Roblets®

Einführung in die Technik

Version: 0.5

Datum: 20. Februar 2008

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung der Roblet®-Technik begann im Jahre 2001. Einer der Autoren verfolgte das Ziel, eine existierende Steuerung für mobile Plattformen des Fraunhofer IPA, Stuttgart, zu reorganisieren. Dabei sollten Teile der Funktionalität über das Netz zur Laufzeit an den jeweiligen Roboter geliefert und zur Ausführung gebracht werden können. Der Wunsch war (und ist), möglichst vom Schreibtisch aus den überwiegenden Teil der Software zur Steuerung des Roboters und seine Einbindung in das Umfeld entwickeln zu können. Erst Jahre später zeigte sich, daß die dabei entstandene Technik ein unerwartet breites Anwendungsspektrum besitzt. Dabei läßt sich generell in verteilten Systemen (Netzen) eine teilweise deutliche Vereinfachung der Entwicklung, der Inbetriebnahme, der Wartung und der Weiterentwicklung beobachten.

Die Roblet®-Technik profitiert dabei entscheidend von der Tatsache, daß im ursprünglichen Anwendungsgebiet, der mobilen Robotik, besonders extreme Randbedingungen einzuhalten sind. So ist z.B. ein Abreißen der Verbindung zwischen zwei (mobilen) Anwendungen im Netz keine Ausnahmesituation. Oft ist auch die Verbindung einfach nur ”schlecht”, weshalb die Bandbreite dann 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. Regelmäßig ist das Umfeld, in dem Roboter eingebunden werden müssen, sehr heterogen. Dann müssen verschiedenste Anwendungen miteinander kommunizieren und die Rechentechnik (Prozessor, Betriebssystem etc.) ist teilweise drastisch unterschiedlich.

Dieses Buch soll dem Leser eine Einführung in die Roblet®-Technik geben. Dazu wird zunächst ausführlich die Technik an sich erläutert. Anschließend werden Erfahrungen hinsichtlich der Entwicklung von verteilten Systemen vermittelt. Vorausgesetzt werden grundlegende Kenntnisse in Java™. Die Beispiele setzen zur Ausführung eine Internetverbindung voraus.

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 Anwendungsentwicklung zu haben. Aus dieser Anwendung werden in Kapitel 3 Beispiele abgeleitet, die schrittweise die Grundlagen erläutern. Die Vielfalt an Möglichkeiten, die sich automatisch auf Grund der Implementierung in Java™ bieten, sind in Kapitel 4 umrissen. Kapitel 5 beschreibt den Teil der Technik, mit nun der Zugriff auf bereitgestellte verteilte Funktionalität kontrolliert möglich wird.

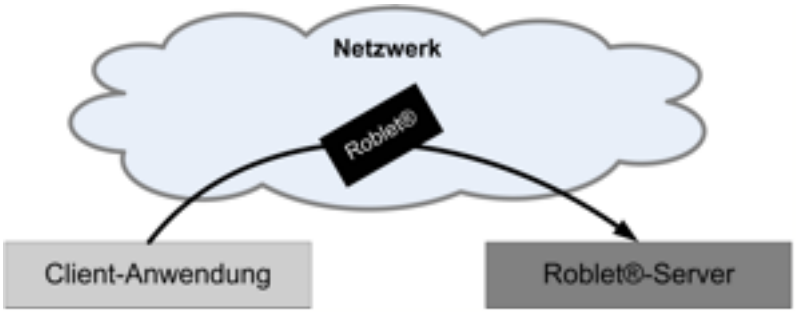

Abbildung 1.1: Einfachste Version einer verteilten Anwendung, eine Client-Anwendung schickt ein Roblet® zu einem Roblet®-Server. Das Roblet® wird anschließend vom Roblet®-Server ausgeführt.

1.1 !!! Ursprüngliche Einleitung

Die ersten Anwendungen die mit Hilfe der Software des genRob®-Projektes erstellt wurden, waren aus dem Bereich der mobilen Robotik. Dabei spielten die folgenden Ideen eine wichtige Rolle, als die vom genRob®-Projekt bereitgestellten Software-Pakete entwickelt wurde[1]:

- Entwickeln einer Anwendung von einem Arbeitsplatzrechner aus.
- Teile einer Anwendung werden zu anderen Rechnern im Netzwerk geschickt und dort ausgeführt.
- Die gesendeten Programmteile nennt man Roblets®.
- Roblets® werden von Roblet®-Servern ausgeführt.
- Über den Roblet®-Server hat man Zugriff auf spezielle Hardware die an dem Rechner, auf dem der Roblet®-Server läuft, angeschlossen ist (z.B. Zugriff auf einen Roboter).

Die Idee Programmteile zu einem Server auf einem Roboter zu senden und dort in einer sicheren Umgebung auszuführen, wird im Folgenden als Roblet®-Technologie bezeichnet. Weitere Details zu den Gedanken, die bei der Entwicklung der Roblet®-Technologie eine Rolle spielten finden sich in den Ausführungen zu den Zielen auf der Webseite des genRob®-Projektes.

Neben der Webseite des genRob®-Projektes ist die Internetseite `http://www.roblet.org` von Bedeutung. Auf ihr wird die Roblet®-Bibliothek bereitgestellt, die die generelle Basis aller Roblet®-Anwendungen und -Server darstellt. Hier werden in

[1] `http://www.genRob.com/de/Why.html`

Zukunft aktuelle Versionen der Bibliothek sowie Erweiterungen derselben bereitgestellt.

Neben der Roblet®-Bibliothek werden einige der Software-Pakete und Bibliotheken aus dem genRob®-Projekt für dieses Tutorial benötigt. Diese lassen sich von der Webseite `http://www.genRob.com/system` herunterladen.

1.2 Verzeichnisstruktur für dieses Tutorial

Die Beispiele dieses Tutorials sind so ausgelegt, dass von der wie folgt beschriebenen Verzeichnisstruktur ausgegangen wird:

```
.../tutorial/
    + lib/
    + src/
        + genRob/
            + tutorial/
                + example1/
                + example2/
                + ...
```

Dabei liegen alle Dateien, die benötigt werden, im Verzeichnis *tutorial* oder in dessen Unterverzeichnissen. Im Verzeichnis *tutorial* selbst liegen die Scripte, um die Beispiele zu kompilieren und auszuführen. Im Unterverzeichnis *lib* liegen die Java™-Bibliotheken die in den Beispielen verwendet werden. Im Verzeichnis *src* liegen die Source-Dateien und nach dem Kompilieren die *class*-Dateien der Beispiele. Dabei gehören alle Beispiele zum Paket *genRob.tutorial*, so dass das Unterverzeichnis *src* noch zwei Ebenen tiefer geschachtelt ist, bevor die eigentlichen Beispiele gespeichert sind.[2]

[2] Informationen darüber, wozu Pakete in Java™ dienen und wie man sie verwendet, finden sich im Java™-Tutorial unter:
`http://java.sun.com/docs/books/tutorial/java/interpack/packages.html`.

2 Hallo Welt!

Um dem Leser einen Eindruck von der Leichtigkeit der Roblet®-Technik zu geben, als erstes eine kleine Roblet®-Anwendung, die lokal den von entfernt stammenden klassischen Text ”Hello World!” ausgibt. Die hier und da auftretenden neuen Begriffe werden im Rahmen dieses Buches nach und nach erklärt.

Listing 2.1 zeigt die Java™-Datei der ”Hallo Welt!”-Anwendung, die man am besten in einem leeren Verzeichnis unter dem Namen ”Hello.java” ablegt.

Um dieses Java™-Programm übersetzen und ausführen zu können, werden noch einige Java™-Bibliotheken benötigt. Diese Bibliotheken sind zur Vereinfachung im Internet bereitgestellt. Anhang A beschreibt, wie man das sog. Klientenpaket bekommt. Das Klientenpaket kann ungeändert auch für professionelle Roblet®-Anwendungen eingesetzt werden.

Die Dateien des Paketes sollten für dieses Beispiel im gleichen Verzeichnis liegen, wie die ”Hello.java”-Datei. Aus diesem Verzeichnis heraus müssen Sie auch kompilieren und ausführen.

Zum Kompilieren ist unter Unix inkl. MacOSX und Linux folgendes einzugeben (in eine Zeile):

```
javac -classpath .:genRob.genControl.client.jar:org.roblet.jar
Hello.java
```

Unter Windows ist hingegen folgendes einzugeben (in eine Zeile):

```
javac -classpath .;genRob.genControl.client.jar;org.roblet.jar
Hello.java
```

Ausgeführt wird unter Unix inkl. MacOSX und Linux mittels:

```
java -classpath .:genRob.genControl.client.jar Hello
```

Unter Windows wird ausgeführt mittels:

```
java -classpath .;genRob.genControl.client.jar Hello
```

Danach sollte folgende Ausgabe erscheinen:

```
Hello World!
```

Dabei läuft folgendes ab:

```
1  import  genRob.genControl.client.*;
2  import  java.io.*;
3  import  org.roblet.*;
4
5  public class  Hello implements Roblet, Serializable {
6      public static void  main (String[] args) throws Exception {
7          System.out.println (
8                  new Client (). getServer ("roblet.org:2000"). getSlot ().
9                      run (new Hello ()));
10     }
11     public Object  execute (Robot robot) {
12         return "Hello World!";
13     }
14 }
```

Listing 2.1: Die "Hallo Welt!"-Roblet®-Anwendung.

- Zunächst wird von der JVM[1] die Methode main aufgerufen (Zeile 6).
- Dann wird eine Instanz der Klasse Hello erzeugt (Zeile 9).
- Zeile 8 bedeutet, daß von dem Server, der auf Host roblet.org an Port 2000 lauscht, ein Fach reserviert wird.
- Zu diesem Fach wird die Roblet®-Instanz transportiert und in ihm zur Ausführung gebracht. Die nun ausgeführte Instanz ist das Roblet®! Für den Transport ist es nötig, daß die Klasse (des Roblets®) serialisierbar ist, was durch die Implementierung der Schnittstelle java.io.Serializable in Zeile 5 angezeigt wird.
- Ausgeführt wird vom Server roblet.org:2000 im Fach die Methode execute der Roblet®-Instanz (Zeilen 11 bis 13).
- In dieser Methode wird eine Instanz der Klasse java.lang.String mit der Zeichenkette "Hello World!" erzeugt.
- Diese Instanz wird von der Methode zurückgegeben (Zeile 12).
- Der Server überträgt diese Instanz zurück zur Anwendung.
- Die übertragene Instanz wird von der Methode run zurückgegeben (Zeile 9).
- Sie wird im Terminal ausgegeben (Zeile 7).

[1] Java™ virtual machine - im vorliegenden Fall das Programm java.

Wesentlich ist dabei, daß die Methode execute auf dem Roblet®-Server roblet.org:2000 ausgeführt wird. Lokal wird nur eine Instanz der Klasse Hello erzeugt - execute wird nicht lokal ausgeführt.

Das Beispiel kann als eine der kleinsten Roblet®-Anwendungen aufgefaßt werden. Eine Roblet®-Anwendung ist dabei einfach eine Anwendung, die die Roblet®-Technik einsetzt und mit Roblet®-Servern kommuniziert. Ein Roblet®-Server genügt den Definitionen der Roblet®-Bibliothek, was im wesentlichen bedeutet, daß man in einer bestimmten Weise kommunizieren kann und gleichzeitig die Roblets® eine gewisse Ablaufumgebung vorfinden. Die Roblet®-Bibliothek ist prinzipiell dadurch definiert, was in der Datei org.roblet.jar zu finden ist.

Generell kann man sagen, daß sich Roblet®-Anwendung und Roblet®-Server deshalb verstehen, weil sie beide die Roblet®-Bibliothek einsetzen. Eine Ausnahme bildet momentan nur die später entstandene und deshalb noch nicht in die Roblet®-Bibliothek eingeflossene Klienten-Bibliothek des Roblet®-Servers genRob®-genControl. Diese Bibliothek ist die Datei genRob.genControl.client.jar. Sie standardisiert die Kommunikation zwischen Roblet®-Anwendung und Roblet®-Server und soll zukünftig in die Roblet®-Bibliothek einfließen, um die Unabhängigkeit der Technik von der jeweiligen Server-Implementierung zu sichern.

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 entfernten Roblet®-Server zur Ausführung gebracht wird. Der zur Ausführung gebrachten Teil muß in Form einer Klasse kodiert sein. Diese Klasse wird zur Verdeutlichung 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 Kodierung, d.h. das Programm eines solchen.

3.1.1 Serialisierbarkeit

In allen Beispielen dieses Buches implementiert die Roblet®-Klasse außerdem noch die Schnittstelle java.io.Serializable. Dies geschieht, weil vor der Übertragung von Instanzen einer Klasse die Instanzdaten serialisiert werden müssen. Dabei erfordert der von Java™ dafür bereitgestellte und den hier beschriebenen Bibiothekten genutzte Mechanismus die Implementierung der Schnittstelle zur Abgrenzung[2].

[1] Vgl. [lib, `.../org/roblet/Roblet.html`]
[2] Vgl. [jdk, Object Serialization, `.../guide/serialization`]

Werden neben der Instanz der Roblet®-Klasse, der Roblet®-Instanz, noch weitere Instanzen mit zum oder auch bei Rückkehr vom Roblet®-Server übertragen, so müssen die zugehörigen Klassen auch serialisierbar sein.

3.1.2 Statisch verschachtelte Roblet®-Klasse

Zur Verdeutlichung der Idee der Roblet®-Klasse trennen wir als erstes gegenüber der ursprünglichen Anwendung Listing 2.1 stärker den Teil der Anwendung, der lokal auf unserem Rechner läuft, von dem Teil, der im Roblet®-Server "roblet.org:2000" 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.

Daneben verlegen wir außerdem aus organisatorischen Gründen unsere Klasse Hello vom unbenannten[4] Java™-Paket in ein eigenes benanntes[5] (Paket hello1). Und schließlich lösen wir noch die Ausdrücke wie z.B. import org.roblet.*[6] zugunsten von z.B. import org.roblet.Roblet[7] auf, um dem Leser zu verdeutlichen, in welchen Paketen sich die verwendeten Typen befinden.

In Listing 3.1 ist die entstandene Anwendung dargestellt. Der abgebildete Quelltext muß in das Unterverzeichnis hello1 in einer Datei mit dem Namen Hello.java abgelegt werden.

Zum Kompilieren ist unter Unix inkl. MacOSX und Linux folgendes einzugeben (in eine Zeile):

```
javac -classpath .:genRob.genControl.client.jar:org.roblet.jar
hello1/Hello.java
```

Unter Windows ist hingegen folgendes einzugeben (in eine Zeile):

```
javac -classpath .;genRob.genControl.client.jar;org.roblet.jar
hello1\Hello.java
```

Ausgeführt wird unter Unix inkl. MacOSX und Linux mittels:

```
java -classpath .:genRob.genControl.client.jar hello1.Hello
```

Unter Windows wird ausgeführt mittels:

```
java -classpath .;genRob.genControl.client.jar hello1.Hello
```

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

[4] Für uns liegt die Klasse 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, §7.5.2 Type-Import-on-Demand Declaration]

[7] [jls, §7.5.1 Single-Type-Import Declaration]

```
1  package hello1;
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 Hello {
9
10     public static void main (String[] args) throws Exception {
11         System.out.println (
12                 new Client (). getServer ("roblet.org:2000"). getSlot ().
13                     run (new HelloRoblet ()));
14     }
15
16     private static class HelloRoblet implements Roblet, Serializable {
17         public Object execute (Robot robot) {
18             return "Hello World!";
19         }
20     }
21
22 }
```

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

Danach sollte wieder folgende Ausgabe erscheinen:

```
Hello World!
```

Zeile 1 von Listing 3.1 ist die angekündigte Reorganisation in das benannte Java™-Paket hello1. Zeilen 3 bis 6 sind aufgelösten import-Ausdrücke. Die Deklaration der Klasse Hello in Zeile 8 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 20 deklarieren diese neue Klasse in statisch verschachtelter Form.

Die Roblet®-Klasse ist in Form einer statisch verschachtelten und nicht als innere[8], 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 serialsiert wird.

Dieser Vorgang ist 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 nicht sofort zu durchschauen sind.

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].

In Listing 3.2 ist die entstandene Anwendung dargestellt. Der abgebildete Quelltext muß diesmal in das Unterverzeichnis hello2 in einer Datei mit dem Namen Hello.java abgelegt werden. Die Kompilierung und Ausführung sind analog zum letzten Beispiel. Wieder sollte folgende Ausgabe erscheinen:

```
Hello World!
```

Die Zeilen 17 bis 21 in Listing 3.2 sind im wesentlichen die verschobenen Zeilen 16 bis 20 von Listing 3.1. Allein die 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 Listing 3.3 mit seiner separaten Roblet®-Klasse HelloRoblet in Listing 3.4.

[8] [jls, §8.1.2 Inner Classes and Enclosing Instances]
[9] Vgl. [jls, §7.6 Top Level Type Declarations].
[10] Vgl. [jls, §8.1.1 Class Modifiers].

```
1  package hello2;
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 Hello {
9
10     public static void main (String[] args) throws Exception {
11         System.out.println (
12                 new Client (). getServer ("roblet.org:2000"). getSlot ().
13                     run (new HelloRoblet ()));
14     }
15
16 }
17 class HelloRoblet implements Roblet, Serializable {
18     public Object execute (Robot robot) {
19         return "Hello World!";
20     }
21 }
```

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

```
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          System.out.println (
9                  new Client (). getServer ("roblet.org:2000"). getSlot ().
10                     run (new HelloRoblet ()));
11     }
12
13 }
```

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

```
1  package hello3;
2
3  import java.io.Serializable;
4  import org.roblet.Roblet;
5  import org.roblet.Robot;
6
7  class HelloRoblet implements Roblet, Serializable {
8      public Object execute(Robot robot) {
9          return "Hello World!";
10     }
11 }
```

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

Die abgebildeten Quelltexte müssen konsequenterweise in das Unterverzeichnis hello3 in die Dateien Hello.java bzw. HelloRoblet.java. Die Kompilierung und Ausführung sind analog zum letzten Beispiel. Wieder sollte folgende Ausgabe erscheinen:

```
Hello World!
```

Im wesentlichen wurden die Zeilen 17 bis 21 von Listing 3.2 aus Listing 3.3 weggelassen und zu den Zeilen 7 bis 11 von Listing 3.4. Außerdem wurden 3 der Import-Deklarationen mit verschoben.

3.2 Der Klient

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

In Listing 3.5 wurde zur Verdeutlichung des Klienten eine eigene Variable für dessen Instanz als Zeile 8 eingefügt. Passend wurde auch die Zeile 10 modifiziert, um diese Variable zu nutzen.

Die Klientenklasse ist eine relativ neue Konstruktion, entstanden nach Wünschen der Nutzer. Früher war direkt ein Protokoll in RMI vorgegeben, welches einer Anwendung erlaubte, mit einem Roblet®-Server Kontakt aufzunehmen und Roblets® zu schicken. Als Beispiel waren einige Klassen verteilt worden, die komplizierten Teile der Kommunikation übernahmen. Nachdem sich herausstellte, daß hauptsächlich diese Beispiele (fast) unverändert überall im Einsatz waren und außerdem RMI auf lange

```
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          System.out.println (
10                 client. getServer ("roblet.org:2000"). getSlot ().
11                     run (new HelloRoblet ()));
12     }
13
14 }
```

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

Sicht abgelöst werden mußte, lag es nahe sie zu einer Bibliothek zusammenzufassen und zu pflegen.

Mittelfristig soll die Klientenklasse und alle zugehörigen in die Roblet®-Bibliothek übergehen, da die Kommunikation zwischen Roblet®-Anwendung und -Server wesentlicher Bestandteil der Definition der Roblet®-Technik ist. Bis Redaktionsschluß für dieses Buch hat ein Verschieben der Klientenklassen von genRob.genControl.client.jar nach org.roblet.jar noch nicht stattgefunden. Zukünftigen Anwendungen entsteht dadurch aber kein Problem, da sie trotzdem ungehindert mit beliebigen Roblet®-Servern kommunizieren können - egal, welche Version an Bibliothek sie benutzen. Allein der Name der verwendeten Klasse könnte sich zukünftig von genRob.genControl.client.Client z.B. nach org.roblet.client.Client verändern.

3.3 Der Repräsentant des Servers

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

Der Klient stellt beim Erzeugen eines Server-Repräsentanten sicher, daß pro (realem) Server nur eine solche Instanz erzeugt wird - oder anders beschrieben, wird man pro Server auf Anfrage immer den gleichen Server-Repräsentanten erhalten[11].

[11] Zu beachten ist jedoch, daß für den hier nicht weiter behandelten Fall von mehreren Klienten

```
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:2000");
11         System.out.println (
12                 server. getSlot ().
13                     run (new HelloRoblet ()));
14     }
15
16 }
```

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

In Listing 3.6 wurde zur Verdeutlichung des Server-Repräsentanten eine eigene Variable in Zeile 10 eingeführt. Diese Variable wird nun in Zeile 12 benutzt. Die Klasse wurde in Zeile 4 deklariert.

Zur Erzeugung eines Repräsentanten baut der Klient eine Verbindung zum Server auf. Kennt er den Server schon, so gibt er die entsprechende Instanz an den Aufrufer zurück. Ist der Server noch nicht bekannt, so wird eine neue Instanz erzeugt und zurückgegeben.

Ein Server ist dabei 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!

Wie schon weiter oben geschrieben, wird in diesem Buch Server-Repräsentant der Einfachheit halber meist kurz mit Server abgekürzt. Da ein (realer) Server in keinem unserer Fälle mit der Anwendung zusammenfällt, kann es daher nur schwer zu Verwechslungen kommen. Man kann sich daher vom Klienten den Server holen und gemeint ist, daß man einen Server-Repräsentanten erhält.

innerhalb einer Anwendung, es auch zu entsprechend vielen Server-Repräsentanten kommen kann.

```
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:2000");
12         Slot    slot   = server. getSlot ();
13         System.out.println (
14                 slot.
15                     run (new HelloRoblet ()));
16     }
17
18 }
```

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

3.4 Ein Fach eines Servers

Ein Roblet®-Server stellt auf Wunsch ein Fach zur Verfügung, in dem ein Roblet® zum Laufen gebracht werden kann. Ein Fach könnte auch als Roblet®-Container aufgefaßt werden.

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.

In Listing 3.7 wurde zur Verdeutlichung des Faches eine eigene Variable in Zeile 12 eingeführt. Diese Variable wird nun in Zeile 14 benutzt. Die Klasse wurde in Zeile 5 deklariert.

Eine Anwendung kann sich (theoretisch) beliebig viele Fächer bereitstellen lassen und verwalten. Pro Fach kann zu einem Zeitpunkt nur ein Roblet® laufen. Ein Fach kann auch leer, d.h. ohne laufendes Roblet® sein.

Ein Fach kann beliebig oft wiederverwendet werden. Es kann mehrfach die gleiche Roblet®-Instanz (vgl. 3.5) geschickt 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 weitere Aktionen durchzuführen. Statt einer Instanz kann auch null geschickt werden, wonach das Fach leer ist.

Geschickt (bzw. beendet) werden Roblets® über die Methode run des Fach-Repräsentanten.

3.5 Die Roblet®-Instanz

Die Instanz einer Roblet®-Klasse wird Roblet®-Instanz genannt.

Die besondere Beleuchtung des Begriffs ist erfahrungsgemäß deshalb sinnvoll, weil die Übertragung über das Netz das Phänomen hervorbringt, daß jedesmal aufs neue eine Instanz "auf der anderen" Seite erzeugt wird, obwohl in manchen Fällen ja die gleiche verschickt wird[12]. Es handelt sich dabei natürlich nicht wirklich um ein Phänomen, sondern ist eine Entscheidung, die beim Entwurf der 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. Aus dem gleichen Grund ist eine Roblet®-Instanz auch kein Roblet® (vgl. 3.7).

In Listing 3.8 wurde zunächst zur Verdeutlichung der Roblet®-Instanz eine eigene Variable in Zeile 14 eingeführt. Diese Variable wird nun in Zeile 17 benutzt. Die Klasse wurde in Zeile 6 deklariert[13].

Zur Ausführung des Roblets® im Fach steht dafür auf Anwendungsseite die Methode run des Fach-Repräsentanten zur Verfügung. Diese Methode liefert eine Instanz vom Typ java.lang.Object oder null zurück[14].

In Listing 3.9 wurde zunächst zur Verdeutlichung des Rückgabewertes der run-Methode eine eigene Variable in Zeile 15 eingeführt. Diese Variable wird nun in Zeile 16 benutzt.

Die Tatsache, daß eine Instanz vom Typ java.lang.Object zurückgegeben wird, mag zunächst wie eine Einschränkung aussehen - ist es aber nicht. Jeder primitiver Typ[15] hat in der Java™-Bibliothek (mindestens) einen zugehörigen Referenztyp[16] und jeder Referenztyp ist von java.lang.Object abgeleitet. Dazu gehören auch Felder[17].

[12] "object streams", wie z.B. ObjectOutputStream (`http://java.sun.com/j2se/1.4.2/docs/api/java/io/ObjectOutputStream.html`) kommen an dieser Stelle nicht zum Einsatz.
[13] Natürlich hätte auch eine Variable vom Typ HelloRoblet erzeugt werden können.
[14] Vgl. [jls, §4.5.2 Variables of Reference Type].
[15] Vgl. [jls, §4.2 Primitive Types and Values].
[16] Vgl. [jls, §4.3 Reference Types and Values].
[17] Vgl. [jls, §10 Arrays].

```
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:2000");
13         Slot    slot   = server. getSlot ();
14         Roblet  roblet = new HelloRoblet ();
15         System.out.println (
16                 slot.
17                     run (roblet));
18     }
19
20 }
```

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

```
1  package hello8;
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:2000");
13         Slot    slot   = server. getSlot ();
14         Roblet  roblet = new HelloRoblet ();
15         Object  object = slot. run (roblet);
16         System.out.println (object);
17     }
18
19 }
```

Listing 3.9: Die "Hallo Welt!"-Roblet®-Anwendung mit separatem Aufruf der run-Methode des Faches.

3.6 Der Typ des Rückgabewertes

In Zeile 9 von Listing 3.4 wird eine Instanz vom Typ java.lang.String zurückgegeben. Aus diesem Grund kann der Rückgabewert auf diesen Typ gewandelt[18] werden.

In Zeile 16 von Listing 3.10 wird eine Variable vom Typ java.lang.String eingeführt und der Rückgabewert der run-Methode passend gewandelt. In Zeile 17 wird die Variable genutzt.

In diesem Buch werden Roblets® stehts im Zusammenhang mit Netz-Übertragungen gebracht. Deshalb sind die verwendeten Rückgabetypen auch stets serialisierbar.

3.7 Das Roblet®

Ein Roblet® ist eine Roblet®-Instanz, welche in einem Roblet®-Server ausgeführt wird.

Sogar eine mehrfach geschickte, gleiche Roblet®-Instanz wird jedes Mal zu einem neuen Roblet® führen.

[18] Vgl. [jls, §5.5 Casting Conversion].

```
1  package hello9;
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:2000");
13         Slot    slot   = server. getSlot ();
14         Roblet  roblet = new HelloRoblet ();
15         Object  object = slot. run (roblet);
16         String  hello  = (String) object;
17         System.out.println (hello);
18     }
19
20 }
```

Listing 3.10: Die "Hallo Welt!"-Roblet®-Anwendung mit gewandeltem Rückgabewert der run-Methode des Faches.

3.8 Zusammenfassung

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

Ein Roblet® ist die in einem Roblet®-Server laufende Instanz einer Klasse, die die Schnittstelle org.roblet.Roblet implementiert. Ein Roblet®-Server ist ein laufendes Programm, welches eine Laufzeitumgebung für Roblets® bereitstellt.

Eine Roblet®-Anwendung kann gleichzeitig mit verschiedenen Roblet®-Servern in Kontakt stehen. In jedem Roblet®-Server können gleichzeitig verschiedene Roblets® erzeugt sein (laufen).

4 Natürliche Fähigkeiten

Dieses Kapitel wird jeder Leser vermutlich nur einmal durcharbeiten müssen. Denn seine eigentliche Aussage ist, daß ein Roblet® im wesentlichen ein kleines Java™-Programm darstellt.

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 wenig 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 Verteilten von Anwendungsfunktionalität banaler Weise mit solch einem Typ, denn so lassen sich schon die Mechanismen des Versendens einer Roblet®-Instanz inkl. Fehlerbehandlung testen.

Die Zeilen 15 bis 19 von Listing 4.1 stellen die Roblet®-Klasse dar. Das einzige, was ein Roblet® tun wird, ist in Zeile 17 dargestellt - es gibt sozusagen nichts zurück und macht nicht mehr.

Da nichts zurückgegeben wird[1], wurde in diesem Beispiel auch der Rückgabewert von run in Zeile 12 ignoriert. Es wird nichts auf dem Terminal ausgegeben.

Beachtenswert ist auch das in Zeile 15 eingeführte Schlüsselwort[2] *private*. Es gibt an, daß die Klasse NaturalRoblet von außerhalb des Körpers der Klasse Natural nicht

[1] Korrekter Weise müßte man sagen, daß null zurückgegeben wird. Aber da dieses null hier keine weitere Bedeutung hat, ist das gleichbedeutend mit nichts. Tatsächlich wird aber null über das Netz übertragen und auch von run zurückgegeben.

[2] Vgl. [jls, §3.9 Keywords].

```
1  package natural00;
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
10     public static void main(String[] args) throws Exception {
11         new Client().getServer("roblet.org:2000").getSlot().
12             run(new NaturalRoblet());
13     }
14
15     private static class NaturalRoblet implements Roblet, Serializable {
16         public Object execute(Robot robot) {
17             return null;
18         }
19     }
20 }
```

Listing 4.1: Roblet®-Anwendung mit der kleinsten Roblet®-Klasse.

zugegriffen werden kann[3]. In unserem Beispiel stellen wir demnach sicher, daß nur wir die Roblet®-Klasse benutzen.

4.2 Die Rückgabe von Werten

Obwohl im vorigen Kapitel behauptet wurde, daß ein Roblet® auch *nichts* zurückgeben kann, muß jedoch immer ein Rückgabewert angegeben werden. Im Fall der einfachsten Roblet®-Klasse wird der Rückgabewert einfach ignoriert, ist aber vorhanden.

Der Definition der Methode execute folgend muß der Rückgabewert eine Instanz vom Typ java.lang.Object sein. Diese Aussage jedoch ist gleichbedeutend mit "beliebige Instanz" (eines Referenztyps[4]).

Einzige muß der jeweilige Wert serialisiert werden können, m.a.W. serialisierbar sein, d.h. die Klasse java.io.Serializable implementieren. Diese Einschränkung entsteht durch die Tatsache, daß im Fall der Nutzung der Klienten-Bibliothek ein Rückgabewert über das Netz übertragen wird - auch wenn der Roblet®-Server möglicherweise

[3] Vgl. [jls, §6.6.1 Determining Accessibility].
[4] Vgl. [jls, §4.3 Reference Types and Values].

```
1  package natural01;
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
10     public static void main (String[] args) throws Exception {
11         Object o =
12             new Client (). getServer ("roblet.org:2000"). getSlot ().
13                     run (new NaturalRoblet ());
14         Integer integer = (Integer) o;
15         int ret = integer. intValue ();
16         System.out.println (ret);
17     }
18
19     private static class NaturalRoblet implements Roblet, Serializable {
20         public Object execute (Robot robot) {
21             int ret = 42;
22             return new Integer (ret);
23         }
24     }
25 }
```

Listing 4.2: Roblet®-Anwendung mit Roblet® welches primitiven Typ zurückgibt.

auf dem gleichen Host zu finden ist.

Viele Standardtypen sind jedoch von Hause aus serialisierbar. Und eigene Typen lassen sich leicht dahingehend einrichten.

4.2.1 Primitiven Typen

Primitive Typen in Java™ sind z.B. int, boolean und double[5]. Da die Definition des Rückgabewertes eines Roblets® eine Instanz der Klasse java.lang.Object sein muß, können primitive Typen nicht direkt zurückgegeben werden.

Stattdessen können jedoch dafür z.B. die "Hüllen"-Klassen[6] der primitiven Typen eingesetzt werden. Für die eben genannten Beispiele wären das java.lang.Integer, java.lang.Boolean und java.lang.Double.

[5] Vgl. [jls, §4.2 Primitive Types and Values].
[6] Vgl. [jpl, §11.1 Wrapper Classes]

In Zeile 21 von Listing 4.2 wird im Roblet® der Variablen ret der Wert 42 zugewiesen. Da ret vom primitiven Typ int ist, verwenden wir in Zeile 22 die ”Hüllen”-Klasse java.lang.Integer, um eine Instanz (eines Referenztyps) zu erzeugen, die dann zurückgegeben werden kann.

Jede Klasse in Java™ ist von java.lang.Object abgeleitet und somit auch java.lang.Integer, wie gefordert. Dazu sind alle ”Hüllen”-Klassen serialisierbar, d.h. implementieren den Typ java.io.Serializable.

Alternativ können auch eigene Typen definiert werden. In Kapitel 4.2.2.3 werden komplexere Rückgabetypen beschrieben, die natürlich im einfachsten Fall z.B. auch nur einen primitiven Typ zurückgeben könnten.

Die in Zeile 22 erzeugte Instanz wird vom Roblet®-Server serialisiert und zu unsere Anwendung zurückübertragen. Die Mechanismen deserialisieren sie wieder auf Seiten der Roblet®-Anwendung im Klienten und geben sie aber zunächst als Instanz vom Typ java.lang.Object an die Anwendung zurück (Zeile 11).

Da wir wissen, daß es sich aber tatsächlich um eine Instanz vom Typ java.lang.Integer handelt, können wir in Zeile 14 auf den Typ java.lang.Integer wandeln[7].

4.2.2 Referenztypen

Referenztypen[8] sind alle Klassen, Felder und Schnittstellen. Instanzen derartiger Typen können direkt von einem Roblet® zurückgegeben werden. Einzige weitere Einschränkung ist nur die schon mehrfach erwähnte Serialisierbarkeit.

4.2.2.1 Vordefinierte Typen

Typen wie java.lang.String erfüllen diese Bedingung von Hause aus, weshalb auch das Hallo-Welt-Beispiel von Kapitel 2 so einfach möglich war.

Analog zum letzten Beispiel wird in Zeile 21 von Listing 4.3 im Roblet® der Variablen ret der Wert 42, diesmal als Zeichenkette, zugewiesen. Da ret vom Referenztyp java.lang.String ist, benötigen wir in Zeile 22 keine ”Hüllen”-Klasse, sondern geben die Instanz direkt zurück.

Etwas aufwendig, aber dafür einfacher mit dem vorigen Beispiel vergleichbar, wandeln wird in Zeile 14 vom Typ java.lang.Object auf java.lang.String. Das ist natürlich möglich, da wir einen solche Instanz zurückgeben.

Die Zeile 15 der Analogie wird zur einfachen Umbenennung[9].

[7] sog. cast - vgl. [jls, §5.5 Casting Conversion].
[8] Vgl. [jls, §4.3 Reference Types and Values].
[9] D.h. in Wirklichkeit haben wir eine zweite Variable.

```
1  package natural02;
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
10     public static void main (String[] args) throws Exception {
11         Object o =
12             new Client (). getServer ("roblet.org:2000"). getSlot ().
13                     run (new NaturalRoblet ());
14         String string = (String) o;
15         String ret = string;
16         System.out.println (ret);
17     }
18
19     private static class NaturalRoblet implements Roblet, Serializable {
20         public Object execute (Robot robot) {
21             String ret = "42";
22             return ret;
23         }
24     }
25 }
```

Listing 4.3: Roblet®-Anwendung mit Roblet® welches vordefinierten Referenztyp zurückgibt.

```
1  package natural03;
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
10     public static void main(String[] args) throws Exception {
11         Object o =
12             new Client().getServer("roblet.org:2000").getSlot().
13                     run(new NaturalRoblet());
14         int[] ret = (int[]) o;
15         for (int i = 0; i < ret.length; ++i)
16             System.out.println(ret[i]);
17     }
18
19     private static class NaturalRoblet implements Roblet, Serializable {
20         public Object execute(Robot robot) {
21             int[] ret = { 42, 43, 44, 45 };
22             return ret;
23         }
24     }
25 }
```

Listing 4.4: Roblet®-Anwendung mit Roblet® welches ein Feld zurückgibt.

4.2.2.2 Felder

Um zu verdeutlichen, daß zu den Referenztypen auch Felder gehören, folgendes Beispiel. Zu bemerken ist, daß ein Feld serialisierbar ist, wenn es der Typ der Feldelemente ist bzw. wenn die Feldelemente einen primitiven Typ haben.

In Zeile 21 von Listing 4.4 wird im Roblet® ein Feld ret erzeugt und und mit Feldelementen initialisiert. Das Feld wird in Zeile 22 zurückgegeben. Es ist auf natürliche Weise die Instanz einer Referenzklasse und serialisierbar.

In Zeile 11 wird die Instanz entgegengenommen. In Zeile 14 findet eine Wandlung statt, von der wir wissen, daß sie fehlerfrei ist, da wir den Referenztyp ja selbst festgelegt hatten. Die Zeile 15 und 16 beschreiben eine Schleife, bei der die Feldelemente ausgegeben werden.

4.2.2.3 Selbstdefinierte Typen

Die Rückgabe von Daten von einem Roblet® ist noch flexibler durch die Tatsache, daß auch selbstdefinierte Typen zurückgegeben werden können. Allein die Serialisierbarkeit muß gegeben sein.

Die Zeilen 25 bis 35 von Listing 4.5 definieren den Rückgabetyp. Für dieses Beispiel wurde der Typ als verschachtelte Klasse angelegt - er hätte z.B. genauso gut in einer separaten Datei residieren können. Die Angabe von static ist bedeutsam für die Serialisierung, da ansonsten bei der Instanziierung eine Referenz der umgebenden Klasse benötigt wurde. Der Rest der Rückgabe-Klasse ist klassisches Java™.

In Zeile 20 wird im Roblet® eine Instanz vom Typ natural04.Natural.Return erzeugt, die in Zeile 21 zurückgegeben wird. In Zeile 11 wird von der Roblet®-Anwendung die Instanz noch als java.lang.Object von run zurückgegeben, aber schon in Zeile 14 in den ursprünglichen Typ gewandelt.

Selbstverständlich kann der selbstdefinierte Typ weitere selbstdefinierte Typen referenzieren. Einzig die Serialisierbarkeit muß gewährleistet sein.

4.3 Parameter - Beigabe von Werten

Im Grunde analog zur Rückgabe von Werten des Kapitel 4.2, können einem Roblet® auch Werte mitgegeben werden.

Diese Werte werden dazu an die Roblet®-Instanz (Kapitel 3.5) gebunden. Die Komplexität ist nicht eingeschränkt - nur die Serialisierbarkeit der gesamten Instanz muß sichergestellt sein.

Die Zeilen 16 bis 26 von Listing 4.6 definieren ein Roblet® mit zwei Attributen und einem passenden Konstruktor. In Zeile 13 wird eine Roblet®-Instanz erzeugt und an run übergeben.

Das Roblet® fügt im Roblet®-Server die beiden Parameterwerte zusammen und gibt das Ergebnis zurück (Zeile 24). Es wird dann per Befehl in Zeile 11 (bis 13) im Terminal ausgegeben.

Selbstverständlich können wie bei den Rückgabewerten auch selbstdefinierte Typen verwendet werden, solange sie serialisierbar sind.

4.4 Variable, Ausdrücke und weitere Klassen

Innerhalb eines Roblets® kann man die volle Funktionalität der Sprache Java™ einsetzen. Es lassen sich Variablen auf Klassen-, Instanz- und Methoden-Ebene definieren. Wie gewohnt setzt man mit Ausdrücken um, was man benötigt, wobei alle bekannten Formen von Verzweigungen, Schleifen usw. einsetzbar sind. Nach Belieben

```java
package natural04;

import genRob.genControl.client.Client;
import java.io.Serializable;
import org.roblet.Roblet;
import org.roblet.Robot;

public class Natural {

    public static void main(String[] args) throws Exception {
        Object o =
            new Client().getServer("roblet.org:2000").getSlot().
                    run(new NaturalRoblet());
        Return ret = (Return) o;
        System.out.println(ret);
    }

    private static class NaturalRoblet implements Roblet, Serializable {
        public Object execute(Robot robot) {
            Return ret = new Return("Level", 42);
            return ret;
        }
    }

    private static class Return implements Serializable {
        private final String s;
        private final int i;
        Return(String s, int i) {
            this.s = s;
            this.i = i;
        }
        public String toString() {
            return s + " " + i;
        }
    }
}
```

Listing 4.5: Roblet®-Anwendung mit Roblet® welches die Instanz eines selbstdefinierten Types zurückgibt.

```java
package natural05;

import genRob.genControl.client.Client;
import java.io.Serializable;
import org.roblet.Roblet;
import org.roblet.Robot;

public class Natural {

    public static void main(String[] args) throws Exception {
        System.out.println(
            new Client().getServer("roblet.org:2000").getSlot().
                    run(new NaturalRoblet("level", 42)));
    }

    private static class NaturalRoblet implements Roblet, Serializable {
        private final String s;
        private final int i;
        NaturalRoblet(String s, int i) {
            this.s = s;
            this.i = i;
        }
        public Object execute(Robot robot) {
            return s + " " + i;
        }
    }
}
```

Listing 4.6: Roblet®-Anwendung mit Roblet® welches verschiedene Parameter erhält.

lassen sich Daten und Programm mit Hilfe von weiteren, eigenen Klassen strukturieren und braucht dabei auch nicht auf Serialisierbarkeit zu achten.

Das in Listing 4.7 angegebene Beispiel hat nicht wirklich eine sinnvolle Funktionalität. Es reißt jedoch gewisse Problembereiche an und zeigt, daß tatsächlich alles möglich ist.

Zu bemerken ist, daß die Klasse Link auch auf Seiten der Roblet®-Anwendung und nicht nur im Roblet® benutzt werden könnte.

```
1  package natural06;
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
10     public static void main(String[] args) throws Exception {
11         new Client().getServer("roblet.org:2000").getSlot().
12                 run(new NaturalRoblet());
13     }
14
15     private static class NaturalRoblet implements Roblet, Serializable {
16         private final static int NUMBER = 3;
17         public Object execute(Robot robot) {
18             Link[] links = new Link[NUMBER];
19             for (int i = 0; i < NUMBER; ++i)
20                 links[i] = new Link();
21             links[2].setNext(links[1]);
22             links[1].setNext(links[0]);
23             return null;
24         }
25     }
26
27     private static class Link {
28         private Link next;
29         void setNext(Link next) {
30             this.next = next;
31         }
32         Link getNext() {
33             return next;
34         }
35     }
36 }
```

Listing 4.7: Roblet®-Anwendung mit Roblet® welches Variable, Ausdrücke und weitere Klassen nutzt.

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 usf.

Ein verteiltes System ist jedoch noch mehr als das. Es besteht aus einzelnen Komponenten, die nicht an oder auf einem normalen Rechner zu finden sind. Solche Komponenten können Hardware sein, wie z.B. das Fahrgestell eines Roboters, aber auch Software, wie z.B. ein Navigationsalgorithmus.

Zwei wesentliche Probleme ergeben sich dabei in der Praxis. Erstens muß man oft prüfen, ob die eigene Software gerade auf dem richtigen Rechner mit der entsprechenden Komponente ist. Und zweitens muß meistens auch sichergestellt sein, daß die Nutzung der Komponente gewissen Regeln folgt. Regeln sind z.B. nötig zur Vermeidung von Zugriffskonflikten, zum Sicherstellung gewisser Handlungen für den Fall, daß die eigene Software inkorrekt funktioniert und dann abrupt abgebrochen wird u.v.a.m.

Dieses Kapitel beschreibt nun, wie das Konzept der Einheiten diese Problematik auffängt.

5.1 Hallo Roblet!

6 Verteilt Rechnen mit Roblets®

Für die Beispiele in diesem Kapitel werden die folgenden *jar*-Archive[1] aus dem genRob®-Projekt benötigt:

- `org.roblet.jar`
- `genRob.genControl.client.jar`
- `jini-core.jar`
- `jini-ext.jar`

Auf den Inhalt der einzelnen Archive wird an den entsprechenden Stellen genauer eingegangen. Die Archive werden mitunter auch als Bibliotheken bezeichnet.

Im weiteren Verlauf werden Programme mit Hilfe von Scripten gestartet. Bei den gezeigten Scripten handelt es sich um Shell-Scripte für Linux. Den Software-Komponenten des genRob®-Projektes liegen neben diesen Shell-Scripten auch Batch-Dateien für Windows bei, die das Ausführen der Programme unter Windows ermöglichen.

6.1 Starten eines Roblet®-Servers

Es kann in Zukunft unterschiedliche Implementierungen eines Roblet®-Servers geben. Dies sollte aber aufgrund der strikt objektorientierten Architektur vor dem Anwendungsentwickler verborgen bleiben. Wie Java™-Programme von den unterschiedlichen Implementierungen der Java™ Virtual Machine nicht betroffen sind, so laufen Roblets® unabhängig von der Implementierung des Roblet®-Servers.

Das genRob®-Projekt bietet eine Implementierung eines Roblet®-Servers als Teil der verfügbaren Softwarekomponenten: *genRob.genControl*. Auf die Möglichkeiten und die Erweiterbarkeit dieses Roblet®-Servers wird in einem späteren Kapitel eingegangen. An dieser Stelle soll nur erläutert werden, wie *genRob.genControl* gestartet werden kann.

[1] *jar*-Archive sind Dateien, die mehrere Dateien zusammenfassen. Der Inhalt eines *jar*-Archives kann komprimiert sein, um Speicherplatz zu sparen. *jar*-Archive werden innerhalb dieses Tutorials oft als Bibliotheken bezeichnet. Weitere Informationen zu *jar*-Archiven finden sich im Java™-Tutorial unter: `http://java.sun.com/docs/books/tutorial/jar/index.htm`

```
1 #!/bin/sh
2
3 cd genControl
4
5 java -jar genRob.genControl.jar
```

Listing 6.1: Script zum Starten von genRob.genControl .

Zuerst muss die Softwarekomponente *genRob.genControl* nach dem Download von der Webseite des genRob®-Projektes[2] in einem geeigneten Verzeichnis entpackt werden. Der Roblet®-Server wird dann durch Aufrufen des mitgelieferten Shell-Scriptes *genControl.sh* gestartet (siehe Listing 6.1). Gegebenenfalls muss das Shell-Script, wie alle in den folgenden Kapiteln vorgestellten Scripte, noch mit dem Befehl

```
> chmod +x genControl.sh
```

ausführbar gemacht werden.

Ein Roblet®-Server wird durch Eingabe von *e* und danach *RETURN* in der Konsole beendet.

6.2 Ein einfaches Beispiel

Wie in der Einleitung erläutert, besteht das einfachste System aus einer Client-Anwendung, einem Roblet® und einem Roblet®-Server (vgl. Abbildung 1.1). Nachdem im vorangegangenen Abschnitt ein Roblet®-Server gestartet wurde, wird in diesem Abschnitt erklärt, wie man eine einfache Client-Anwendung entwickelt, die ein Roblet® zu diesem Server sendet. Dazu bleiben wir vorerst auf demselben Rechner und legen in einem neuen Verzeichnis[3] die im Folgenden vorgestellten und erläuterten Dateien an.

[2] Download unter `http://www.genRob.com/system/`
Hinweis AB TAMS: Auf den Rechnern des AB TAMS kann *genRob.genControl* einfach durch Aufrufen des Befehls *genControl* in einer Shell gestartet werden. Dazu ist es allerdings notwenig, dass das Verzeichnis */local/tams1.2/develop/bin* zur Umgebungsvariablen *$PATH* hinzugefügt wird.

[3] Für dieses Beispiel und alle folgenden gilt: Die *package*-Zeile zu Beginn der Datei ist zu beachten! Dies bedeutet, dass die Java™-Datei *MyRoblet.java* im Verzeichnis *.../tutorial/src/genRob/tutorial/example1/* angelegt werden soll.

```
1  package genRob.tutorial.example1;
2
3  import java.io.Serializable;
4  import java.util.Date;
5  import org.roblet.Robot;
6  import org.roblet.Roblet;
7
8  public class MyRoblet
9     implements Roblet, Serializable
10 {
11    public Object execute (Robot robot)
12    {
13      return new Date ();
14    }
15 }
```

Listing 6.2: Ein einfaches Roblet®, das ein *Date*-Objekt mit der aktuellen Uhrzeit des Servers zurückgibt.

6.2.1 Das erste Roblet®

Zuerst brauchen wir ein Roblet®, welches auf dem Server ausgeführt werden soll. In diesem Beispiel soll das Roblet® nicht viel auf dem Server rechnen, sondern einfach das Datum und die Uhrzeit des Systems ermitteln. Listing 6.2 zeigt die Java™-Datei, die ein solches Roblet® implementiert.

Die Klasse *MyRoblet* implementiert zwei Interfaces: *org.roblet.Roblet* und *java.io.Serializable*.

Das erste Interface *Roblet* stammt aus der Bibliotek *org.roblet.jar* und definiert die Methoden, die von einem Roblet® implementiert werden müssen. Dabei ist die einzige Methode, die zu implementieren ist:

```
public Object execute (Robot robot)
{
  ...
}
```

Als Parameter wird an *execute()* ein Objekt vom Typ *Robot* übergeben. *Robot* wird in einem späteren Kapitel genauer betrachtet und an dieser Stelle nicht näher erläutert.

Ist eine Instanz von *MyRoblet* zum Roblet®-Server übertragen, ruft der Server die Methode *execute()* auf. Das Roblets® endet, wenn die Methode durchlaufen ist.[4]

[4] Weitere Möglichkeiten ein Roblet zu beenden werden in späteren Kapiteln erklärt.

Als Ergebnis einer Berechnung kann eine Instanz vom Typ *Object* von der Methode *execute()* zurückgegeben werden. Diese Instanz wird an die Client-Anwendung zurückgesandt. Die Klasse *Object* ist die Oberklasse aller Klassen in Java™. Daher kann ein Roblet® eine Instanz einer beliebigen Klasse zurückgeben, aber keine primitive Datentypen wie *int* oder *double*. Diese müssen in Instanzen der entsprechenden Wrapper-Klassen umgewandelt werden.[5] Felder mit primitiven Datentyp wie etwa *int[]* müssen nicht umgewandelt werden, sondern sind direkt als Rückgabewert verwendbar.

Neben dem Interface *Roblet* implementiert die Klasse *MyRoblet* das Interface *Serializable*. Da das Roblet® über eine Netzwerkverbindung auf einen anderen Rechner übertragen werden soll, ist es notwendig, dass das Roblet® serialisierbar ist. Dies wird erreicht, indem *Serializable* implementiert wird. Es müssen dafür keine Methoden überschrieben werden.[6]

In diesem ersten Beispiel rechnet das Roblet® auf dem Roblet®-Server nicht viel, sondern holt nur das aktuelle Datum und die Uhrzeit in Form eines *Date*-Objektes[7]. *Date* implementiert genau wie *MyRoblet* das Interface *Serializable* und kann ohne Probleme zur Client-Anwendung zurück gesandt werden.

6.2.2 Verschicken eines Roblets®

Nachdem der Roblet®-Server gestartet und ein eigenes Roblet® entwickelt wurde, fehlt als letzte Komponente eines einfachen verteilten Systems die Client-Anwendung. Diese soll eine Instanz des Roblets® erzeugen und verschicken. Ein Beispiel hierfür wird in Listing 6.3 gezeigt.[8] In der Methode *main()* werden die folgenden fünf Schritte durchgeführt:

1. Eine Netzwerkverbindung aufbauen,
2. eine Instanz von *MyRoblet* erzeugen,
3. die Instanz versenden,
4. das Datum ausgeben und
5. die Netzwerkverbindung schließen.

[5] Für *int* ist die Wrapper-Klasse zum Beispiel Integer.
[6] Mehr zur Serialisierung von Objekten findet man auf den Webseiten `http://java.sun.com/j2se/1.4.2/docs/api/java/io/Serializable.html` und `http://java.sun.com/docs/books/tutorial/essential/io/serialization.html`
[7] Dokumentation: `http://java.sun.com/j2se/1.4.2/docs/api/java/util/Date.html`
[8] Im Verlauf des Tutorials werden alle Klassen *Main* genannt, in denen die Methode *public static void main (String[] args)* implementiert wird.

```java
package genRob.tutorial.example1;

import java.lang.Exception;
import java.util.Date;
import genRob.genControl.client.Client;

public class Main
{
  public static void main(String[] args)
        throws Exception
  {
    // Aufbau der Netzwerkverbindung
    Client client = new Client();

    try {
      // Roblet erstellen
      MyRoblet roblet = new MyRoblet();

      // Roblet verschicken
      Date date = (Date) client.run(roblet);

      // Datum und Uhrzeit ausgeben
      System.out.println
        ("Datum und Uhrzeit des Roblet-Servers: " + date);
    }
    finally {
      // Netzwerkverbindung schließen
      client.close();
    }
  }
}
```

Listing 6.3: Eine einfache Anwendung, die ein Roblet® verschickt.

Um im ersten Schritt die Netzwerkverbindung aufzubauen, reicht es eine Instanz der Klasse *Client* aus der Bibliothek *genRob.genControl.client.jar* zu erzeugen. *Client* versteckt die Details der Kommunikation zwischen Client-Anwendung und Roblet®-Server. Wie immer bei der Kommunikation über Netzwerke können bei der Initialisierung schon Ausnahmen auftreten. Der Aufruf von

```
Client client = new Client ();
```

kann solche Ausnahmen auslösen. In diesem Fall wird die Methode *main()* beendet und der *Stack Trace* der Ausnahme ausgegeben.[9]

Danach folgt ein *try-finally*-Block, innerhalb dessen der Rest der Anwendung programmiert wird. Sollte dieser Block durch eine Ausnahme verlassen werden, wird in jedem Fall noch der Befehl *client.close()* ausgeführt. Damit ist sichergestellt, das die Netzwerkverbindung sauber beendet wird.[10]

Steht die Netzwerkverbindung, kann eine Instanz von *MyRoblet* erzeugt werden:

```
MyRoblet roblet = new MyRoblet ();
```

Mit dem Aufruf der Methode *client.run (roblet)* wird das Roblet® versandt:

```
Date  date = (Date) client.run (roblet);
```

Es sind die folgenden beiden Punkt zu beachten:

- Wird der Methode *run()* nur ein Roblet® als Parameter übergeben und kein Server, so wird dieses Roblet® an einen Roblet®-Server versandt, der auf demselben Rechner wie die Client-Anwendung läuft.

- Der Rückgabewert von *run()* entspricht dem Rückgabewerte der Methode *execute()*, die das Roblet® implementiert. In beiden Fällen wird eine Instanz vom Typ *Object* zurückgegeben. Da der Entwickler der Client-Anwendung ebenfalls das Roblet® implementiert, ist der wirkliche Typ des Rückgabeobjektes bekannt und kann entsprechen umgewandelt werden.

6.2.3 Kompilieren und Ausführen

Nachdem die ersten beiden Beispieldateien erstellt sind, müssen diese noch kompiliert werden, um die Anwendung starten zu können. Sowohl das Kompilieren als auch das Starten übernimmt das in Listing 6.4 gezeigte Script.

[9] Ein kurze Erklärung, was ein *Stack Trace* ist, findet sich im Online-Buch „*Java ist auch eine Insel*“ der Galileo Press GmbH unter: `http://www.galileocomputing.de/openbook/javainsel5/javainsel07_004.htm`

[10] Mehr zu *try-finally*-Blöcken unter: `http://java.sun.com/developer/TechTips/1998/tt0915.html#tip2`

```
1  #!/bin/sh
2
3  # Abbrechen bei Fehlern
4  set -e
5
6  # Hilfsvariablen für Verzeichnisse definieren
7  SRC=src
8  LIB=genControl
9  DESTINATION=classes
10
11 # Java-Classpath
12 CLASSPATH=$DESTINATION:\
13 $LIB/genRob.genControl.client.jar:\
14 $LIB/org.roblet.jar:\
15 $LIB/jini-ext.jar
16
17 # Kompilieren
18 javac -sourcepath $SRC \
19       -classpath $CLASSPATH \
20       src/genRob/tutorial/example1/Main.java \
21       -d $DESTINATION
22
23 # Ausführen
24 java -classpath $CLASSPATH genRob.tutorial.example1.Main
```

Listing 6.4: Das Listing zeigt das Shell-Script *example1.sh* zum Kompilieren und Ausführen des ersten Beispiels.

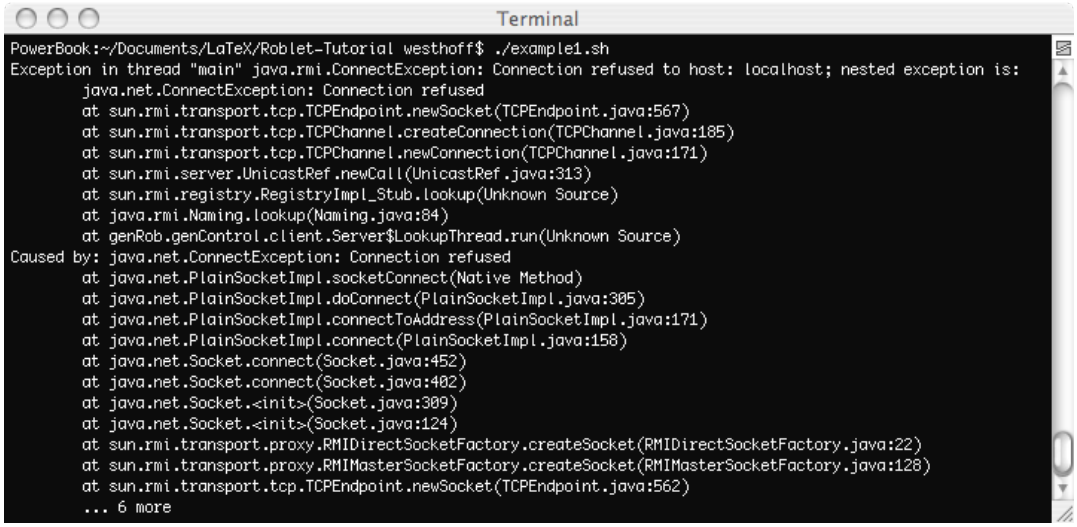

Abbildung 6.1: Wenn *example1.sh* ausgeführt wird ohne das ein Roblet®-Server auf dem selben Rechner läuft, wird eine Ausnahme ausgelöst und der dargestellte *Stack Trace* ausgegeben.

Der Befehl *javac* übersetzt die erstellten Quellcode-Dateien in Java™-Bytecode. Die Pfade zu den Dateien und Bibliotheken werden als Umgebungsvariablen zu Beginn des Scripts definiert. Die erzeugten *class*-Dateien werden in das Unterverzeichnis *classes* geschrieben.[11]

In der letzten Zeile des Scripts wird die Klasse *Main* gestartet, die das Roblet® *MyRoblet* zu einem Roblet®-Server auf demselben Rechner schickt. Sollte nicht bereits, wie in Kapitel 2.1 beschrieben, ein Roblet®-Server laufen, wird eine Ausnahme vom Typ *java.net.ConnectException* ausgelöst (vgl. Abbildung 6.1). Läuft ein Roblet®-Server so sollte das aktuelle Datum und die Uhrzeit des Systems ausgegeben werden (vgl. Abbildung 6.2).

Es bringt erst einmal keinen Vorteil einen Roblet®-Server auf demselben Rechner zu starten wie die Client-Anwendung. Daher wird im folgenden Kapitel erläutert, wie ein Roblet® zu einem anderen Rechner geschickt wird, der im selben Subnetz läuft.

6.3 Roblet®-Server auf entfernten Rechnern

Um eine verteilte Anwendung zu schreiben ist es notwendig, dass Programmteile der Gesamtanwendung auf verschiedenen Rechnern laufen. Die Roblet®-Technologie erlaubt es, dass die gesamte Anwendung auf einem Rechner entwickelt wird und

[11] Man beachte, dass aufgrund der Angabe eines Paketes jeweils in der ersten Zeile der beiden Quellcode-Dateien entsprechende Unterverzeichnisse im Verzeichnis *classes* erzeugt werden.

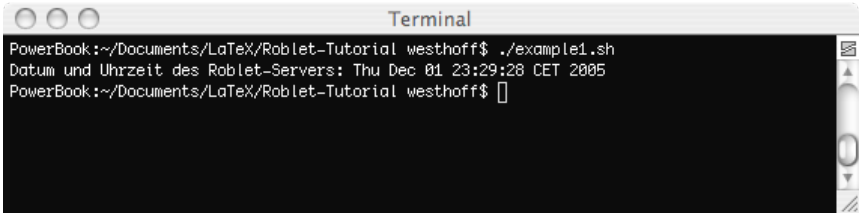

Abbildung 6.2: Wird *example1.sh* erfolgreich ausgeführt, wird eine Meldung ähnlich der abgebildeten Ausgabe erzeugt.

```
1  #!/bin/sh
2
3  cd genControl
4
5  java -DgenRob.genControl.port=$1 -jar genRob.genControl.jar
```

Listing 6.5: Das Script *genControl2.sh* erlaubt das Starten von *genRob.genControl* auf einem bestimmten Port. Der Port wird als Kommandozeilenparameter ($1) der Systemeigenschaft *genRob.genControl.port* zugewiesen.

dann auf das Netzwerk verteilt wird. Dazu werden Teile des Programms als Roblet® entwickelt und an Server im Netzwerk verschickt, die diese Programmteile ausführen. Die folgenden Unterkapitel erläutern wie man die Roblets® an Roblet®-Server im Netzwerk verschickt.

6.3.1 Starten eines Roblet®-Servers auf einem entfernten Rechner

Genauso wie in Abschnitt 6.1 erläutert, kann auf einem beliebigen Computer im Netzwerk ein Roblet®-Server gestartet werden. Zusätzlich soll an dieser Stelle erwähnt werden, dass auch mehrere Server auf einem Rechner gestartet werden können. Dies ist zum Beispiel nützlich wenn Multiprozessor-Rechner zur Verfügung stehen.[12] Die Server unterscheiden sich dabei durch den Port auf dem sie auf eingehenden Verbindungen lauschen. Eine einfache Änderung im Script *genControl.sh* aus Listing 6.1 erlaubt das Starten des Roblet®-Servers auf einem vorgegebenen Port (vgl. Listing 6.5).

Mit dem Kommandozeilenparameter *-D* des *java*-Befehls können als Eigenschaft bezeichnete Variablen definiert und gesetzt werden.[13] Diese können inner-

[12] Auch bei der Entwicklung von Modulen zur Abstraktion bestimmter Hardware, wie es in einem späteren Kapitel erklärt wird, ist das Starten mehrerer Roblet®-Server auf einem Rechner sinnvoll, um neue Funktionalität zu Testen ohne den Betrieb bestehender Anwendungen zu gefährden.

[13] Mehr zu Eigenschaften unter: `http://java.sun.com/docs/books/tutorial/essential/`

halb der Java™-Anwendung abgefragt werden. Durch Angabe der Eigenschaft *genRob.genControl.port=port* wird eine Portnummer gesetzt. Ist diese belegt, muss eine andere Portnummer angegeben werden. Für das Script *genControl2.sh* muss die Portnummer als Kommandozeilenparameter angegeben werden (siehe Listing 6.5).

6.3.2 Senden eines Roblets® an einen entfernten Rechner

Um ein Roblet® an einen Roblet®-Server auf einem anderen Rechner zu schicken ist nur eine kleine Erweiterung des Beispiels aus Abschnitt 6.2 notwendig. In Listing 6.3 muss die Zeile geändert werden, in der *client.run()* aufgerufen wird:

```
Date date = (Date) client.run (roblet, "rechnername:port");
```

Für *rechnername* muss der Name oder die IP-Adresse des Rechners, auf dem der Roblet®-Server läuft eingesetzt werden. Läuft der Server nicht auf dem Standardport, sondern wurde wie in Unterabschnitt 6.3.1 beschrieben auf einem bestimmten Port gestartet, so muss die Portnummer im Anschluss an den Rechnernamen angegeben werden. Rechnername und Portnummer werden durch einen Doppelpunkt getrennt.

Um das zweite Beispiel zu kompilieren und auszuführen, muss nur die Datei *example2.sh* aufgerufen werden. Vorher sollte ein Roblet®-Server auf einem anderen Rechner gestartet werden und in der Datei *src/genRob/tutorial/example2/Main.java* sollte der Name des Rechners und die Portnummer des Roblet®-Servers angepasst werden. Wenn das Programm ausgeführt wird, sollte das Datum und die Uhrzeit des entfernten Rechners angezeigt werden.

Es können eine Reihe unterschiedlicher Ausnahmen auftreten, wenn man Roblets® an einen anderen Rechner verschickt. Die Ursachen für die Ausnahmen sind dabei vielseitig:

- **java.rmi.ConnectionException:** Der Roblet®-Server kann nicht erreicht werden. Vermutlich läuft der Server auf einem anderen Port oder ist nicht gestartet worden (vgl. Abbildung 6.1).
- **java.rmi.UnknownHostException:** Der Name ist dem System nicht bekannt, der für den Rechner angegeben wurde, an den das Roblet® verschickt werden soll (vgl. Abbildung 6.3).

6.3.3 Zusammenfassung

- Eine Client-Anwendung erhält Zugriff auf die Netzwerkumgebung mit einer Instanz der Klasse *Client* aus der Bibliothek *genRob.genControl.client.jar*.
- Roblets® werden an einen Roblet®-Server versendet, indem sie als Parameter an die Methode *run()* der Instanz der Klasse *Client* übergeben werden.

`system/properties.html`

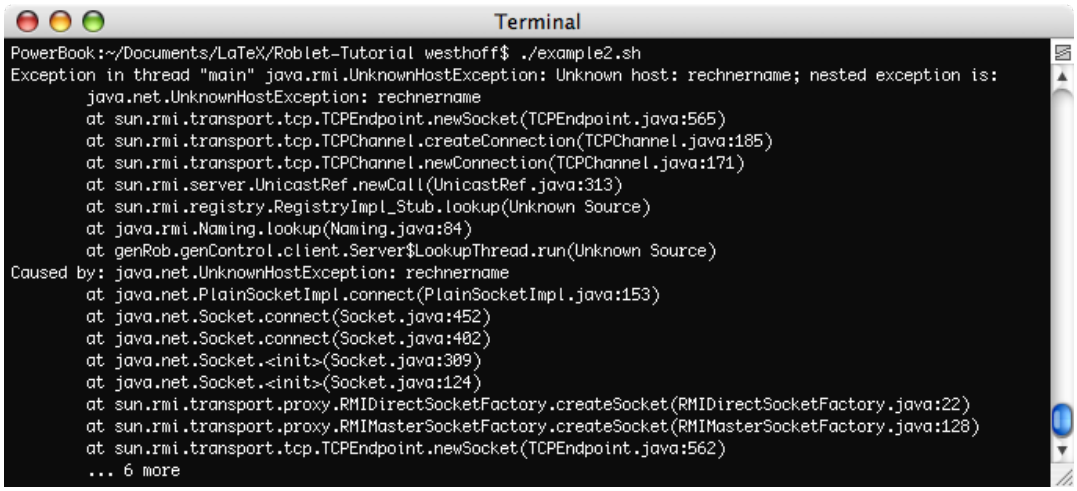

Abbildung 6.3: Ein Roblet® konnte nicht versandt werden, da der angegebene Rechner dem System unbekannt ist.

- Roblets® sind Klassen, die die beiden Interfaces *org.roblet.Roblet* und *java.io.Serializable* implementieren.
- Nachdem ein Roblet® übertragen wurde, ruft der Roblet®-Server die Methode *execute()* des Roblets® auf.
- Ein Roblet® ist beendet, wenn die Methode *execute()* abgearbeitet wurde.
- Die Methode *execute()* gibt eine Instanz vom Typ *Object* an die Client-Anwendung zurück.
- Dieses Objekt muss das Interface *Serializable* implementieren.
- Der Zugriff auf die Netzwerkumgebung wird durch den Aufruf der Methode *close()* der Instanz der Klasse *Client* beendet.

7 Debuggen entfernt laufender Programme

Nachdem gezeigt wurde, wie man Programmteile mit der Roblet®-Technologie auf entfernten Rechner ausführt, sollen in diesem Kapitel die Möglichkeiten zum Debuggen einer auf Roblets® basierenden verteilten Anwendung vorgestellt werden. Die mit der Roblet®-Technologie gegebenen Möglichkeiten dienen dazu, sowohl die Entwicklung als auch das Debuggen fehlerhafter Anwendungen von einem Client-Rechner aus durchzuführen. Dies ist um so wichtiger, da in den meisten Fällen kein anderer Zugriff auf den Rechner auf dem der Roblet®-Server läuft möglich ist, als über Roblets® selbst.

7.1 Log- und Debug-Ausgaben in einem Roblet®

Die einfachste Methode die korrekte Ausführung von Programmen zu überwachen, ist die Ausgabe von Informationen mit *System.out.println()*. Da ein Roblet® nicht auf dem lokalen Rechner ausgeführt wird, sondern von einem Roblet®-Server auf einem entfernten Rechner, werden mit *System.out.println()* ausgegebene Informationen auf der Konsole des Roblet®-Servers ausgegeben.

Listing 7.1 zeigt ein ähnliches Roblet® wie in Listing 6.2. Allerdings werden hier das Datum und die Uhrzeit mit *System.out.println()* auf der Konsole des Roblet®-Servers ausgegeben. Erst danach werden das Datum an die Uhrzeit an die Client-Anwendung zurückgeschickt.

Um das Beispiel auszuführen, starten sie zuerst *genControl.sh* und dann *example3.sh* in einem weiteren Terminalfenster. In beiden Terminals wird die gleiche Zeitangabe ausgegeben.

```
1  package genRob.tutorial.example3;
2
3  import java.io.Serializable;
4  import java.util.Date;
5  import org.roblet.Robot;
6  import org.roblet.Roblet;
7
8  public class MyRoblet
9     implements Roblet, Serializable
10 {
11    public Object execute (Robot robot)
12    {
13          // Datum und Uhrzeit bestimmen
14      Date date = new Date ();
15
16      // Datum und Uhrzeit ausgeben
17       System.out.println
18         ("Datum und Uhrzeit des Roblet-Servers: " + date);
19
20          // Datum und Uhrzeit an die Client-Anwendung zurückgeben
21      return date;
22    }
23 }
```

Listing 7.1: Dieses Roblet® holt sich die Zeit und das Datum auf dem Roblet®-Server und gibt diese auf dessen Konsole aus. Dann werden Zeit und Datum an die Client-Anwendung zurückgesandt.

8 Hardwareabstraktion mit Roblet®-Servern

9 Umgang mit mehreren Roblet®-Servern

10 Ein Modul für genControl

TODO:

- Hier sollten wir mal was schreiben :-)
- man kann auch mehrere TODO Punkte einfügen

11 Changes

A Klientenpaket besorgen

Das Klientenpaket ist ein Sammlung von Bibliotheken zur Entwicklung von Roblet®-Anwendungen. Mit ihrer Hilfe kann mit Roblet®-Servern kommuniziert werden. Das Paket kann von der roblet.org-Website heruntergeladen werden.

Dazu genügt es im Regelfall, folgendes in einem Browser einzugeben:

```
http://roblet.org/client.zip
```

Der Browser fordert dazu auf, die angegebene Datei abzuspeichern. Nach dem Speichern können Sie die Datei auspacken.

Für die in diesem Buch beschriebenen Beispiele kopiert man die entstandenen Dateien schließlich einfach in das (Wurzel-)Verzeichnis der eigenen Quellen.

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/1.4.2/docs, 2003

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

[lib] "Die Roblet®-Bibliothek 2.0 (20Aug05)", Hagen Stanek, http://roblet.org/library/2.0/doc/lib, 2005