Fachhochschule
Bonn-Rhein-Sieg

FH Bonn-Rhein-Sieg
FB Angew. Informatik

Seminararbeit
für
Verteilte und parallele System I

„Socket Factories“

Themensteller:
Prof. Dr. Rudolf Berrendorf

Autorenteam:
Nicole Maier
Alexander Piel

INHALT:

1 Design Pattern

1.1 Grundlegendes zu Design Patterns

Christopher Alexander sagte einmal “Jedes Muster (Pattern) beschreibt ein Problem, das in unserer Umgebung immer und immer wieder vorkommt und beschreibt dann den Kern der Lösung für dieses Problem auf eine solche Weise, dass diese Lösung Millionen Male benutzt werden kann, ohne jemals dasselbe zweimal tun zu müssen.“ Zwar sprach Christopher Alexander über Muster in Gebäuden und Städten, aber was er sagte ist auch zutreffend für objekt-orientierte Design Patterns.
Objekt-orientierte Design Patterns liefern wiederverwendbare Lösungen für ein Problem in Form von Objekten und Schnittstellen.

Ein Design Pattern ist das für einen bestimmten Problemkreis schriftlich festgehaltene Expertenwissen. Es beschreibt ein Problem, welches in ähnlicher Form immer wieder auftritt und seine Lösung.

Die wesentliche Struktur eines Design Patterns:

- Der Name des Pattern
  Der Name des Pattern beschreibt das Problem, die Lösung und die Folgen in einem kurzen prägnanten und ausdruckstarken Begriff.
- Das Problem
  Das Problem, welches von dem Pattern gelöst wird.
- Die Lösung
  Die Lösung beschreibt die Elemente die das Design ausmachen, ihre Beziehung zueinander, ihre Verantwortlichkeiten und ihre Zusammenarbeit.
- Die Konsequenzen
  Die Ergebnisse die entstehen, wenn das Pattern verwendet wird.

Ein Design Pattern benennt, abstrahiert und identifiziert den Schlüsselaspekt einer allgemeinen Designstruktur, um ein wiederverwendbares objekt-orientiertes Design erstellen zu können.

Um Design Patterns wiederverwenden zu können, müssen Intention, Motivation und ähnliches, was zur Entstehung des Design Patterns geführt hat, festgehalten werden. Auch konkrete Beispiele sind sehr wichtig, um zu veranschaulichen was das Design Pattern tut.
Im folgenden soll gezeigt werden, wie man ein Design Pattern beschreibt:

- Name des Pattern
  Der Name eines Patterns spiegelt den Kern des Patterns wieder. Daher ist ein gut gewählter Name sehr wichtig.
- Zweck
  Dies ist ein kurzes Statement dazu, was das Pattern tut und für welches spezielle Problem es gedacht ist.
- Auch bekannt als
  An dieser Stelle werden andere Namen (falls vorhanden), unter denen das Pattern ebenfalls bekannt ist, genannt.
- Motivation
  Eine eher abstrakte Beschreibung des Design Problems, für welches das Pattern gedacht ist.

- Anwendbarkeit
  Kurze Beschreibung der Situationen in denen das Pattern Anwendung findet.
- Struktur
  Graphische Darstellung der Klassen die im Pattern Verwendung finden.
- Beteiligte Klassen und/oder Objekte
  Klassen und Objekte, die im Design Pattern Verwendung finden.
- Zusammenarbeit der Beteiligten Klassen / Objekte
  Beschreibung des Zusammenwirkens der im Pattern beteiligten Klassen und Objekte um ihre Verantwortlichkeiten herauszufinden.
- Konsequenzen
  Beschreibung der Folgen, die durch die Benutzung eines Patterns eintreten.
- Implementation
  Beschreibt welcher Fallen und Techniken man sich bewusst sein sollte, wenn man das Pattern implementiert.
- Codestück
  Hier findet man ein Codefragment, um zu zeigen, wie das Pattern zu implementieren ist.
- Bekannte Nutzungen
  Beispiele an gefundenen Systemen, in denen das Pattern benutzt wird.
- Ähnliche Patterns
  Aufzählung von Patterns, die sehr eng mit diesem Pattern verbunden sind und Heraushebung ihrer Unterschiede.

1.2 Wozu sind Design Patterns gut

Design Patterns dienen dazu, eine geordnete Beschreibung von effizienten Lösungen zur Verfügung zu haben, die immer wieder verwendet werden kann.
Auf diese Weise wird eine Art von Schablone für immer wiederkehrende Probleme erzeugt, so dass die Lösung für diese Probleme nicht immer und immer wieder neu entwickelt werden muss.

1.3 Design Probleme die von Design Patterns gelöst werden können

Design Patterns lösen viele der Tag für Tag auftretenden objekt-orientierten Design Probleme auf viele verschiedene Arten.
Im nachfolgenden sollen einige dieser Probleme, die von Design Patterns gelöst werden können aufgezeigt werden. Zusätzlich soll gezeigt werden, wie Design Patterns die Probleme lösen.

Passende Objekte finden:
Objekt-orientierte Programme bestehen aus Objekten. Diese Objekte enthalten sowohl Daten, als auch Prozeduren, die auf diesen Daten operieren. Prozeduren werden im allgemeinen als Methoden oder Operationen bezeichnet. Eine Operation wird ausgeführt, sobald ein Objekt einen Request von einem Client erhält.
Solche Requests sind der einzige Weg, um ein Objekt eine Operation ausführen zu lassen und Operationen sind die einzige Möglichkeit, um die internen Daten eines Objekts zu verändern. Das heißt, der interne Zustand eines Objekts ist nach außen hin unsichtbar (verkapselt), und kann nicht direkt angesprochen werden.
Das schwierigste am objekt-orientierten Design ist, das System in Objekte aufzuteilen. Dies ist deshalb so schwierig, weil viele Faktoren, wie Verkapselung,

Abstimmung, Flexibilität, Abhängigkeiten, Performance, Entwicklung, Wiederverwendbarkeit, usw., zusammenspielen, und einander oftmals auch in widersprüchlicher Art beeinflussen.
Es gibt viele verschiedene Vorgehensweisen, für Objekt-Orientierte Design Methoden. Man kann eine Problemdarstellung ausschließlich aus Nomen und Verben schreiben, und entsprechende Klassen und Operationen entwickeln. Des weiteren kann man den Fokus auf die Zusammenarbeit und die Verantwortlichkeiten des Systems legen. Oder man modelliert die reale Welt und übersetzt die während der Analyse gefundenen Objekte in ein Design. Es wird immer unterschiedliche Meinungen geben, welches Vorgehen das beste ist.
Viele Objekte in einem Design kommen aus dem Analyse-Modell. Jedoch enden objekt-orientierte Designs oft mit Klassen, die kein Ebenbild in der realen Welt haben. Die strikte Modellierung der realen Welt führt zu einem System, welches die heutige Realität wiederspiegelt, jedoch nicht die morgige. Die Abstraktionen die während des Designs entsteht sind der Schlüssel dazu ein Design flexibel zu machen.
Design Patterns helfen weniger ersichtliche Abstraktionen zu erkennen, und die Objekte, die dazugehören.

Bestimmen der Objektauflösung:
Objekte können in ihrer Größe und Anzahl stark variieren. Sie können alles bis hinunter zur Hardware, und wieder herauf bis zu den gesamten Programmen repräsentieren. Design Patterns helfen zu entscheiden, was ein Objekt ist, und was nicht.

Objektschnittstellen spezifizieren:
Die Schnittstelle eines Objekts charakterisiert den kompletten Satz an Requests die zum Objekt gesendet werden können.
Ein Typ ist ein Name, der dazu verwendet wird, eine spezielle Schnittstelle zu kennzeichnen. Ein Objekt kann mehrere Typen haben und weitgehend verschiedene Objekte können einen Typ teilen. Ein Teil einer Objektschnittstelle kann von einem Typ charakterisiert werden und ein anderer Teil von anderen Typen. Zwei Objekte des selben Typs brauchen nur Teile ihrer Schnittstelle zu teilen. Schnittstellen können andere Schnittstellen als Teilmenge enthalten. Man nennt einen Typ Untertyp eines anderen, wenn seine Schnittstelle die Schnittstelle des Elterntyps enthält. Oft spricht man davon, das ein Untertyp die Schnittstelle seines Elterntyps erbt.
Objekte sind nur durch ihre Schnittstellen nach außen bekannt. Es gibt keine Möglichkeit etwas über ein Objekt herauszufinden, oder es zu veranlassen etwas zu tun, ohne über die Schnittstelle zu gehen.
Design Patterns helfen Schnittstellen zu definieren, indem ihre Schlüsselelemente und die Art der Daten die über die Schnittstelle gesendet werden identifiziert werden. Ein Design Pattern kann auch definieren, was nicht in eine Schnittstelle gehört.
Design Patterns spezifizieren auch die Beziehungen zwischen Schnittstellen.

Wiederverwendbare Mechanismen zum funktionieren bringen:
Die meisten Menschen verstehen Konzepte wie Objekte, Schnittstellen, Klassen und Vererbung. Die Herausforderung liegt darin, sie zu verwenden um flexible, wiederverwendbare Software zu entwickeln, und Design Patterns können zeigen, wie das geht.

Vererbung versus Struktur

Die beiden gebräuchlichsten Techniken für wiederverwendbare Funktionalitäten in objekt-orientierten System sind Klassenvererbung und Objektstruktur. Klassenvererbung sorgt dafür, dass man die Implementation einer Klasse im Hinblick auf eine andere definiert. Wiederverwendung durch Unterklassen wird oft als „White-box Wiederverwendung“ bezeichnet. Der Begriff „White-box“ bezieht sich auf die Sichtbarkeit, denn mit Vererbung sind die Interna der Elternklasse oftmals für die Unterklassen sichtbar.
Objektstruktur ist eine Alternative zur Klassenvererbung. Hier erhält man neue Funktionalitäten durch zusammenbauen oder zusammensetzen von Objekten, um eine größere Komplexe Funktionalität zu erhalten. Objektstruktur erfordert, dass das Objekt, welches zusammengesetzt ist, wohl definierte Schnittstellen hat. Diese Art der Wiederverwendung nennt man „Black-box Wiederverwendung“, weil keine internen Details der Objekte sichtbar sind. Die Objekte erscheinen nur als „Black-boxes“.
Vererbung und Struktur haben jeweils ihre Vor- und Nachteile. Klassenvererbung macht es einfacher die Implementation so zu modifizieren, dass sie wiederverwendet werden kann. Wenn eine Unterklasse einige aber nicht alle Operationen überschreibt, kann es sich auch auf die Operationen die sie vererbt auswirken, angenommen sie rufen die überschriebenen Operationen auf.
Aber Klassenvererbung hat auch einige Nachteile. Der gravierernste Nachteil ist, dass durch die Vererbung an eine Unterklasse die Details der Elternklassen Implementation offen gelegt werden. Daher wird oftmals gesagt, das die Vererbung die Verkapselung zerstört. Die Implementation einer Unterklasse ist so stark mit der Implementation ihrer Elternklasse Verbunden, das jede Veränderung in der Elternklassen-Implementation erzwingt, dass auch die Unterklasse verändert wird.
Implementationsabhängigkeiten können Probleme verursachen, wenn man versucht eine Unterklasse wiederzuverwenden.

Objektstruktur erfordert, dass Objekte ihre gegenseitigen Schnittstellen respektieren. Das wiederum erfordert sorgfältiges Design der Schnittstellen, damit ein Objekt zusammen mit mehreren anderen Objekten benutzt werden kann. Dadurch das auf die Objekte ausschließlich über ihre Schnittstellen zugegriffen werden kann, wird die Verkapselung gewahrt. Jedes Objekt kann durch ein anderes ersetzt werden, welches vom gleichen Typ ist.
Zusätzlich wird durch die Objektstruktur gewährleistet, dass die Klassen überschaubar bleiben.
Andererseits besitzen objekt-orientierte Designs die mit Objektstruktur arbeiten mehr Objekte, und das Systemverhalten hängt von ihrer Beziehung zu einander ab, und ist nicht in einer Klasse definiert.
Im allgemeinen sollte man jedoch Objektstruktur vor der Vererbung vorziehen. Viele Designer strapazieren die Vererbung zu sehr. Daher wird in Design Patterns wieder und wieder Objektstruktur angewendet.

Für Veränderungen designen:
Der Schlüssel zur Maximierung der Wiederverwendbarkeit liegt darin neue Anforderungen und Veränderungen der bestehenden Anforderungen vorauszuahnen und das System so zu designen, dass es sich dementsprechend entwickeln kann.
Um ein System so zu designen, dass es robust gegenüber solchen Veränderungen ist, muss man bedenken wie sich das System in seiner Lebenszeit verändern können muss. Ein System welches solche möglicherweise notwendigen Veränderung nicht

beachtet, riskiert, dass in der Zukunft eine große Neugestaltung notwendig ist. Solche Veränderungen können Klassenneugestaltung und –neuimplementation, Clientmodifikationen und erneutes Testen einschließen. Eine Neugestaltung wirkt sich auf viele Teile des Softwaresystems aus und unvorhergesehene Veränderungen sind immer sehr teuer.
Design Patterns helfen dem auszuweichen, indem gewährleistet wird, dass sich ein System auf spezielle Weisen verändern kann. Jedes Design Pattern lässt einige Aspekte der Systemstruktur unabhängig von anderen Aspekten abwandeln, dadurch wird das System robuster gegenüber speziellen Arten von Veränderungen.
Im nachfolgenden sollen einige allgemeine Ursachen für Neuentwicklung erklärt werden:

- *Ein Objekt kreieren durch deutliches bestimmen einer Klasse.* Das Bestimmen eines Klassennamens, sobald man ein Objekt kreiert, verpflichtet zu einer speziellen Implementation anstatt zu einer speziellen Schnittstelle. Diese Verpflichtung kann zukünftige Veränderungen verkomplizieren.
  Um dies zu vermeiden, sollten Objekte auf indirektem Wege kreiert werden.
- *Abhängigkeit von speziellen Operationen.* Wenn eine spezielle Operation bestimmt wird, verpflichtet man sich zu einem Weg einen Request zu erfüllen. Durch Umgehen von hart codierten Requests, macht man es einfacher die Art und Weise wie ein Request erfüllt wird zu verändern.
- *Abhängigkeiten zu Hard- und Softwareplattformen.* Externe Betriebssystemsschnittstellen und Programmierschnittstellen sind verschieden auf unterschiedlichen Hard- und Softwareplattformen. Software die von einer speziellen Plattform abhängig ist, ist schwerer von einer Plattform auf eine andere Plattform zu portieren. Es kann auch schwerer sein sie auf ihrer ursprünglichen Plattform aktuell zu halten. Daher ist es sehr wichtig ein System so zu designen, dass seine Plattformabhängigkeit reduziert wird.
- *Abhängigkeiten zu Objektrepräsentationen oder Implementationen.* Clients die wissen, wie ein Objekt repräsentiert, gespeichert, lokalisiert oder implementiert ist, könnten verändert werden müssen, wenn das Objekt sich ändert. Durch verstecken dieser Informationen vor dem Client wird dieses Problem gelöst.
- *Algorithmische Abhängigkeiten.* Algorithmen werden während der Entwicklung und Wiederverwendung oft erweitert, optimiert und ersetzt. Objekte die von einem Algorithmus abhängen, müssen auch verändert werden, wenn sich der Algorithmus verändert. Daher sollten Algorithmen, die wahrscheinlich verändert werden müssen, isoliert werden.
- *Enge Kopplung.* Klassen die sehr eng aneinander gekoppelt sind, sind in Isolation sehr schwer wiederzuverwenden, solange sie voneinander abhängen. Enge Kopplung führt zu Systemen, in denen man eine Klasse nicht verändern oder löschen kann, ohne viele andere Klassen zu verstehen und zu ändern. Das System wird zu einer dichten Masse, die schwer zu verstehen, zu portieren und zu pflegen ist. Löst man sich von dem Koppeln, wächst die Wahrscheinlichkeit, das eine Klasse von sich selbst wiederverwendet werden kann und das ein System einfacher erlernt, portiert, modifiziert und erweitert werden kann. Design Patterns verwenden Techniken wie abstrakte Kopplung und Überlagerung um locker gekoppelte Systeme zu fördern.
- *Erweiterte Funktionalitäten durch Unterklassen.* Die Anpassung eines Objektes durch Unterklassen ist oft nicht einfach. Das definieren einer Unterklasse erfordert ein tiefes Verständnis der Elternklasse. Zum Beispiel,

das Überschreiben einer Operation könnte das Überschreiben einer anderen erfordern. Eine überschriebene Operation könnte gefordert sein, eine vererbte Operation aufzurufen. Und Unterklassenbildung könnte zu einer Explosion aus Klassen führen, weil man viele neue Unterklassen einbringen muss, um ein simple Erweiterung zu bewirken. Viele Design Patterns produzieren Designs, in welchen man angepasste Funktionalitäten durch die Definierung einer Unterklasse einbringen kann.

- *Unfähigkeit Klassen in geeigneter Weise zu ändern.* Manchmal muss man Klassen die nicht in geeigneter Weise modifiziert werden können modifizieren. Vielleicht braucht man den Quellcode und hat ihn nicht. Oder eventuell würde jede Veränderung eine Modifizierung vieler existierender Unterklassen erfordern. Design Patterns bieten Wege an, Klassen unter solchen Umständen zu modifizieren.

Diese Beispiele spiegeln die Flexibilität wieder die man mit Hilfe von Design Patterns in die Software einbauen kann. Wie entscheidend solche Flexibilität ist, hängt von der Art der Software ab, die man entwickelt.

2 Socket Factories

2.1 Wiederholung: Socket

Ein Socket bezeichnet einen Endpunkt einer Punkt-zu-Punkt Verbindung zwischen zwei Kommunikationspartnern. Betrachtet wird hierbei die Schnittstelle eines Client- und eines Serverprogramms.
Ein Server stellt hierbei einen Port (Stichwort: TCP) zur Verfügung an den ein Socket gebunden ist. Über den Port kann ein Clientprogramm eine Verbindung zu einem Server für eine Datenübertragung herstellen. Der Client benötigt zusätzlich zur Portnummer den Namen des Servers. Befinden sich beide Kommunikationspartner (Server und Client) auf demselben Rechner kann als Servername „localhost“ angeben werden.
Nach Starten des Servers fragt er den Socket ab, ob ein Client eine Verbindung herstellen will. Besteht bereits eine Verbindung zu einem Client kann der Server einen weiteren Socket bzw. Port generieren und über diesen eine zweite Verbindung herstellen.

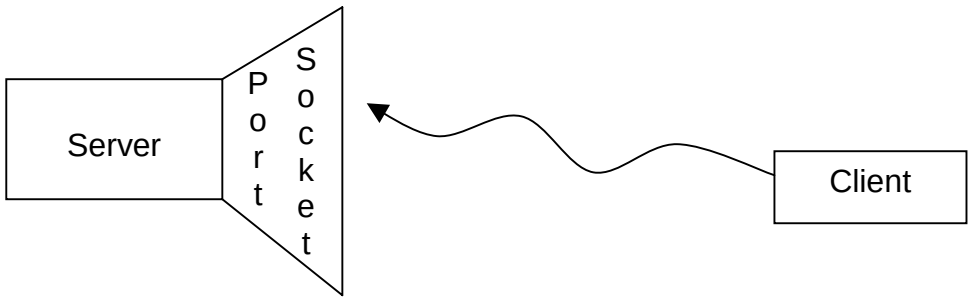

Abbildung 1: Verbindung über Socket

Um eine Socketverbindung realisieren zu können, wird das Paket *java.net* benötigt. Dieses Paket beinhaltet die Klasse *socket* mit der Clients sich mit einem Server verbinden können. Mit der Klasse *serversocket* lassen sich Server entwickeln, die den Clients Ports für eine Verbindung anbieten.

2.2 Socket Factory

Socket Factories gehören zu der Gruppe der Design Pattern über die sich Muster für Sockets erstellen lassen. Sockets, die über eine Socket Factory erstellt wurden, unterscheiden sich weder funktional noch im Aufbau von „manuell“ erzeugen Sockets. In einer Socket Factory werden nur Voreinstellungen für eine bestimmte Art von Sockettypen vorgenommen. Somit wird das Erstellen von Sockets (z.B. Serversockets) auf einen Methodenaufruf verringert und somit stark vereinfacht.
Für das Arbeiten mit Socket Factories wird das Interface *SocketFactory* sowie die Klassen java.net.ServerSocket für serverseitige Sockets und java.net.Socket für clientseitige Sockets benötigt.
Hier nun die Methode, mit der ein serverseitiger Socket erzeugt wird.

CreateServerSocket *(int port[, int backlog]*
[,java.net.InetAddress bindAddr])
throws java.io.IOException

- port = an diesen Port wird der Socket gebunden
- backlog = maximale Anzahl der parallel-laufenden Anfragen
- bindAddr = an diese Adresse wird der ServerSocket gebunden (mehrere Netzwerkkarten)

Die Methode gibt im Erfolgsfall einen neuen ServerSocket zurück. Im Fehlerfall wird eine „Standard"-Ausnahme (java.io.IOException) geworfen.

Die Methode zum Erzeugen eines clientseitigen Socket ist ähnlich aufgebaut. Zusätzlich ist es notwendig die Adresse bzw. der Name des Servers sowie den Port, an den der Socket gebunden ist, anzugeben.
Hierzu ein Beispiel:

CreateSocket *(java.net.InetAddress addr | java.lang.String host,*
int port)
throws java.net.UnknownHostException, java.io.IOException

Wie oben schon angesprochen kann dieser Methode entweder die IP-Adresse eines Servers oder dessen Namen als Argument übergeben werden.

3 Socket Factories für verschiedene Socket-Typen

In diesem Abschnitt soll auf das eigentliche Erzeugen von Socket Factories sowie auf die verschiedenen Arten eingegangen werden.
Grundsätzlich sollte als erstes geklärt werden welche Aufgabe eine Verbindung zwischen einem Server und einem Client erfüllen soll. Bei der Übermittlung von Daten, die z.B. einen gewissen Geheimhaltungswert haben sollte ein SSL-Typ verwendet werden. Sollen hingegen große Datenmengen übermittelt werden (z.B. Filme) ist es sinnvoll einen Typ zu verwenden, der eine Kompression beinhaltet.

3.1 Singletype

Diese Socket Factories erzeugen nur eine Socketart. Hier ist es wichtig sich, wie oben bereits erwähnt, Gedanken um das Einsatzgebiet zu machen. Bei Verwendung dieser Socket Factory werden später nur Sockets erstellt, die nur für diese eine Anwendung vorgesehen sind.

Beispiel: Single Type (Kompression)

```
/**** SERVER-SEITE ****/
import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class CompressionServerSocketFactory
    implements RMIServerSocketFactory, Serializable
{
    public ServerSocket createServerSocket(int port)
        throws IOException
```

```
    {
        CompressionServerSocket server =
            new CompressionServerSocket(port);
        return server;
    }
}
/****/

/**** CLIENT-SEITE ****/

import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class CompressionClientSocketFactory
    implements RMIClientSocketFactory, Serializable
{
    public Socket createSocket(String host, int port)
        throws IOException
    {
        CompressionSocket socket =
           new CompressionSocket(host, port);
        return socket;
    }
}
/****/
```

Mit folgender Methode können Sockets nun über die obige Socket Factory erzeugt werden (siehe auch Übungen).

```
//Server-Seite
MyServerSocket = createServerSocket(1717);

//Client-Seite
MySocket = createSocket(„localhost“, 1717);
```

3.2 Multitype

Anders als bei den Socket Factories, die Sockets des Single-Typs erstellen, können hier, wie der Name schon sagt, verschiedene Arten von Sockets erstellen werden.

```
/****SERVER-SEITE****/
import java.io.*;
import java.net.*;
import java.rmi.server.*;
public class MultiServerSocketFactory
    implements RMIServerSocketFactory, Serializable
{
    private static RMISocketFactory defaultFactory =
        RMISocketFactory.getDefaultSocketFactory();
    private String protocol;
    private byte[] data;

    public MultiServerSocketFactory(String protocol,  byte[]
    data)
```

```
    {
        this.protocol = protocol;
        this.data = data;
    }

    public ServerSocket createServerSocket(int port)
        throws IOException
    {
        if (protocol.equals("compression")) {
            return new CompressionServerSocket(port);
    }
    else if (protocol.equals("xor")) {
        if (data == null || data.length != 1)
            throw new IOException("invalid argument  for XOR
    protocol");
        return new XorServerSocket(port, data[0]);
   }
   return defaultFactory.createServerSocket(port);
}
/****/

/**** CLIENT-SEITE****/
import java.io.*;
import java.net.*;
import java.rmi.server.*;
public class MultiClientSocketFactory
    implements RMIClientSocketFactory, Serializable
{
    private static RMISocketFactory defaultFactory =
        RMISocketFactory.getDefaultSocketFactory();

    private String protocol;
    private byte[] data;

    public MultiClientSocketFactory(String protocol,
    byte[] data) {
        this.protocol = protocol;
        this.data = data;
    }

    public Socket createSocket(String host, int port)
        throws IOException
    {
        if (protocol.equals("compression")) {
            return new CompressionSocket(host, port);
        } else if (protocol.equals("xor")) {
            if (data == null || data.length != 1)
            throw new IOException("invalid argument for   XOR
     protocol");
         return new XorSocket(host, port, data[0]);

        }
        return defaultFactory.createSocket(host, port);
    }
}
/****/
```

Über die oben angegebene Socket Factory *MultiClientSocketFactory* lassen sich drei verschiedene Arten von Sockets erzeugen. Diese Arten werden über den übergebenen Parameter *protocol* unterschieden. Wird kein Auswahl vorgenommen findet eine Erzeugung über die Standardregel (java.net.*) statt.

3.3 Customtype

Bevor Socket Factories für Costum Type Sockets benutzt werden können muss eine sogenannte policy-Datei geschrieben werden. In diesem Beispiel soll es genügen jedem Benutzer von jedem Ort aus alle Rechte zu geben. Diese Datei sollte aber unbedingt für einen ernsthaften Einsatz umgeschrieben werden.

```
grant {
    permission java.security.AllPermission;
}
```

Die Erzeugung geschieht über ein Remote-Object, das den Konstruktor *UnicastRemoteObject* aufruft. Dieser Konstruktor verwendet die Parameter der *RMIClientSocketFactory* sowie der *RMIServerSocketFactory*.

```
protected UnicastRemoteObject(int port,
                              RMIClientSocketFactory client_sf,
                              RMIServerSocketFactory server_sf)


public CustomImpl(String protocol, byte [] pattern)
       throws RemoteException
    {
     super(0, new MultiClientSocketFactory(protocol, pattern),
           new MultiServerSocketFactory(protocol, pattern));
    }
```

Der hier angegebene Typ wird für die Erzeugung der Sockets auf der Server- und Clientseite verwendet.

4 Literatur / Abbildungen

Literatur

Design Pattern: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Addison Wesley 1995

Abbildung