entstehen kann. Das muß bei der Erstellung eines Java™-Archiv immer berücksichtigt werden.

Es ist zu bemerken, daß die Klassen vom Java™-Archiv **module02-units.jar** in dem eben erzeugten Java™-Archiv auch mit enthalten sind. Für sehr große Module

```
1  package module02;
2
3  import  genRob.genControl.client.Client;
4  import  java.io.Serializable;
5  import  org.roblet.Roblet;
6  import  org.roblet.Robot;
7
8  public class TestApp {
9
10     public static void  main (String[] args) throws Exception {
11         System.out.println (
12                 new Client (). getServer ("localhost"). getSlot ().
13                         run (new TestRoblet ()));
14     }
15
16     private static class  TestRoblet implements Roblet, Serializable {
17         public Object  execute (Robot r) throws Exception {
18             MessagesUnit  mu = (MessagesUnit) r. getUnit (
19                                                     MessagesUnit.class);
20             mu. store ("Hello!");
21             return mu. retrieve ();
22         }
23     }
24 }
```

Listing 8.8: Roblet®-Anwendung zum Test der Implementierung des Moduls.

ist diese Vorgehensweise nicht optimal, aber für dieses Beispiel völlig ausreichend.[2] Weiterhin wurde das Java™-Archiv gegenüber dem letzten Modul (**module01**) im Namen noch um ein **-module** ergänzt. Dies dient der Abgrenzung gegenüber dem Java™-Archiv mit den Einheiten-Schnittstellen. Derartige Benennungen sind natürlich dem Architekten überlassen.

8.3.5 Test des Moduls

Für den Test des Moduls genügt in unserem Falle eine kleine Roblet®-Anwendung.

Quelltext 8.8 zeigt eine kleine mögliche Test-Roblet®-Anwendung. Es wird ein Roblet® an einen Roblet®-Server auf `localhost` geschickt, welches eine Nachricht schreibt und danach wieder liest. Man könnte auch zwei unabhängige Anwendungen schreiben - eine zum Schreiben und eine zum Lesen - und die Tests sollten auch viel ausführlicher sein und z.B. auch die Ausnahme-Erzeugung prüfen, aber im Rahmen

[2] Man würde mit Hilfe eines Manifestes das Java™-Archiv der Einheitendefinitionen in den Klassenpfad nehmen und es dann immer mit ausliefern.

dieses Kapitels soll nur die Form des Testens angedeutet werden.

Das Paketverzeichnis (**module02**) vom Beispiel sei ein direktes Unterverzeichnis und das Java™-Archiv für die Einheiten-Definition ist bereits erzeugt, dann erfolgt eine Kompilation unter den Annahmen von Abschnitt 8.1 nun folgendermaßen (in eine Zeile; Unix, Linux und MacOSX):

```
javac  -cp org.roblet.jar:module02-units.jar
        module02/TestApp.java
```

Bzw. (in eine Zeile; Klassenpfad ohne Leerzeichen; Windows):

```
javac  -cp org.roblet.jar;module02-units.jar
        module02\TestApp.java
```

Den Roblet®-Server **localhost** mit dem Modul und Logging-Ausgaben in das Terminal bringt man folgendermaßen zum Laufen (in eine Zeile; Unix, Linux, MacOSX):

```
java  -DgenRob.genControl.log=terminal
        -classpath  org.roblet.jar:module02-module.jar
        org.roblet.Main  server
        module02.ModuleImpl
```

Bzw. (in eine Zeile; Windows):

```
java  -DgenRob.genControl.log=terminal
        -classpath  org.roblet.jar;module02-module.jar
        org.roblet.Main  server
        module02.ModuleImpl
```

Für die Test-Roblet®-Anwendung benötigt man nun ein zweites Terminal. Dort bringt man sie folgendermaßen zum Laufen (in eine Zeile; Unix, Linux, MacOSX):

```
java  -classpath  org.roblet.jar:module02-units.jar:.
        module02.TestApp
```

Bzw. (in eine Zeile; Windows):

```
java  -classpath  genRob.genControl.client.jar;module02-units.jar;.
        module02.TestApp
```

Die Anwendung gibt bei erfolgreichem Lauf **Hello!** im Terminal-Fenster aus. Im Terminal-Fenster des Roblet®-Server *localhost:2010* erscheinen Log-Ausgaben des Nachrichtenverwalters.

8.3.6 Auslieferung

Wie schon erwähnt geht Einheiten-Definition inkl. Dokumentation, d.h. die **zip**-Datei, an Entwickler von Roblets® bzw. Roblet®-Anwendungen. Das Java™-Archiv des Moduls geht an Administratoren, d.h. Personen, die Roblet®-Server zum Laufen bringen und betreuen.

8.4 Server mit mehreren Modulen

Angenommen es liege nun zwei Module vor, d.h. z.B. der Anbieter X einer Hardware hat ein Modul mitgeliefert und der Anbieter Y einer anderen Hardware oder Software ebenso. Die Module seien als einfachster Fall in je einem Java™-Archiv mit Namen **x.module.jar** bzw. **y.module.jar** untergebracht und die Modul-Klassen haben die Namen **x.module.ModuleImpl** und **y.module.ModuleImpl**. Die Dateien liegen im gleichen Verzeichnis, wie die Dateien des Servers und es gelten die Annahmen von Abschnitt 8.1.

Dann läßt sich der Server mit den Modulen folgendermaßen starten (in eine Zeile; Unix, Linux, MacOSX):

```
java  -cp org.roblet.jar:x.module.jar:y.module.jar
        org.roblet.Main  server
        x.module.ModuleImpl
        y.module.ModuleImpl
```

Unter Windows muß statt des Doppelpunktes im Klassenpfad ein Semikolon geschrieben werden (in eine Zeile; Windows):

```
java  -cp org.roblet.jar;x.module.jar;y.module.jar
        org.roblet.Main  server
        x.module.ModuleImpl
        y.module.ModuleImpl
```

8.5 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie Server mit Modulen zum Laufen gebracht werden. Weiterhin wurde die Entwicklung eines Moduls inkl. Einheiten-Definition zur Verwaltung einer beispielhaften Ressource beschrieben.

Teil III

Anhang

A Bauen und Ausführen

Zum Bau und zur Ausführung eines Beispieles werden einige Java™-Archive, d.h. **jar**-Dateien, benötigt. Diese Archive sind im Internet bereitgestellt. Unter A.1 ist beschrieben, wie man sie bekommt.

Wie ein Beispiel übersetzt wird, kann bei A.2 nachgelesen werden. Die Ausführung wird unter A.3 beschrieben.

A.1 Herunterladen von benötigten Java™-Archiven

Alle benötigten Java™-Archive sind in zwei Software-Paketen enthalten. Das eine der beiden, das *Roblet®-Development-Kit* , besteht aus mehreren Java™-Archiven und enthält alles, was zur Kommunikation mit Roblet®-Servern benötigt wird. Die andere Bibliothek, hier kurz *Buch-Bibliothek* genannt, besteht nur aus einem Java™-Archiv und enthält die Einheiten-Definitionen, die in diesem Buch erläutert werden.

A.1.1 Roblet®-Development-Kit

Das Roblet®-Development-Kit enthält alles, was für die Entwicklung von Roblet®-Anwendungen grundlegend benötigt wird. Es kann ungeändert auch für professionelle Anwendungen eingesetzt werden.

Zum Herunterladen des aktuellen Roblet®-Development-Kit ohne Dokumentation genügt es, folgendes in einem Browser einzugeben:

`http://roblet.org/rdk`

Nach dem Lesen der Nutzungsbedingungen und weiterer Informationen besteht am Ende dieser Seite die Möglichkeit, die hier benötigten *Arbeitsdateien* herunterzuladen. Der Browser fordert dazu auf, die angegebene Datei abzuspeichern. Nach dem Speichern müssen Sie die Datei auspacken. Sie enthält die benötigten Java™-Archive.

A.1.2 Buch-Bibliothek

Die Buch-Bibliothek enthält die Einheitendefinitionen, die in diesem Buch erläutert werden. Enthalten ist nur ein Java™-Archiv. Es ermöglicht die Nutzung der Einheiten des Roblet®-Servers `roblet.org`.

Zum Herunterladen der aktuellen Buch-Bibliothek ist in einem Browser folgendes einzugeben:

http://roblet.org/book.unit.zip

Der Browser fordert dazu auf, die angegebene Datei abzuspeichern. Nach dem Speichern müssen Sie die Datei auspacken. Sie enthält das benötigte Java™-Archiv mit den Einheitendefinitionen.

A.2 Kompilieren

A.2.1 Verzeichnisstruktur

Bevor man die in diesem Buch beschriebenen Beispiele kompiliert, muß man sicherstellen, daß eine bestimmte Verzeichnisstruktur vorhanden ist. Ist die Struktur anders, so wären andere Pfade anzugeben.

```
(Arbeitsverzeichnis)
    | book.unit.jar
    | org.roblet.jar
    | ...
    | Hello.java
    - hello1
    |   - Hello.java
    - hello2
    |   - Hello.java
   ...
```

Das Arbeitsverzeichnis kann beliebig liegen und heißen. Die beispielhaft angegebenen **hello1** und **hello2** sollen Unterverzeichnisse darstellen. Die angegebenen **java**-Dateien müssen nur vorhanden sein, wenn man das jeweilige Beispiel kompilieren will. Wie in Java™ üblich unterscheiden sich die **java**-Dateien mit Namen **Hello.java** in den Unterverzeichnissen und im Arbeitsverzeichnis dadurch voneinander, daß sie in verschiedenen Java™-Paketen liegen. Abhängig vom Java™-Paket müssen sie in gleichnamigen Unterverzeichnissen liegen. Die heruntergeladenen Java™-Archive müssen direkt im Arbeitsverzeichnis liegen.

A.2.2 JDK für die Kompilation

Eine Kompilation setzt voraus, daß ein JDK[1] installiert ist. Für die Beispiele in diesem Buch genügt ein JDK ab Version 1.5. Das Apple Betriebssystem Mac OS X kommt seit vielen Jahren mit einer passenden Installation. Für x86-Linux, Solaris und Windows wird ein JDK von Sun/Oracle bereitgestellt[2]. Seit einiger Zeit

[1] Java™ development kit

[2] http://java.sun.com

ist auch http://OpenJDK.org ein passender Anlaufpunkt. Für fast alle anderen Betriebssystem-Prozessor-Kombinationen findet sich nach kurzer Suche im Internet auch ein JDK.

Die Installation sollte zur Vereinfachung sichergestellt haben, daß die beiden Programme **java**[3] und **javac**[4] über die Pfadliste der Umgebungsvariable **PATH** gefunden werden. Alternativ muß man ansonsten wissen, wo sie liegen und den kompletten Pfad in den folgenden Zeilen an den Stellen einsetzen, wo sie angegeben sind.

A.2.3 Kommandozeile

Die nachfolgenden Erläuterungen gehen von dem primitivsten Fall aus, daß keine Entwicklungsumgebung zur Verfügung steht. Diesen Fall trifft man immer seltener an, aber die für diesen Fall zu beschreibenden Dinge sind auch für Nutzer von Entwicklungsumgebungen oftmals interessant.

Ohne Entwicklungsumgebung benötigt man für die Kompilation und die Ausführung unter Unix, Mac OS X und Linux ein Terminalfenster[5] mit einer Shell. Unter Windows öffnet man eine DOS-Box oder "Eingabeaufforderung" mit einem Befehlsinterpreter. Terminalfenster bzw. Eingabeaufforderung starten im allgemeinen bereits eine Shell bzw. einen Befehlsinterpreter.

A.2.4 Wechsel zum Arbeitsverzeichnis

Innerhalb der Shell bzw. des Befehlsinterpreters wechselt man als erstes mit **cd** zum Arbeitsverzeichnis.

Das sieht im Windows-Befehlsinterpreter so aus:

```
cd  \(Pfad)\(Arbeitsverzeichnis)
```

Das sieht in einer Unix-Shell so aus:

```
cd  /(Pfad)/(Arbeitsverzeichnis)
```

Es ist wichtig, daß wir dieses Arbeitsverzeichnis im folgenden nicht mehr verlassen werden. Ist man nicht mehr im Arbeitsverzeichnis, führen die nachfolgenden Befehle zu sehr komplexen Fehlermeldungen.

A.2.5 Der Klassenpfad

Bevor wir zur Kompilation kommen, noch eine Bemerkung zu den nachfolgend verwendeten Klassenpfaden (engl. *classpath*, kurz *cp*).

[3] **java.exe** unter Windows
[4] **javac.exe** unter Windows
[5] Falls noch jemand hat - funktioniert natürlich auch ein Terminal.

- Die Klassenpfade für Unix, Mac OS X und Linux einerseits und Windows andererseits unterscheiden sich durch das Trennzeichen für Pfadelemente. Unter Unix, Mac OS X und Linux wird der Doppelpunkt **:** verwendet und unter Windows das Semikolon **;**.
- Für die Shell bzw. den Befehlsinterpreter muß der Klassenpfad zusammenhängend, d.h. ohne Leerzeichen, Zeilenumbruch o.ä., eingegeben werden. Die Umbrüche in diesem Buch sind allein der Tatsache geschuldet, daß ohne sie keine sinnvolle Darstellung der jeweiligen Eingabezeile möglich ist.
- Nicht alle Java™-Archive aus dem Arbeitsverzeichnis müssen im Klassenpfad angegeben werden. Jini™ wird z.B. nie direkt vom Entwickler im Quelltext verwendet.
- Als letztes Pfadelement für die Beispiele dieses Buches muß bei der Ausführung stets der Punkt **.** zu finden sein. Da wir uns im Arbeitsverzeichnis befinden, bedeutet das, daß das Arbeitsverzeichnis Teil des Klassenpfades ist. Bei der Kompilation ist das nicht nötig, schadet aber in unseren Fällen nicht.

Ausführliche Informationen findet man auch in der Dokumentation zu den Java™-Werkzeugen[6].

A.2.6 Kompilieren unter Unix, Mac OS X und Linux

Zum Kompilieren der Datei **Hello.java** im Arbeitsverzeichnis ist unter Unix, Mac OS X und Linux folgendes einzugeben:

```
javac  -cp org.roblet.jar:book.unit.jar  Hello.java
```

Die Datei **Hello.java** im Unterverzeichnis **hello1** wird folgendermaßen kompiliert:

```
javac  -cp org.roblet.jar:book.unit.jar  hello1/Hello.java
```

Bitte beachten Sie, daß stets vom Arbeitsverzeichnis aus kompiliert wird.

A.2.7 Kompilieren unter Windows

Zum Kompilieren der Datei **Hello.java** im Arbeitsverzeichnis ist unter Windows folgendes einzugeben:

```
javac  -cp org.roblet.jar;book.unit.jar  Hello.java
```

Die Datei **Hello.java** im Unterverzeichnis **hello1** wird folgendermaßen kompiliert:

```
javac  -cp org.roblet.jar;book.unit.jar  hello1\Hello.java
```

Bitte beachten Sie, daß stets vom Arbeitsverzeichnis aus kompiliert wird.

[6] [jdk, Tools and Utilities, http://java.sun.com/j2se/1.5.0/docs/tooldocs]

A.2.8 Ausgaben bei der Kompilation

Normalerweise gibt es keine Ausgabe. Wenn doch etwas kommt, so ist es eine Fehlermeldung.

A.2.9 Resultat der Kompilation

Neben den **java**-Dateien sind gleichnamige **class**-Dateien als Resultat der Kompilation in den jeweiligen Verzeichnissen angelegt worden.

A.3 Ausführung

Zu Beachten ist bei der Ausführung, daß die jeweilge **class**-Datei ohne Dateiendung **class** angegeben werden muß.

A.3.1 Ausführen unter Unix, Mac OS X und Linux

Ausgeführt wird die Datei **Hello.class** im Arbeitsverzeichnis unter Unix, Mac OS X und Linux mittels:

```
java  -cp org.roblet.jar:book.unit.jar:.  Hello
```

Die Datei **Hello.class** im Unterverzeichnis **hello1** erfolgt mittels:

```
java  -cp org.roblet.jar:book.unit.jar:.  hello1/Hello
```

A.3.2 Ausführen unter Windows

Ausgeführt wird die Datei **Hello.class** im Arbeitsverzeichnis unter Windows mittels:

```
java  -cp org.roblet.jar;book.unit.jar;.  Hello
```

Die Datei **Hello.class** im Unterverzeichnis **hello1** erfolgt mittels:

```
java  -cp org.roblet.jar;book.unit.jar;.  hello1\Hello
```

Literaturverzeichnis

[jls] "The Java Language Specification, Second Edition", James Gosling, Bill Joy, Guy Steele, Gilad Bracha, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html, April 2000

[jpl] "The Java Programming Language, Third Edition", Ken Arnold, James Gosling, David Holmes, Addison-Wesley, November 2000

[jdk] "Java™ 2 SDK, Standard Edition Documentation", Sun Microsystems, Inc., http://java.sun.com/j2se/1.5.0/docs, 2006

[C++] "The C++ Programming Language, Third Edition", Bjarne Stroustrup, Addison-Wesley, 1997

[rdk] "Roblet®-Development-Kit 1.2", roblet®.org, Hagen Stanek, http://roblet.org/rdk/1.2, 22. Juli 2010