In diesem Kapitel:

- *Einleitung*
- *Distributed Computing mittels RMI*
- *Parameter in RMI*
- *Client-seitige Callbacks*
- *Automatische Distribution*
- *Distribured Garbage Collection*
- *Berechnungsserver*
- *Eigene SocketFactory*
- *Aktivierung von remote Objekten*
- *Adapter Emtwurfsmuster*
- *Persistenz und Marshalled Objekte*
- *Dynamisches Herunterladen von Code*
- *Factory Emtwurfsmuster*
- *RMI over IIOP*
- *alternative RMI Ansätze*

2. *Java Remote Methode Invocation - RMI Praxis*

2.1. EInleitung

Nachdem wir uns mit sehr viel Theorie, der Spezfallsikation von RMI befasst haben, wollen wir nun RMI in voller Aktion sehen. Dazu entwickeln wir verschiedene Programme, welche die einzelnen Aspekte der Spezfallsikation illustrieren.

In verteilten Objekt Orientierte Systeme geht es um die Kommunikation von Objekten, die in verschiedenen virtuellen Maschinen aktiv sind. RMI, Remote Methode Invocation, vereinfacht den Aufruf einer entfernten Methode: es scheint als ob die Methode lokal aufgerufen würde. RMI schafft ein lokales "Proxy-Objekt", welches wie das entfernte Objekt reagiert. In diesem Kapitel wollen wir Vorteile und Nachteile dieser Technik kennen lernen.

2.1.1. Groblernziele

Nach dem Durcharbeiten dieses Moduls sollten Sie in der Lage sein:

- zu beurteilen, was Distributed Computing mit Java bedeuten kann
- die RMI Architektur zu verstehen und RMI einzusetzen
- das verteilte Objektmodell, welches in RMI definiert und unterstützt wird, zu verstehen.

Nach dem Durcharbeiten der Übungsbeispiele sollten Sie in der Lage sein:

- Objekte zwischen JVMs hin und her zu senden, mittels Objektserialisierung
- neue Methoden für die Verteilung und die Installation von Java Programmen zu kennen und einzusetzen.

2.2. Einführung in Distributed Computing mittels RMI

Remote Method Invocation (RMI) Technologie, die zuerst im JDK 1.1 eingeführt wurde, hebt die Netzwerkprogrammierung auf eine höhere Ebene. Obschon RMI relativ leicht zu benutzen ist, ist die Technik mächtig und ermöglicht dem Java Programmierer völlig neue Konzepte, das Paradigma der "Verteilten Objekte".

2.2.1. Entwurfsziele

Eines der Hauptziele der RMI Designer (Jim Waldo et al) war es, ein System zu entwickeln, mit dessen Hilfe der Java Programmierer verteilte Anwendungen entwickeln kann, welche die selbe Syntax und Semantik wie nicht-verteilte Systeme verwenden. Dazu mussten die Entwickler / Designer von RMI die Art und Weise, wie Java Klassen und Objekte in einer einzigen Java Virtuellen Maschine arbeiten, abbilden auf die Art und Weise, wie Klassen und Objekte in einer verteilten Umgebung, also mehreren JVMs arbeiten.

In diesem Abschnitt führen wir die RMI Architektur ein, aus dem Blickwinkel verteilte oder remote / entfernte Java Objekte, und grenzen diese Technik ab gegen Systeme mit lediglich lokalen Java Objekten. Die RMI Architektur definiert, wie Objekte sich verhalten, wie und wann Ausnahmen auftreten können und wie der Speicherplatz verwaltet wird, wie Parameter übergeben werden und von entfernten Methoden wieder zurück geliefert werden.

2.2.2. Vergleich verteilter mit nichtverteilten Java Programmen

Die RMI Architektur versuchte den Einsatz verteilter Java Objekte möglichst ähnlich zu gestalten, wie der Einsatz lokaler Java Objekte. Das ist den Architekten im Prinzip auch gelungen. Einige wichtige Unterschiede sind in der folgenden Tabelle enthalten. Sie brauchen nicht gleich alle Begriffe, die in der Tabelle auftauchen, zu verstehen; die in der Tabelle aufgeführten Themen werden im Folgenden genauer besprochen werden.

	Lokale Objekte	Remote Objekte
Objekt Definition	ein lokales Objekt wird mit Hilfe einer Java Klasse definiert.	das exportierte, nach aussen sichtbare Verhalten eines remote Objekts wird mit Hilfe einer Schnittstelle definiert. Diese muss das Remote interface erweitern.
Objekt Implementation	ein lokales Objekt wird durch seine Java Klasse implementiert.	das Verhalten eines remote Objekts wird durch eine Java Klasse, die das remote Interface implementiert, zur Verfügung gestellt.
Objekt Kreation	eine neue Instanz eines lokalen Objekts wird mit dem new Operator kreiert.	eine neue Instanz eines remote Objekts wird auf dem Host Rechner mit dem new Operator kreiert. Ein Client kann nicht direkt ein neues remote Objekt kreieren, ausser der Client verwendet die in Java 2 eingeführte Remote Objekt Aktivierung.
Objekt Zugriff	auf ein lokales Objekt wird direkt mit Hilfe einer Objektreferenzvariable zugegrfallsfen.	auf ein remote Objekt wird mit Hilfe einer Objektreferenzvariable zugegrfallsfen, die auf eine Proxy Stub (Stumpf, Stummel) des remote Interface zeigt.
Referenzen	in einer einfachen JVM, zeigt eine Objekt Referenz direkt auf ein Objekt auf dem Heap.	eine "remote Referenz" ist ein Zeiger / pointer auf ein Proxy Objekt (ein "Stub") im lokalen Heap. Dieser Stub enthält Informationen, die es ihm erlauben auf ein remote Objekt zuzugrefallsen, welches die Implementation der Methoden enthält.
Aktive Referenzen	in einer einfachen JVM wird ein Objekt als "lebendig / alive" angesehen, falls midestens eine Referenz darauf zeigt.	in einer verteilten Umgebung können remote JVMs abstürzen und Netzwerk- verbindungen können verloren gehen. Ein remote Objekt wird als aktive remote Referenz betrachtet, falls darauf innerhalb eines bestimmten Zeitraumes (der lease Periode) zugegrfallsfen wurde. falls

		alle remote Referenzen explizit abgebrochen wurden, oder falls die Leasingdauer aller remote Referenzen abgelaufen ist, dann steht ein remote Objekt dem distributed Garbage Collectior zur verfügung, wird also über kurz oder lang vernichtet.
Finalization	falls an Objekt die finalize() Methode implementiert, wird diese aufgerufen, bevor ein Objekt vom Garbage Collector weggeräumt wird.	falls ein remote Objekt das UnReferenced Interface implementiert (aus dem java.rmi.server Package), dann wird die unReferenced Methode dieses Interfaces aufgerufen, sobald alle remote Referenzen aufgehoben wurden.
Garbage Collection	sobald alle lokalen Referenzen auf ein Objekt aufgehoben wurden wird ein Objekt ein Kandidat für den Garbage Collector.	Der distributed Garbage Collector arbeitet mit dem lokalen Garbage Collector zusammen: falls keine remote Referenzen mehr vorhanden sind und alle lokalen Referenzen auf ein remote Objekt aufgehoben wurden, dann wird das Objekt ein Kandidat für Garbage Collection.
Exceptions	Exceptions sind entweder Runtime / Laufzeit Exceptions oder generelle Exceptions. Der Java Compiler zwingt ein Programm alle Exceptions abzufangen.	RMI zwingt Programme alle möglichen RemoteException abzufangen, die Objekte werfen können. Damit soll die Robustheit der verteilten Applikationen gewährleistet werden.

2.2.3. Java RMI Architektur

Das Entwurfsziel der RMI Architektur war, ein Java distributed Objekt-modell zu entwickeln, welches sich auf natürliche Art und Weise in die Java Programmiersprache und das lokale Objektmodell integriert. Die RMI Architekten waren erfolgreich: sie schufen ein System, welches die Sicherheit und Robustheit der Java Architektur auf das distributed computing Paradigma erweitert.

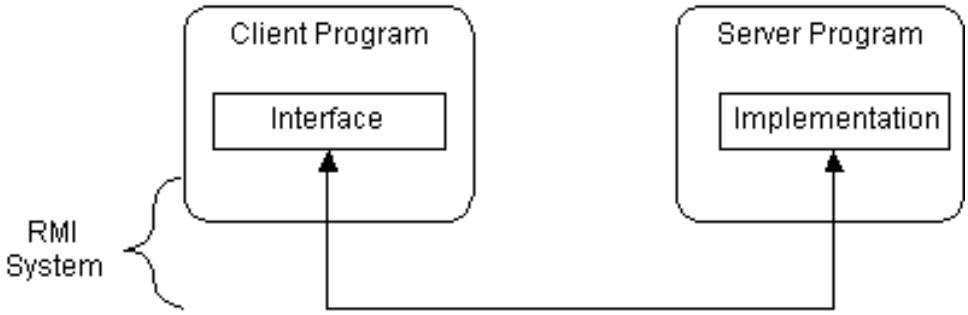
2.2.3.1. Interfaces: das Kernstück von RMI

Die RMI Architektur basiert auf einem wichtigen Prinzip: *die Definition des Verhaltens und die Implementation des Verhaltens sind zwei separate Konzepte*. RMI erlaubt es, den Programmcode, mit dem das Verhalten definiert wird und der Programmcode der dieses Verhalten implementiert zu trennen und auf unterschiedlichen JVMs auszuführen.

Dies passt gut zu den typischen Anforderungen an ein verteiltes System: die Clients definieren einen Dienst / Service während die Server sich auf das Zurverfügungstellen des Services konzentrieren.

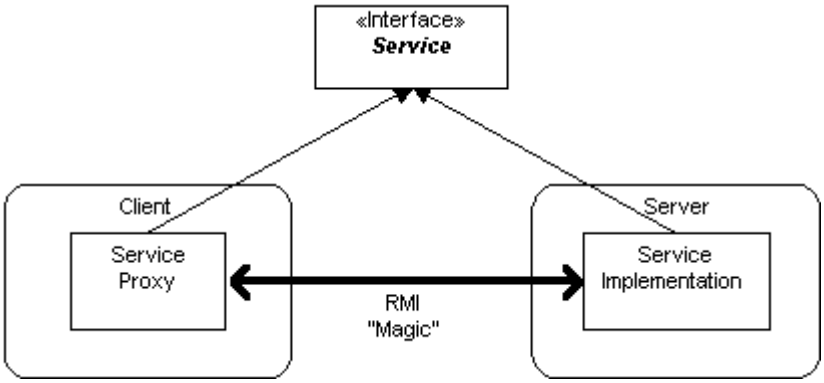
In RMI wird die Definition eines remote Service mittels eines Java Interface spezifiziert. Die Implementation des remote Service wird in Form einer Klasse programmiert. Daher ist es ganz wichtig immer daran zu denken, dass in RMI *Interfaces das Verhalten spezifizieren* und, *Klassens definieren die Implementation*.

Die folgende Grafik illustriert diese Aufteilung.



Aber ein Java Interface enthält keinen ausführbaren Programmcode. RMI untersützt zwei Klassen, die das selbe Interface implementieren.

- die erste Klasse implementiert das Verhalten und läuft auf dem Server.
- die zweite Klasse arbeitet als Proxy für den remote Service und läuft auf dem Client



Ein Client ruft eine Methode des Proxyobjekts auf, RMI sendet den Aufruf andie entfernte / remote JVM und schickt den Aufruf an die Implementation. Allfällige Rückgabewerte werden entlang des selben Weges an das Proxyobjekt gesandt und von dort an das Client Programm.

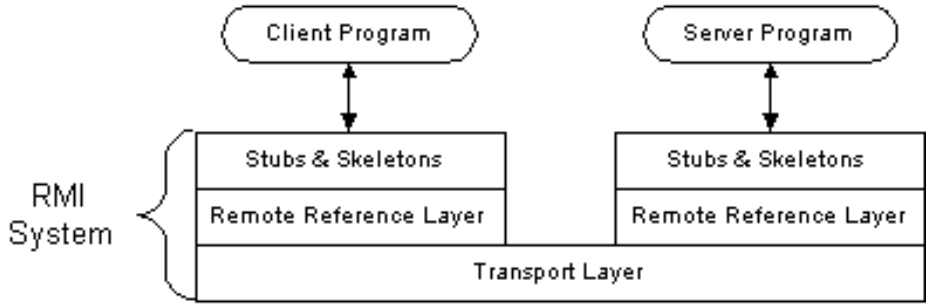
2.2.3.2. RMI Architektur Layers

Nachdem wir den grundsätzlichen Mechanismus dargestellt haben, wollen wir nun etwas in die Tiefe gehen und uns einige Implementationsdetails anschauen.

Die Implementation verwendet im Wesentlichen drei Ebenen, Layers:

- der erste Layer ist der Stub und Skeleton Layer, auf dem der Entwickler aufbaut.
  Dieser Layer fängt die Methodenaufrufe des Clients auf und leitet sie an eien RMI Dienst weiter.
- der nächste Layer ist der Remote Reference Layer.
  Dieser Layer kann die Referenzen eines Clients auf ein remote (Service) Objekt interpretieren. In JDK1.1 wurde mit Hilfe dieses Layers der Client mit dem Service verknüpft. Die Verbindung ist eine Unicastverbindung. In Java 2 wurde dieser Layer durch die Möglichkeit der Objektaktivierung erweitert. Diese erlaubt die Aktivierung von schlafenden (dormand) Objekten bzw. Services. (*Remote Object Activation)*.

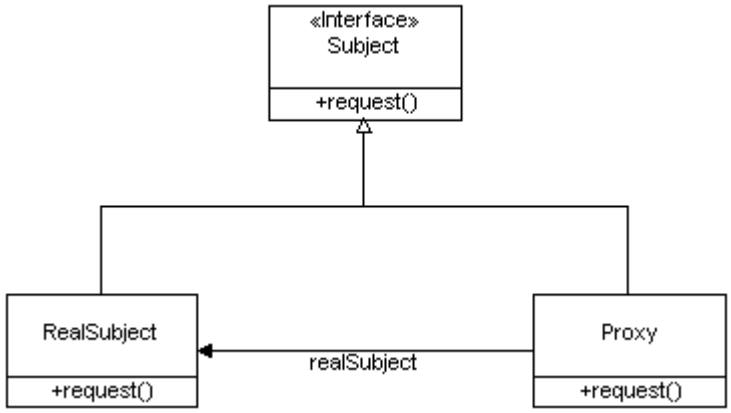
- der Transportlayer basiert auf TCP / IP Verbindungen zwischen den Maschinen im Netzwerk. Die Grundfunktion ist also die Verbindung / Connectivity. Zudem werden einige Firewall Penetrationsstratgien implementiert.



Die Layer Architektur erlaubt es, jeden Layer einzeln zu erweitern oder zu ersetzen, ohne den Rest des Systems zu sehr zu verändern. Beispielsweise könnte der Transportlayer durch einen UDP / IP Layer ersetzt werden, ohne dass die darüberliegenden Layer davon betroffen würden.

2.2.3.2.1. Stub and Skeleton Layer

Der Stub und Skeleton Layer von RMI unterhalb der Java Programmierebene verwendet das Proxy Design Pattern / Entwurfsmuster, wie es beispielsweise im Buch *Design Patterns* von Gamma, Helm, Johnson und Vlissides. Im Proxy Pattern wird ein Objekt aus einem Kontext durch ein anderes (das Proxy Objekt) in einem separaten Kontext dargestellt (des Server Objekt in der Server JVM wird als Proxy Objekt in der Client JVM dargestellt).Der Proxy weiss, wie Methodenaufrufe zwischen den beteiligten Objekten ausgetauscht werden müssen.



Dieses Muster wird im folgenden Bild schematisch dargestellt:

In RMI übernimmt die Stubklasse die Rolle des Proxies; und die entfernte Serviceimplementation übernimmt die Rolle des echten Objekts, des RealSubject.

Ein Skeleton ist eine Hilfsklasse, helper Klasse, welche vom RMI System automatisch generiert wird und ab Java 2 nicht mehr benötigt wird. Das Skeleton weiss, wie mit dem Stub über eine RMI Verbindung kommuniziert wird. Das Skeleton hält die Kommunikation mit dem Stub aufrecht; es liest die Parameter für die Methodenaufrufe, führt die Aufrufe des entfernten Services (Objekt) aus, akzeptiert die Rückgabewerte und schreibt diese dann an den Stub zurück.

Die Skeleton Klasse wurde in Java 2 wegen einer Neudefinition des Wire Protokolls überflüssig. RMI verwendet das Reflection API, um die Verbindung zum entfernten Objekt herzustellen. Man muss sich also nur noch um den Stub bemühen.

Zudem hat IBM, zusammen mit Sun, eine Implementation von RMI vorgestellt, welche das IIOP (Internet Inter ORB Protokoll) verwendet. Dieses Protokoll ist standardisiert und wird auch von CORBA, dem Common Object Request Broker der OMG (Object Management Group) eingesetzt. Damit werden die Einsatzmöglichkeiten von Java RMI wesentlich erweitert.

2.2.3.2.2. Remote Referenz Layer

Der Remote Reference Layer (RRL) definiert und unterstützt die Aufrufsemantik der RMI Verbindung. Dieser Layer stellt ein RemoteRef Objekt zur Verfügung, welches die Verbindung zur remote Serviceimplementation, dem remote Objekt, zur Verfügung stellt.

Die Stub Objekte verwenden die invoke() Methode in RemoteRef um die Methodenaufrufe weiterzuleiten. Das RemoteRef Objekt weiss dann, wie der Aufruf konkret auszuführen ist: es versteht die Aufrufsemantik.

In der JDK 1.1 implementation von RMI gab es nur eine Möglichkeit für Clients mit der remote Serviveimplementation Verbindung aufzunehmen: eine unicast, Punkt zu Punkt Verbindung.

Bevor ein Client einen remote Service benutzen kann, muss der remote Service auf dem Server instanziert und an das RMI System exportiert werden (falls es sich um den primären Service handelt, muss dieser benannt und in die RMI Registry eingetragen werden).

Die Java 2 SDK Implementation von RMI verfügt über zusätzliche Möglichkeiten, eine neue Semantik für die Client-Server Verbindung. In dieser neuen Version unterstützt RMI aktivierbare remote Objekte. Falls ein Methodenaufruf für ein aktivierbares Objekt an den Proxy geschickt wird, bestimmt das RMI System als erstes, ob die remote Serviceimplementation für das Objekt schlafend ist. Falls das Objekt / der Dienst schläft, dann instanziert RMI das Objekt und liest seinen Zustand aus dem Speicher (Festplatte / persistenter Speicher) und stellt somit den Zustand des Objekts vor dem Schlaf wieder her. Falls ein aktivierbares Objekt im Hauptspeicher ist, verhält es sich wie ein remote Objekt aus der Java Version JDK1.1.

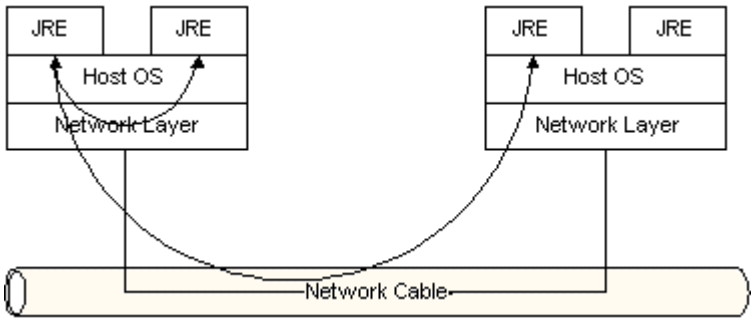
Es sind auch andere Semantiken denkbar. Beispielsweise könnte ein Proxy einen Methodenaufruf mit Hilfe von Multicast an mehrere Implementationen gleichzeitig senden und mit der ersten Antwort weiter arbeiten (FIFO). Damit würden die Antwortzeiten unter Umständen reduziert. Zudem würde die Verfügbarkeit des Systems erhöht.

2.2.3.2.3. Transport Layer

Der Transport Layer stellt die Verbindung zwischen den Virtuellen Maschinen her. Alle Verbindungen basieren auf Streams Netzwerkverbindungen, auf der Basis von TCP/IP.

Die Verbindung benutzt selbst dann TCP/IP, falls beide JVMs auf dem selben Rechner laufen. Es ist also zwingend, dass TCP/IP auf dem Rechner, auf dem RMI laufen soll, installiert wird und funktioniert.

Das folgende Diagramm zeigt schematisch diese Verbindung zweier JVMs über TCP/IP.



TCP/IP stellt eine dauerhafte (persistente), stream-basierte Verbindung zwischen zwei Maschinen auf Grund der IP Adresse und der Portnummer an beiden Enden her. In der Regel wird man die IP Nummer mittels DNS auflösen, also mit Namen arbeiten, statt mit IP Nummern.

Eine Verbindung würde also beispielsweise zwischen `joller.org:3461` und `joller.com:3224` aufgebaut. In der gegenwärtigen Implementation wird TCP / IP als Basis für die RMI Maschinen-Maschinen Kommunikation eingesetzt. Oberhalb von TCP wurde ein Wire Protokoll spezifiziert, das sogenannte Java Remote Method Protocol (JRMP). JRMP ist ein proprietäres, stream-basiertes Protokoll, welches nur teilweise `spezifiziert` wurde. Es existiert nun in zwei Versionen: die erste Version wurde mit JDK 1.1 herausgegeben; die zweite Version mit der Version Java 2. Die erste version verlangt den Einsatz der Skeletons; die zweite Version verzichtet darauf. Es gibt auch proprietäre Implementationen des Wire Protokolls, beispielsweise von BEA Weblogic und von NinjaRMI: beide verwenden nicht JRMP. ObjectSpace, ein weiterer Anbieter, erkennt JRMP, verwendet aber intern ein eigenes Protokoll.

IBM (und Sun) arbeiten an der Version `RMI-IIOP`, welche ab JDK1.3 zur Verfügung steht und separat von SUNs Java Web oder IBMs Alpha Web heruntergeladen werden kann. Interessant an RMI- IIOP ist, dass diese Version das von der `Object Management Group` (OMG) standardisierte Internet Inter-ORB Protocol, IIOP, verwendet, um zwischen Clients und Servern zu kommunizieren.

Die OMG ist eine Gruppe von mehr als 800 Mitgliedern, welche gemeinsame, herstellerneutrale Architekturen und Standards definieren und publizieren. Ein Beispiel ist die verteilte Objekt Architektur, genannt Common Object Request Broker Architectur (CORBA). CORBA Objekt Request Broker (ORB) Clients und Server kommunizieren mittels IIOP. Mit der Adaption dieser Objects-by-Value CORBA Erweiterung und dem Java to IDL Mapping wurde die Grundlage für eine direkte RMI zu CORBA Integration gelegt. Diese neue (RMI-IIOP) Implementation unterstützt die meisten RMI Feature, mit Ausnahme von:

- `java.rmi.server.RMISocketFactory`
- `UnicastRemoteObjekt`
- `UnReferenenced`
- die DGC Interfaces

Der RMI Transport Layer (RTL) wurde so definiert, dass selbst im Falle von Netzwerkfehlern versucht wird, eine Client Server Verbindung aufzubauen.

Obschon auf dem Transportlayer in der Regel versucht wird, mehrere TCP/IP Verbindungen zu nutzen, erlauben einige Netzwerke lediglich eine einzelne TCP/IP Verbindung (beispielsweise kann ein Browser auf eine und nur eine Verbindung eines Applets mit seinem Server bestehen). In diesen Fällen verwendet RMI ein Multiplexing auf dem Transportlayer, um mehrere virtuelle Verbindungen mittels einer einzigen TCP/IP Verbindung realisieren zu können.

2.2.4. Namensgebung für Remote Objekte

Eine der am meisten gestellten Fragen in der RMI Gemeinschaft ist die Frage : "wie findet ein Client einen RMI remote Service?". Die Antwort ist denkbar einfach: RMI verwendet einen Namensdienst oder einen Verzeichnisdienst. Diese Antwort erscheint auf den ersten Blick wie ein Zirkelschluss: wie findet der RMI Client denn den Namens- oder Verzeichnisdienst? Die Antwort darauf ist: mittels eines bekannten Standarddienstes an einem Standardport (bekannt heisst dabei: bekannt innerhalb der Organisation, also nicht global).

RMI kann für den Suchdienst unterschiedliche Verzeichnisdienste einsetzen, beispielsweise das Java Naming and Directory Interface (JNDI). RMI enthält, der Einfachheit halber, einen einfachen Dienst, die RMI Registry, rmiregistry. Die RMI Registry läuft auf jeder Maschine, die Host / Server Objekte anbietet, standardmässig auf Port 1099.

Der zeitliche Ablauf auf dem Server zum Kreieren eines Service Objekts geschieht schrittweise: zuerst wird ein lokales Objekt kreiert, welches den Dienst implementiert. Dann wird das Objekt nach RMI exportiert. Nach dem Exportieren kreiert RMI einen Listener Service, der auf Anfragen von Clients des Services wartet.Nach dem Export registriert der Server das Objekt in der RMI Registry unter einem öffentlichen Namen.

Auf der Client Seite wird mittels der statischen Klasse `Naming` auf die RMI Registry zugegriffen. Die Klasse enthält eine Methode `lookup()`, mit deren Hilfe ein Client die Registry abfragen kann. Die **`lookup()`** Methode akzeptiert eine URL als Parameter. Diese URL spezifiziert den Server Hostnamen und den Namen des gewünschten Services. Die Methode liefert eine remote Referenz zum Serverobjekt. Die URL ist von der Form / Syntax:

```
rmi://<host_name>
      [:<name_service_port>]
            /<service_name>
```

wobei der host_name der Namen des Hosts im lokalen Netzwerk (LAN) oder ein DNS Namen im Internet sein kann. Der name_service_port braucht nur dann anzugeben werden, falls der Namensdienst nicht den Standardport 1099 verwendet.

2.2.5. Praktischer Einsatz von RMI

Nun sollten wir allmählich mit konkreten Beispielen beginnen und RMI austesten! In diesem Abschnitt wollen wir also einen Berechnungsdienst entwickeln, der von Clients genutzt werden kann, mittels RMI.

Ein funktionierendes RMI System besteht aus mehreren Bestandteilen:

- Interface Definitionen für die remote Services
- Implementationen der remote Services
- Stub und Skeleton Dateien
- einen Server auf dem die remote Services installiert werden

- einen RMI Naming Service, mit dessen Hilfe die remote Services gefunden werden.
- einen Class Datei Provider (einen HTTP oder FTP Server)
- ein Client Programm, welches die remote Services ausnutzt

Im nächsten Abschnitt werden Sie ein einfaches RMI System schrittweise aufbauen. Der Einfachheit halber werden wir Client und Server in ein und demselben Verzeichnis abspeichern. Dadurch müssen auch die Class Dateien nicht mit Hilfe eines Servers (HTTP, FTP) heruntergeladen werden. Wie HTTP und FTP Server für die Verteilung von Class Dateien, auch durch eine Firewall, eingesetzt werden können, werden wir noch besprechen.

Falls das RMI System bereits entworfen wurde, besteht die Entwicklung des Systems aus folgenden Schritten:

1. schreiben und übersetzen von Java Code für die Interfaces
2. schreiben und übersetzen von Java Code für die Implementationsklassen
3. generieren von Stubs (und Skeletons, falls JDK 1.1 eingesetzt werden muss). Dies geschieht entweder durch setzen der passenden Schalter im JBuilder (rechte Maustaste auf der Quellcodedatei) oder manuell mittels **`rmic <Implementationsklassen>`**
4. schreiben des Java Codes für einen remote Service (Host Programm)
5. entwickeln des Java Codes für das RMI Client Programm
6. installieren und starten des RMI Systems

2.2.5.1. Interfaces

Als erstes müssen wir also den Java Code für die Schnittstellen / Java Interfaces schreiben. Das Taschenrechner Interface definiert alle remote Möglichkeiten, die der Service anbietet:

```
public interface Taschenrechner
          extends java.rmi.Remote {
    public long addiere(long a, long b)
        throws java.rmi.RemoteException;

    public long subtrahiere(long a, long b)
        throws java.rmi.RemoteException;

    public long multipliziere(long a, long b)
        throws java.rmi.RemoteException;

    public long dividiere(long a, long b)
        throws java.rmi.RemoteException;
}
```

Wichtig ist, dass das Interface Remote erweitert und jede Methodensignatur die Ausnahme RemoteException werfen kann.

Jetzt sollten Sie die obigen Zeilen in ein Verzeichnis kopieren und mit folgendem Befehl übersetzen:

```
>javac Taschenrechner.java
```

Bei der Definition des Interfaces müssen wir die RMI Packages noch nicht importieren.

2.2.5.2. Implementation

Als nächstes schreiben wir die Implementation für den remote Service. Wir nennen die Klasse TaschenrechnerImpl (die Namensgebung ist nicht zufällig gewählt, sondern weist auf die Schnittstelle hin):

```
// jetzt benötigen wir das Remote Package

import java.rmi.Remote;
import java.rmi.RemoteException;

public class TaschenrechnerImpl
    extends java.rmi.server.UnicastRemoteObject
    implements Taschenrechner {

    // die Implementation muss einen expliziten
    // Konstruktor definieren, damit die
    // RemoteException Exception deklariert werden kann

    public TaschenrechnerImpl()
        throws java.rmi.RemoteException {
        super();
    }

    // addiere
    public long add(long a, long b)
        throws java.rmi.RemoteException {
        return a + b;
    }

    // subtrahiere
    public long sub(long a, long b)
        throws java.rmi.RemoteException {
        return a - b;
    }

    // multipliziere
    public long mul(long a, long b)
        throws java.rmi.RemoteException {
        return a * b;
    }

    // dividiere
    public long div(long a, long b)
        throws java.rmi.RemoteException {
        return a / b;
    }
}
```

Kopieren Sie auch dieses Programm in das Verzeichnis und übersetzen Sie es mit der CLASSPATH:

javac **CLASSPATH .** TaschenrechnerImpl.java

Der Punkt hinter dem CLASSPATH besagt, dass im aktuellen Verzeichnis nach Class Dateien, konkret der Class Datei zur Taschenrechner Interface Beschreibung, gesucht werden soll. Die Implementationsklasse verwendet `UnicastRemoteObject` um eine Verbindung zum RMI System herzustellen. In diesem Beispiel verwenden wir die Implementationsklasse als Erweiterung des UnicastRemoteObject. Das ist aber nicht

unbedingt nötig: falls unsere Serviceklasse das UnicastRemoteObject nicht erweitert, kann das Programm die exportObject() Methode einsetzen, um das Objekt an RMI zu binden. Für den Moment halten wir uns aber an die einfacher Art und Weise mit import ...

Falls eine Klasse das UnicastRemoteObject erweitert, muss die Klasse auch einen Konstruktor besitzen, der ein RemoteException Objekt werfen kann. Sobald dieser Konstruktor super(),aufruft, wird Programmcode in UnicastRemoteObject aktiviert, der die RMI Bindung und remote Objektinitialisierung vornimmt.

2.2.5.3. Stubs und Skeletons

Als nächstes müssen wir mit Hilfe des RMI Compiler, rmic, Stub und Skeleton Dateien generieren. Der Compiler wird auf die *Classdatei der Implementationsdatei* angewandt.

```
>rmic -classpath . TaschenrechnerImpl
```

Achtung: beim rmic muss die Option CLASSPATH mit einem Bindestrich und *klein* geschrieben werden!

Nach erfolgreichem Ausführen des RMI Compilers befinden sich folgende Dateien im Verzeichnis:

Taschenrechner.class
Taschenrechner.java
TaschenrechnerImpl.class
TaschenrechnerImpl.java
TaschenrechnerImpl_Skel.class
TaschenrechnerImpl_Stub.class

Falls Sie den RMI Compiler mit der Option -v1.2 starten, wird kein Skeleton generiert. Und hier einige weitere Optionen:

```
Usage: rmic <options> <class names>

wobei <options> umfassen:
  -keep  die generierten Sub und Skeleton Java Dateien nicht löschen
  -keepgenerated (wie "-keep")
  -g              generieren von Debugging Info
  -depend         neu übersetzen von veralteten Dateien (rekursiv)
  -nowarn         generiere keine Warnungen
  -verbose        anzeigen, was der Compiler gerade tut
  -classpath <path>     spezifiziert, wo die Eingabedateien (Java,
                        Class) zu finden sind
  -d <directory>  spezifiziert, wo die generierten Dateien
                  hingeschrieben werden sollen
  -J<runtime flag> Argumentübergabe (an den Java Interpreter)
```

Die Java 2 Platform Version kennt drei weitere Optionen für rmic:

```
  -v1.1  kreiere Stubs/Skeletons für JDK 1.1 Stub Protokollversion
  -vcompat  (Standard)
            kreiere Stubs/Skeletons, die mit beiden JDKs
            JDK 1.1 und Java 2 Stub Protokoll Versionen kompatibel
            sind
```

```
  -v1.2  kreiere Stubs nur für Java 2 Stub Protokoll
```

2.2.5.4. Host Server

Remote RMI Services müssen innerhalb eines Serverprozesses verwaltet werden. Die Klasse TaschenrechnerServer ist ein sehr einfacher Server, der das absolut Notwendige für einen Serverprozess zur Verfügung stellt.

```
import java.rmi.Naming;

public class TaschenrechnerServer {

  public TaschenrechnerServer() {
    try {
      Taschenrechner c = new TaschenrechnerImpl();
      Naming.rebind("rmi://localhost:1099/TaschenrechnerService", c);
    } catch (Exception e) {
      System.out.println("Ausnahme: " + e);
    }
  }

  public static void main(String args[]) {
    new TaschenrechnerServer();
  }
}
```

Kopieren Sie auch dieses Programm in Ihr Arbeitsberzeichnis und übersetzen Sie es mit der CLASSPATH Option:

```
>javac -classpath . TaschenrechnerServer
```

2.2.5.5. Client

Der Quellcode für den Client könnte folgendermassen aussehen:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class TaschenrechnerClient {

    public static void main(String[] args) {
        try {
            // Client und Server auf verschiedenen Hosts
            // Taschenrechner c = (Taschenrrechner)
           //Naming.lookup("rmi://remotehost/TaschenrechnerService");
            Taschenrechner c = (Taschenrrechner)
             Naming.lookup("rmi://localhost/TaschenrechnerService");
            System.out.println( c.sub(4, 3) );
            System.out.println( c.add(4, 5) );
            System.out.println( c.mul(3, 6) );
            System.out.println( c.div(9, 3) );
        }
        catch (MalformedURLException murle) {
            System.out.println();
            System.out.println(
              "MalformedURLException");
            System.out.println(murle);
```

```
        }
        catch (RemoteException re) {
            System.out.println();
            System.out.println(
                        "RemoteException");
            System.out.println(re);
        }
        catch (NotBoundException nbe) {
            System.out.println();
            System.out.println(
                       "NotBoundException");
            System.out.println(nbe);
        }
        catch (
            java.lang.ArithmeticException
                                    ae) {
            System.out.println();
            System.out.println(
             "java.lang.ArithmeticException");
            System.out.println(ae);
        }
    }
}

>javac -classpath . TaschenrechnerServer
```

2.2.5.6. Ausführen des RMI Systems

Nun ist alles bereit zum Start des Systems! Hier zuerst eine Übersicht:

Sie benötigen drei Konsolfenster:

1. eines für die RMIRegistry.
   Sie müssen die Registry im Verzeichnis starten, in dem die Klassendateien sind. Dadurch können Sie das Laden der Klassen mittels HTTP oder FTP umgehen.
2. in einem weiteren Fenster starten Sie den Server, TaschenrechnerServer.
3. im dritten Fenster starten Sie schliesslich den Client, den TaschenrechnerClient.

Starten Sie nun die Registry! Wie erwähnt müssen Sie dies im Verzeichnis tun, in dem sich die Class Dateien befinden. *Löschen Sie vor dem Start den CLASSPATH!*

```
set CLASSPATH=
rmiregistry
```

startet die RMI Registry. Falls der CLASSPATH nicht gelöscht wird, kann beim nächsten Schritt (Starten des Services) ein Fehler resultieren
(Ausnahme: TaschenrechnerServer java.rmi.ServerException: RemoteException occurred in server
thread; nested exception is:
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
java.lang.ClassNotFoundException: TaschenrechnerImpl_Stub)

Falls alles gut aussieht, können Sie nun als nächstes den TaschenrechenrServer starten.

```
>java -cp . TaschenrechnerServer
```

Der Server sollte nun starten und die Taschenrechner Implementation (TaschenrechnerImpl und damit Taschenrechner) in den Speicher laden und auf Client Anfragen warten

Nun folgt schlussendlich nur noch der TaschenrechnerClient:

```
>java -cp . TaschenrechnerClient
```

Falls alles wie geplant läuft, sollten Sie folgende Ausgabe sehen:

```
1
9
18
3
```

Falls Sie beim Starten des Services oder des Clients -verbose als Option verwenden, können Sie verfolgen was alles geladen wird.

Falls Sie nun die RMIRegistry stoppen und wieder starten, müssen Sie das Serverobjekt erneut anmelden, da die Registry den Namensdienst lediglich temporär aufbaut und Bindings nicht persistent speichert.

Das war's! Sie haben eine arbeitende RMI Anwendung. Obschon Sie alle Konsolfenster auf einem Rechner ausgeführt haben, auf dem TCP/IP funktionieren muss, verwendet RMI den Netzwerkstack und TCP/IP, um zwischen (den hier drei) verschiedenen JVMs zu kommunizieren. Sie haben also ein voll funktionsfähiges RMI System.

2.2.6. Übung

Lösen Sie jetzt die folgenden Aufgaben aus dem Aufgabensatz "RMI Praxis"

1. UML Diagramm des RMI Beispiel-Programms (Bankkonto)
2. Einfaches Bankkonto: ergänzen und vervollständigen Sie den Programmcode

2.2.7. Parameter in RMI

Sie haben bisher lediglich gesehen, dass RMI Methodenaufrufe von entfernten Objekten unterstützt. Die Fragen, denen wir uns nun widmen lauten:

1. wie werden Parameter zwischen den unterschiedlichen JVMs hin und her transportiert?
2. Welche Semantik wird dazu verwendet?
3. Verwendet RMI einen "pass-by-value" oder einen "pass-by-reference" Mechanismus?

Die Antwort hängt davon ab, ob die Parameter primitive Datentypen, Objekte oder remote Objekte sind.

2.2.8. Parameter in einer einzelnen JVM

Als erstes untersuchen wir, wie Parameter lokal in einer einzigen JVM ausgetauscht werden. Die normale Semantik der Parameterübergabe für Java ist "pass-by-value". Wann immer ein Parameter an eine Methode übergeben wird, erstellt die JVM eine Kopie des Werts, stellt diese Kopie auf den Stack und führt dann die Methode aus. Falls ein Programmteil innerhalb der Methode einen Parameter benutzen möchte / muss, greift dieser Programmteil auf den Stack zu und verwendet die Kopie des Parameters. Auch die Rückgabewerte eines Methodenaufrufs sind Kopien.

Wann immer ein einfacher Datentyp (boolean, byte, short, int, long, char, float, oder double) als Parameter an eine Methode übergeben wird, ist dieser Mechanismus "pass-by-value" problemlos und leicht zu verstehen. Schwieriger wird die Angelegenheit, wenn ein Objekt als Parameter übergeben wird.

Die Objekte werden im Heap im Hauptspeicher gespeichert; auf diese Objekte wird mittels Referenzvariablen zugegriffen. Obschon im folgenden Programmausschnitt scheinbar ein Objekt als Parameter an die print-Methode übergeben wird, handelt es sich beim Parameter s um eine Referenz auf das String Objekt:

```
String s = "Test";
System.out.println(s);
```

An Stelle der Objekte wird also die Referenzvariable an die Methode übergeben. Im obigen Beispiel wird eine Kopie der Referenzvariable s erstellt (dadurch wird auch der Referenzzähler auf das String Objekt um eins erhöht). Diese Kopie (der Referenzvariable) wird auf den Stack gelegt. Innerhalb der Methode verwendet der Programmcode diese Kopie der Referenzvariable, um auf das Objekt zuzugreifen.

Nun müssen wir uns noch mit der komplexeren Situation befassen, bei der mehrere JVMs zusammenarbeiten, möglicherweise weit verteilt.

2.2.8.1. Primitive, einfache Parameter

Wann immer ein einfacher Datentyp als Parameter an eine remote Methode übergeben wird, geschieht dies indem das RMI System den Parameter "by-value", also lediglich eine Kopie des Werts übergeben wird. RMI macht eine Kopie des einfachen Datentyps und sendet diese Kopie an die entfernte / remote Methode.
Falls eine Methode auch einen einfachen Datentyp zurück liefert, geschieht auch dies durch Aufruf der JVM "by-value".

Die Werte werden in einem maschinenunabhängigen Format übertragen. Damit können die JVMs unterschiedlicher Systeme zuverlässig miteinander kommunizieren.

2.2.8.2. Objekt Parameter

Falls ein Objekt als Parameter einer remote Methode eingesetzt wird, verändert sich die Semantik im Vergleich zu einfachen JVM Systemen. *RMI sendet in diesem Fall das ganze Objekt*, nicht die Referenz, von einer JVM zur (möglicherweise weit) entfernten JVM. Es wird also das *Objekt*, "Object-by-value", nicht die Referenz auf das Objekt transportiert. Analog wird im Falle einer Rückgabe eines Objekts eine Kopie dieses Objekts an den Client übermittelt.

Dieses Senden eines Objekts an eine remote JVM ist eine nichttriviale Aufgabe. Ein Java Objekt kann eine sehr einfache Struktur haben und recht isoliert sein; es kann aber auch sehr komplex und zusammengesetzt sein oder es könnte sich um eine komplexe Objektstruktur, eine Baumstruktur, ... handeln. Da die unterschiedlichen JVMs Heap Speicher nicht teilen, muss RMI das referenzierte Objekt und all seine Objektreferenzen hin und her senden. Dies kann einige CPU Zeit und Netzwerk Bandbreite in Anspruch nehmen.

RMI verwendet dazu die *Objektserialisierung*, um Objekte in eine lineare Darstellung zu transformieren, in dem sie dann über das Netzwerk gesandt werden können. Objektserialisierung linearisiert also ein Objekt samt seinen Objektreferenzen. Serialisierte

Objekte können auch wieder deserialisiert werden. Dies geschieht in der Ziel JVM und damit steht das Objekt in der entfernten JVM zur weiteren Bearbeitung zur Verfügung.

2.2.8.3. Remote Objekt Parameter

RMI stellt eine neue Art Parameter zur Verfügung: remote Objekte. Wie Sie bereits im Beispiel oben gesehen haben, kann ein Client eine Referenz auf ein remote Objekt erhalten. Dies geschieht mittels des RMI Registry Programms. Dabei können remote Objekte entweder als Parameter oder auch als Rückgabewerte auftauchen. Im folgenden Beispiel für einen Bankmanager wird die Methode getKonto() benutzt, um eine remote Referenz auf einen entfernten Dienst Konto zu erhalten.

```
BankManager bm;
Konto     a;
try {
    bm = (BankManager) Naming.lookup(
              "rmi://BankServer/BankManagerService"
         );
    a  = bm.getKonto( "Gates" );
    // Code, der mit der Kontoinformation arbeitet
    // ...
}
catch (RemoteException re) {
}
```

Die Implementation der getKonto() Methode zeigt, dass diese Methode eine lokale Referenz auf den remote Service zurück liefert.

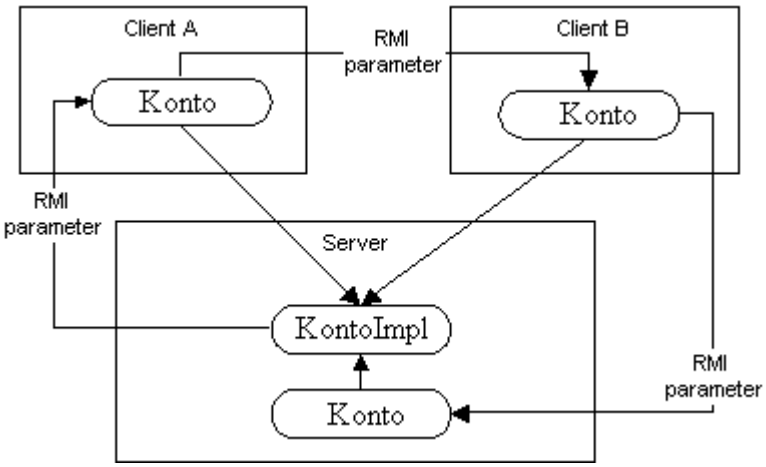
```
public Konto
   getKonto(String kontoName) {
    // Code zum Finden des Kontos
    // ...
    KontoImpl ki = ...

    // liefere die Referenz,die gefunden wurde
    // ...
    return KontoImpl;
}
```

Nun müssen wir noch die Rolle der Stubs, der Proxy Objekte verstehen!
Falls eine Methode eine lokale Referenz auf ein exportiertes Objekt liefert, liefert RMI nicht das Objekt. RMI substituiert ein anderes Objekt (das Proxy Objekt für diesen Service).

Das folgende Diagramm illustriert, wie RMI Aufrufe eingesetzt werden können, um:

- eine remote Referenz vom Server an den Client A zu liefern
- eine remote Referenz vom Client A zum Client B zu senden
- die remote Referenz vom Client B zurück zum Server zu senden



Zum Ablauf:
wenn das KontoImpl Objekt an den Client A zurück geliefert wird, wird stattdessen ein Proxy Objekt Konto geliefert. Anschliessend sendet der Methodenaufruf die Referenz an den Client B und dann schliesslich zurück an den Server.

Im gesamten Ablauf wird auf ein und das selbe Objekt referenziert.

Auch nach der Rückgabe des Objekts an den Server, an dessen lokale Maschine, wird nicht einfach eine lokale Kopie des (lokal vorhandenen) Objekts verwendet. Dies wäre zwar schnell, aber würde die Semantik, die Bedeutung der remote Objekte komplexer gestalten.

2.2.9. Übung

Lösen Sie jetzt die Aufgabe zu den RMI Parametern

2.2.10. RMI Client-seitige Callbacks

Es sind Anwendungsarchitekturen denkbar und üblich, bei denen der Server dem Client eine Mitteilung senden muss. Beispiels dafür wären: Feedback über den Fortschritt einer Berechnung oder Suche, Warnung vor Problemen etc.

Damit dies möglich ist, muss der Client auch RMI Serverfunktionalität besitzen. Für RMI ist dies an für sich kein Problem, da RMI nicht zwischen Server und Client unterscheiden muss. Allerdings kann es wenig Sinn machen, dass der Client `java.rmi.server.UnicastRemoteObjekt` erweitert. In diesen Fällen kann das remote Objekt (beim Client) sich für den remote Einsatz vorbereiten, indem es die statische Methode UnicastRemoteObjekt.exportObjekt (<remote_Objekt>) verwendet.

2.2.11. Übung

Lösen Sie die Aufgabe "RMI Client Callbacks"

2.2.12. Verteilen und Installieren von RMI Software

RMI ergänzt die Java Platform durch ein Distributed Class Modell und erweitert die Java Technologie auf mehrere JVMs gleichzeitig. Es dürfte klar sein, dass das Installieren eines RMI Systems komplexer ist als das Aufsetzen eines Java Laufzeitsystems auf einem einzelnen Rechner. In diesem Abschnitt befassen wir uns mit Fragen im Zusammenhang mit dem Installieren und Verteilen von RMI Applikationen.

Dazu gehen wir davon aus, dass Sie eine verteilte Applikation entwickelt haben und sich nun Gedanken machen müssen, wie Sie diese Applikation auf die einzelnen Knoten verteilen und auf den Knoten, sprich Rechnern, installieren können.

2.2.12.1. Verteilung von RMI Klassen

Um eine RMI Applikation zu betreiben, müssen die Class Dateien am richtigen Platz sein, damit sie vom Server und den Clients gefunden werden können.

Server
der Server benötigt folgende Klassen (der Server Class Loader muss folgende Klassen finden):

1. Remote Service Interface Definitionen
2. Remote ServiceIimplementationen
3. Skeletons für die Implementationsklassen (nur für JDK 1.1 basierte Server)
4. Stubs für die Implementationsklassen
5. alle andern Serverklassen (spezifische Klassen der Anwendung)

Client
der Client benötigt folgende Klassen (der Client Class Loader muss folgende Klassen finden)

1. Remote Service Interface Definitionen
2. Stubs für die remote Service Implementationsklassen
3. Serverklassen für Objekte, die vom Client benötigt werden (beispielsweise Rückgabeobjekte)
4. alle andern Clientklassen

Nachdem bekannt ist, welche Klassen auf welchem Knoten vorhanden sein müssen, ist es ein leichtes dafür zu sorgen, dass die Dateien dem entsprechenden JVM Class Loader zur Verfügung stehen. Es hat sich bewährt, die Klassen jeweils in ein jar Archiv zu packen. Sie finden im Anhang einweiteres vollständiges Beispiel, in dem diese Technik eingesetzt wird.

2.2.12.2. Automatische Distribution von Klassen

Die RMI Designer erweiterten das Konzept des Ladens von Klassen, indem zum Laden auch FTP oder HTTP Server eingesetzt werden können.

Diese Erweiterung führt dazu, dass die Klassendateien an einem oder wenigen Stellen gespeichert werden können und alle Knoten sich die benötigten Klassen von dort mit Hilfe eines FTP oder eines HTTP Servers laden können.

RMI unterstützt dieses remote Klassenladen mittels der `RMIClassLoader`. Falls ein Client oder Server in einem RMI System läuft und sieht, dass eine Klasse von einer remote Lokation laden muss, wird der RMIClassLoader aufgerufen, um diese Arbeit zu erledigen.

Die Art und Weise, wie RMI die Klassen lädt, wird durch Properties kontrolliert. Properties werden beim Starten der JVM gesetzt.

```
java [ -D<PropertyName>=<PropertyWert> ]+ <ClassFile>
```

Beispielsweise wird mit der Option **java.rmi.server.codebase** eine URL gesetzt. Diese URL zeigt auf eine file:, ftp:, oder http: Lokation, an der sichClass Dateien für Objekte befinden,

welche von Applikation auf der JVM (die mit der Property gestartet wird) an andere Applikationen gesandt werden: die zu sendenden Objekte stammen *von* der JVM.

Falls ein Programm, welche in einer JVM läuft, ein Objekt an eine andere JVM sendet (beispielsweise als Rückgabewert eines Methodenaufrufs), muss die Ziel JVM wissen wo sich die Class Datei befindet, damit sie die Class Datei für dieses Objekt laden kann.

Falls RMI ein Objekt via Objektserialisierung schickt, wird die URL, welche als Property Parameter angegeben wurde, in das serialisierte Objekt eingebettet.

Achtunge:	*RMI sendet nicht die Class Datei mit den serialisierten Objekten.* *Die URL (als Adresse) wird in das serialisierte Objekt eingebettet.*

Falls die remote JVM die Class Datei laden muss, findet sie die URL im serialisierten Objekt und kann dann den Server an der URL bitten, die Datei zu liefern bzw. die Datei beim Server abholen.

Falls die Property java.rmi.server.useCodebaseOnly auf true, gesetzt wird, dann lädt die JVM die Klassen entweder von den Lokationen im CLASSPATH oder von der spezifizierten URL.

Durch den Einsatz unterschiedlicher Kombinationen der verfügbaren Systemeigenschaften (Properties) können unterschiedliche RMI Systemkonfigurationen kreiert werden. Hier einige Beispiele:

abgeschlossen:
alle Klassen, die von Client oder Server benutzt werden, müssen auf der JVM gefunden werden und im CLASSPATH referenziert werden. Dynamisches Laden von Klassen wird nicht unterstützt.

Server basiert:
ein Client wird wie ein Applet von der CODEBASE des Servers, zusammen mit allen benötigten Klassen geladen, analog wie bei einem Applet.

Client dynamisch:
die primären Klassen werden mit Hilfe der im CLASSPATH gefundenen Angaben geladen. Unterstützende, zusätzliche Klassen werden vom `java.rmi.server.RMIClassLoader` vom HTTP oder FTP Server im Netzwerk, gemäss Property Angaben geladen.

Server dynamisch:
Die primären Klassen werden auch hier gemäss Angaben im CLASSPATH geladen. Unterstützenden Klassen werden mit Hilfe des `java.rmi.server.RMIClassLoader` von einem HTTP oder FTP Server im Netzwerk geladen. Die Lokationsangabe zu den zusätzlichen Klassen stammt vom Client.

Bootstrap Client:
in dieser Konfiguration wird der *gesamte* Client Code von einem HTTP oder FTP Server, der sich irgendwo im Netzwerk befindet, geladen. Bei der Client JVM befindet sich lediglich ein minimaler Bootstrap Loader.

Bootstrap Server:
in dieser Konfiguration wird der *gesamte* Server Code von einem HTTP oder FTP Server, der sich irgendwo im Netzwerk befindet, geladen. Bei der Server JVM befindet sich lediglich ein minimaler Bootstrap Loader.

In der Übung zu dieser Sektion kreieren Sie eine Bootstrap Client Konfiguration. Beachten Sie alle Hinweise genaustens, da alle Dateien am richtigen Ort sein müssen. Sonst funktioniert das Beispiel sicher nicht!

2.2.13. Übung

Bootstrap Beispiel

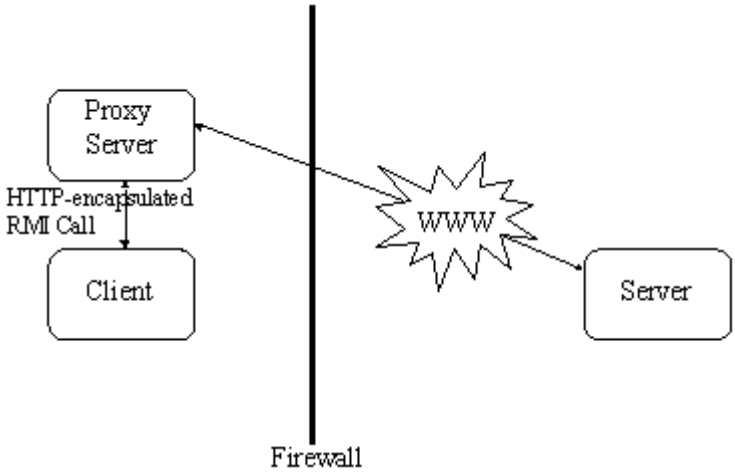
2.2.13.1. Firewall Themen

Netzwerk basierte Applikationen werden sich automatisch mit Firewalls beschäftigen müssen. Typischerweise blockiert eine Firewall den gesamten Netzwerkverkehr, ausser jenem für bestimmte, "geprüfte Ports"

Da der RMI Transport Layer dynamische Socket Verbindungen zwischen Client und Server aufbaut, um die Kommunikation zu ermöglichen, wird der JRMP Datenverkehr typischerweise durch die meisten Firewalls blockiert.

Zum Glück haben die RMI Designer an dieses Problem gedacht und eine Lösung in den RMI Transport Layer eingebaut. Damit RMI durch Firewalls hindurch kommunizieren kann, verwendet RMI HTTP Tunneling, indem es die RMI Aufrufe in HTTP POST Anfragen einbettet.

Schauen wir uns dieses HTTP Tunneling genauer an.Betrachten wir als erstes folgendes Szenario, bei dem sich der Client, der Server oder beide hinter einer Firewall befinden. Das Diagramm zeigt den Fall, dass der RMI Client, der sich hinter der Firewall befindet, mit einem externen Server kommunizieren möchte.

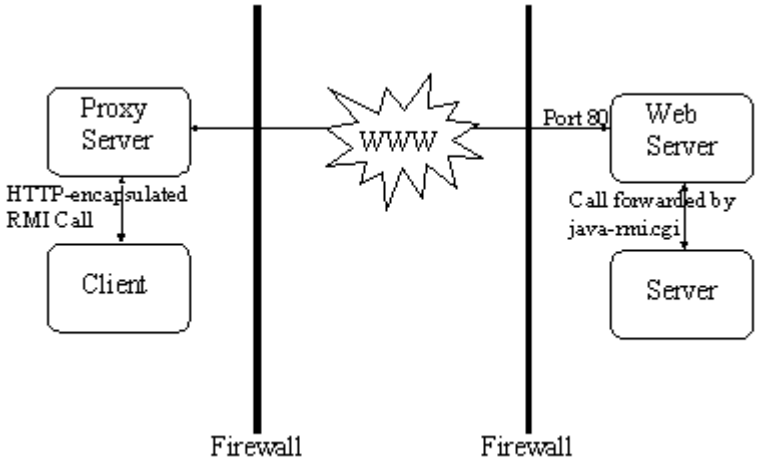


Im obigen Szenario wird die Verbindung mit Hilfe des Transportlayers zwischen Client und Server durch die Firewall blockiert. In diesem Fall versucht der RMI Transport Layer automatisch die JRMP Daten in einen HTTP POST Request einzubetten. Der HTTP Post Header besitzt folgenden Header:

```
http://hostname:port
```

Falls ein Client sich hinter der Firewall befindet ist es wichtig, die Systemeigenschaft http.proxyHost passend zu setzen. Da Firewalls in der Regel das HTTP Protokoll durchlassen, sollte es dem skizzierten Proxy Server möglich sein, den Methodenaufruf zum Port des externen Servers zu übermitteln. Sobald die HTTP verschlüsselte JRMP Daten eintreffen, werden sie automatisch dekodiert und an den RMI Transportlayer weitergeleitet. Die Antwort des Servers wird dann wieder verpackt und als HTTP Daten an den Client / Proxy zurück geschickt.

Das folgende Diagramm zeigt das Szenario, bei dem beide, der RMI Client und der RMI Server, sich hinter einer Firewall befinden und die Daten lediglich mittels HTTP (Port 80) Einbettung transportiert werden können.



In diesem Fall benötigt der RMI Transportlayer eine zusätzliche Umleitung. Denn nun kann der Client die HTTP verschlüsselten JRMP Daten nicht an direkt an den Prot senden, da der Server zum Port sich ebenfalls hinter der Firewall befindet. In diesem Fall plaziert der RMI Transportlayer die JRMP Aufrufe in HTTP Pakete und sendet diese an den Port 80 des Servers. Der HTTP POST Header sieht in diesem Fall etwas anders aus:

```
http://hostname:80/cgi-bin/java-rmi?forward=<port>
```

Alternativ und in neueren Versionen wird ein Servlet eingesetzt, sofern Servlets unterstützt werden.

```
#!/bin/sh
#
# java-rmi.cgi
#
# This file handles rmi requests to download RMI code
# The work is done by the class:
#      sun.rmi.transport.proxy.CGIHandler
# This class supports a QUERY_STRING of the form
# "forward=<port>" with a REQUEST_METHOD of "POST"
# The body of the request will be forwarded to the server (
# as aPOST request) to the port given in the URL. The response
# will be returned to the orginal requester.

# Set the path to include the location of the jdk to run
PATH=/opt/java/bin:$PATH
```

```
java   \
   -DAUTH_TYPE=$AUTH_TYPE \
   -DCONTENT_LENGTH=$CONTENT_LENGTH \
   -DCONTENT_TYPE=$CONTENT_TYPE \
   -DDOCUMENT_ROOT=$DOCUMENT_ROOT \
   -DGATEWAY_INTERFACE=$GATEWAY_INTERFACE \
   -DHTTP_ACCEPT="$HTTP_ACCEPT" \
   -DHTTP_CONNECTION=$HTTP_CONNECTION \
   -DHTTP_HOST=$HTTP_HOST \
   -DHTTP_USER_AGENT="$HTTP_USER_AGENT" \
   -DPATH_INFO=$PATH_INFO \
   -DPATH_TRANSLATED=$PATH_TRANSLATED \
   -DQUERY_STRING=$QUERY_STRING \
   -DREMOTE_ADDR=$REMOTE_ADDR \
   -DREMOTE_HOST=$REMOTE_HOST \
   -DREMOTE_IDENT=$REMOTE_IDENT \
   -DREMOTE_USER=$REMOTE_USER \
   -DREQUEST_METHOD=$REQUEST_METHOD \
   -DSCRIPT_NAME=$SCRIPT_NAME \
   -DSERVER_NAME=$SERVER_NAME \
   -DSERVER_PORT=$SERVER_PORT \
   -DSERVER_PROTOCOL=$SERVER_PROTOCOL \
   -DSERVER_SOFTWARE=$SERVER_SOFTWARE \
   sun.rmi.transport.proxy.CGIHandler
```

Dies führt dazu, dass das CGI Skript java-rmi.cgi auf der Serverseite ausgeführt wird. Sie erkennen dies im obigen Skript: am Anfang wird die Java VM gestartet. der Rest sind Properties. Die lokale JVM entpackt das HTTP Paket und leitet den Methodenaufruf an den Server weiter, an den spezifizierten Port. RMI JRMP-basierte Antworten werden vom Server als HTTP REPLY Pakete an den ursprünglichen Client Port gesandt, wo sie entpackt werden und die RMI Informationen an den RMI Stub weitergeleitet werden.

.

Natürlich muss das obige Skript java-rmi.cgi angepasst werden, beispielsweise der Pfad,. Zudem muss das Skript in das cgi-bin Verzeichnis des Servers kopiert werden. Zudem muss beim Starten des RMI Servers der voll qualifizierte Host / Domainname als Systemeigenschaft angegeben werden, um allfällige DNS Auflösungsprobleme zu verhindern:

```
java.rmi.server.hostname=host.domain.com
```

Hinweis:
An Stelle des CGI Skripts kann man ein Servlet verwenden. Dadurch wird das Tunneln effizienter implementiert. Sie finden den Quellcode des Servlets bei Sun: RMI FAQ.
Wie im obigen Satz bereits implizit erwähnt, führt das Tunneln zu signifikanten Leistungseinbussen. Zudem kann die RMI Applikation kein Multiplexing mehr realisieren. Die Kommunikation gehorcht also indiesem Falle einem klassischen diskreten request / response (reply) Protokoll. Zudem ergibt sich beim Einsatz des CGI Skripts ein Scherheitsloch, welches beim Einsatz des Servlets geschlossen werden kann.Zudem darf die Applikation keine Callbacks verwenden, weil HTTP ein zustandsloses Protokoll ist. Falls Sie all dies überzeugt hat, dass HTTP Tunneln keine gute Lösung ist, können Sie dies auch mit der Systemeigenschaft explizit verbieten :

```
java.rmi.server.disableHttp=true
```

2.2.14. Distributed Garbage Collection

Einer der Vorteile der Java Programmierung ist der, dass man sich keinerlei Gedanken über die Speicherverwaltung machen muss. Die JVM besitzt einen automatischen Garbage Collector, der dafür sorgt, dass alle Objekte, die nicht mehr benötigt werden (referenziert werden) automatisch aus dem System entfernt werden, der Speicherplatz also wieder frei wird.

Ein Desingziel für RMI war eine problemlose Integration von RMI in die Java Sprachumgebung, also auch Garbage Collection. Der Entwurf eines effizienten Garbage Collectors ist schon eine schwierige Aufgabe; der Entwurf eines verteilten Garbage Collectors ist noch wesentlich schwieriger.

Das RMI System verwendet einen Referenzzählalgorithmus als Basis für den verteilten Garbage Collector. Der Algorithmus basiert auf jenem von Modula-3's Network Objects. In diesen Systemen führt der Server eine Liste, in die Client Zugriffe auf die Objekte festgehalten werden. Das Objekt wird als "dirty" bezeichnet, falls Referenzen auf das Objekt weisen; sonst wird das Objekt als "clean" bezeichnet. Ein "dirty" Objekt wird "clean" sobald die Referenzen auf ein Objekt aufgehoben werden.

Das Interface zum DGC (distributed garbage collector) ist im Stubs und Skeletons Layer angesiedelt. Ein remote Objekt kann den DGC steuern, indem es das Interface `java.rmi.server.UnReferenced` implementiert. Die `unReferenced` Methode liefert die nötige Notifikation, falls keine Referenzen auf das Objekt verweisen.

Da das System verteilt ist, muss man zusätzlich eine Lease Zeit einbauen, die angibt, wie lange allenfalls auf weitere Anfragen gewartet werden soll. Falls innerhalb dieser Lease Zeit keine weitere Anfrage kommt, kann das Objekt gelöscht werden bzw. der Referenzzähler reduziert werden. Die Lease Zeit wird mittels der Systemeigenschaft java.rmi.dgc.leaseValue angegeben. Der Wert wird in Millisekunden angegeben, Standardwert sind 10 Minuten.

Wegen dem Garbage Collector muss ein Client damit rechnen, dass ein remote Objekt "verschwindet". In der folgende Übung haben Sie Gelegenheit mit dem DGC zu experimentieren.

2.2.15. Übung

Distributed Garbage Collection

2.2.16. Serialisierung von Remote Objekten

Wenn man ein System entwirft, welches RMI einsetzt, möchte man Flexibilität bezüglich der Lokation der Objekte haben. Man möchte also möglichst flexibel oder spät festlegen, wo die Objekte ausgeführt werden. In der Regel muss dieser Entscheid bereits sehr früh gefällt werden. Man kann remote Objekte nicht so einfach von einer JVM auf die andere verschieben und dann dort ausführen.

In RMI Applikationen werden remote Objekte als Implementationen des `java.rmi.Remote` Interfaces deklariert, und schon verhindert RMI, dass das Objekt serialisiert und von einer JVM zur andern geschickt wird. Statt die Implementationsklassen eines `java.rmi.Remote` Interfaces hin und her zu senden substituiert RMI die Stub Klasse. Diese Substitution geschieht RMI intern, kann also von aussen nicht gesteuert werden.

Das Problem kann man auf zwei Arten lösen:

1. die erste Variante besteht in der **manuellen Serialisierung** und dem manuellen Senden und Deserialisierung. Dies kann man auf zwei Arten bewerkstelligen:
   i) indem man eine ObjektInputStream und ObjektOutputStream Verbindung zwischen den zwei JVMs herstellt. Über diese Verbindung kann das Objekt verschickt werden
   ii) zuerst wird es in ein Byte Array serialisiert werden und als Byte Array als Rückgabe auf einen Methodenaufruf zwischen den JVMs hin und her gesandt werden.

   Beide Techniken bedingen eigentlich Programmcode unterhalb des RMI Levels. Das hat zur Folge, dass die Wartung komplex werden kann.

2. eine zweite Strategie besteht im Einsatz des **Delegation Pattern**.
   In diesem Pattern wird die Funktionalität in einer Klasse implementiert, die:
   - das Interface `java.rmi.Remote` *nicht* implementiert
   - das Interface `java.io.Serializable` *implementiert*

Nachdem man so ein eigenes remote Interface gebaut hat, muss man nun den remote Zugriff deklarieren.

Schauen wir uns einmal an, wie eine solche Implementation in Form des Patterns aussehen könnte. Dabei können wir aber nicht alle Details der Implementation zeigen (siehe Übungen). Aber die grundsätzliche Struktur einer solchen Lösung sollte Ihnen nach dem Durcharbeiten der folgenden Seiten klar sein.

Der folgende Programmcode entspricht keinem funktionsfähigen Programm sondern dient lediglich der Illustration der Konzepte!

```
// definieren wir die Funktionalität in einem lokalen Objekt
public class LokalesModell implements java.io.Serializable {
  public String getVersionNumber() {
    return "Version 1.0";
  }
}
```

Nun deklarieren wir ein `java.rmi.Remote` Interface, in dem die selbe Funktionalität definiert wird:

```
interface RemoteModellRef extends java.rmi.Remote {
  String getVersionNumber() throws java.rmi.RemoteException;
}
```

Die Implementation des remote Services akzeptiert eine Referenz auf das lokale Modell LokalesModell und delegieren die Arbeit an das Objekt:

```
public class RemoteModellImpl extends java.rmi.server.UnicastRemoteObject
    implements RemoteModellRef {
  LokalesModell lm;

  public RemoteModellImpl(LokalesModell lm)throws java.rmi.RemoteException{
    super();
    this.lm = lm;
  }
  // Delegation an die lokale Modell Implementation
  public String getVersionsNumber() throws java.rmi.RemoteException {
```

```
    return lm.getVersionsNumber();
  }
}
```

Schiesslich wird ein remote Service definiert, der Zugriffe auf Clients gestattet. Dies geschieht mit Hilfe des `java.rmi.Remote` Interface und einer Implementation:

```
interface RemoteModellMgr extends java.rmi.Remote
{
  RemoteModellRef getRemoteModellRef()
    throws java.rmi.RemoteException;

  LokalesModell     getLokalesModell()
    throws java.rmi.RemoteException;
}

public class RemoteModellMgrImpl
    extends
     java.rmi.server.UnicastRemoteObjekt
    implements RemoteModellMgr
  {
  LokalesModell lm;
  RemoteModellImpl rmImpl;

  public RemoteModellMgrImpl()
      throws java.rmi.RemoteException
  {
    super();
  }

  public RemoteModellRef getRemoteModellRef()
      throws java.rmi.RemoteException
    {
    // Instanzierung
    if (null == lm)
    {
      lm = new LokalesModell();
    }

    // Instanzierung des remote Interface Wrappers
    if (null == rmImpl)
    {
      rmImpl = new RemoteModellImpl (lm);
    }

    return ((RemoteModellRef) rmImpl);
  }

  public LokalesModell getLokalesModell()
      throws java.rmi.RemoteException
  {
    // gibt eine Referenz auf das LokaleModell zurück
    if (null == lm)
    {
      lm = new LokalesModell();
    }
    return lm;
  }
}
```

2.2.17. Übungen

1. Serialisierung von remote Objekten : Server
2. Serialisierung von remote Objekten : Client

2.2.18. Architektur Mobiler Agenten

Das hin und her senden ganzer ausführbarer Objekte und Objektsysteme wurde bereits in den 60er Jahren in der Künstlichen Intelligenz untersucht (Verteilte Intelligente Systeme). Solche Systeme wurden unter dem Schlagwort "Agentensysteme" bekannt; sie haben sich aber bis heute nicht durchgesetzt. Und safür gibt es auch klare Gründe. Die Lösung des Agentenproblems mittels RMI ist auch eher ein "von Hand stricken". Andere verteilte Java Architekturen und Agentensysteme wurden bereits vor einiger Zeit entwickelt und veröffentlicht. Sie werden unter dem Begriff *mobile agent architectures* zusammengefasst.

Einige Beispiele dafür sind:
1. IBM's Aglets Architektur und das
2. ObjektSpace's Voyager System.

Diese Systeme wurden speziell dafür entwickelt, Java Objekte zwischen JVMs zu verschieben und auf den Gast JVMs auszuführen.

2.2.19. Alternative Implementationen

Neben der Sun Implementation von RMI gibt es auch noch andere:

- NinjaRMI
  eine Implementation der Universität von Californien, Berkeley, jedoch für JDK 1.1, aber mit Erweiterungen.
- BEA Weblogic Server
  der BEA Weblogic Server ist ein Applikations Server, der RMI, Microsoft COM, CORBA und EJB (Enterprise JavaBeans) und andere Services unterstützt.
- Voyager
  ObjektSpace's Voyager unterstützt RMI neben DOM, CORBA, EJB, Microsoft's DCOM, und Transaktions Services.

2.2.20. Zusätzliche Quellen

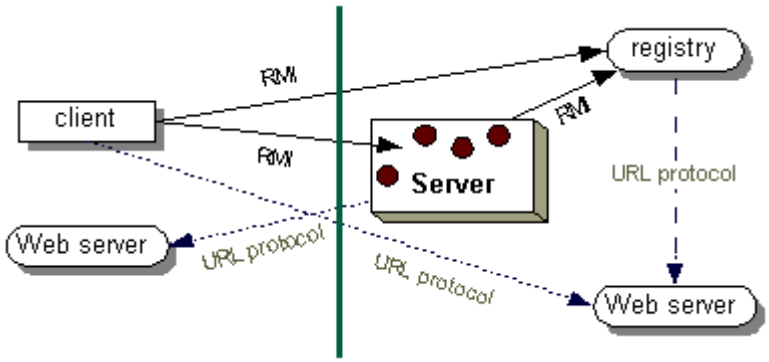
2.2.20.1. Bücher und Artikel

- *Design Patterns*, von Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (The Gang of Four)
  Vor kurzem ist eine "Java Version" erschienen, die neu im 2. Studienjahr eingesetzt wird:
  Java Design Patterns - A Tutorial, von Kames W. Cooper Addison-Wesley, 2000
  eine Vorversion steht als PDF zur Verfügung. Die obige Version wird ind Deutsche übersetzt.
- Sun's RMI FAQ
- RMI over IIOP
- RMI-USERS Mailing List Archive
- *Implementing Callbacks with Java RMI*, by Govind Seshadri, Dr. Dobb's Journal, March 1998

2.3. Ein weiteres vollständig gelöstes Beispiel: der Berechnungsserver

Die im folgenden zu entwickelnede Applikation nennen wir Berechnungsserver. Der Berechnungsserver erhält Anfragen von Clients und ist selbst ein remote Objekt in einem Server. Dieses Objekt erhält Aufgaben für Berechnungen, führt diese aus und liefert die Ergebnisse an den Client zurück. Die Applikation ist praxisrelevant, weil beispielsweise der Server besonders leistungsfähig sein könnte, eventuell mit Spezialhardware.

Neu an der Berechnungsmaschine ist, dass die Aufgaben, die sie zu lösen hat, zum Zeitpunkt, zu dem die Berechnungsmaschine geschrieben wird, nicht bekannt sein müssen. Alles was benötigt wird, ist ein universelles Interface und Klassen (Berechnungsklassen) , die dieses implementieren und unterschiedliche Aufgaben lösen können. Die Berechnungsklassen können zu einem späteren Zeitpunkt definiert und realisiert werden. Sie können auch erst programmiert werden nachdem der Berechnungsserver schon lange gestartet wurde. Der Programmcode für die spezielle Aufgabe kann von RMI zu einem späteren Zeitpunkt heruntergeladen werden.



Diese Art Applikationen werden durch die dynamische Art und Weise, wie Java Klassen lädt, im Speziellen wie RMI dies tut, ermöglicht. Dieses Beispiel zeigt, wie Agenten-ähnliche Systeme, also Systeme mit dynamischen Klassen und variablen Aufgaben mit RMI in Java realisiert werden können.

2.3.1. Vorgehensweise

Wie bereits erwähnt, geht man bei der Realisierung von RMI Applikationen in folgenden Schritten vor:

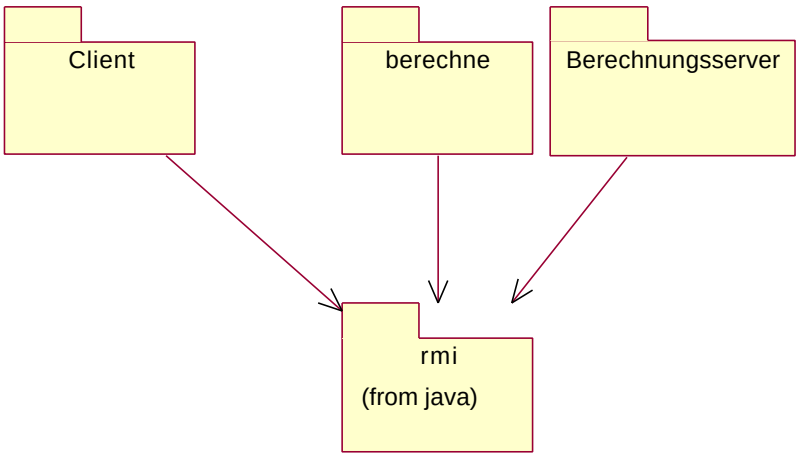
1. Design und Implementation der Komponenten ihrer verteilten Applikation.
2. Übersetzen der Quellcodes und generieren der Stubs.
3. Klassen auf dem Netzwerk verfügbar machen.
4. Starten der Applikation.

2.3.1.1. Design und Implementation der Komponenten ihrer verteilten Applikation.

Als erstes muss die Architektur der Applikation festgelegt werden:
welche Objekte sind lokal?
welche stehen remote zur Verfügung?

Dieser Schritt umfasst:
1. Definition der remote Interfaces
2. Implementation der remote Objekte
3. Implementation der Clients

Unsere Architektur für das Gesamtsystem dürfte demnach etwa folgendermassen aussehen, auf Stufe Packages. Wesentlich sind die drei oberen Packages; über RMI werden alle miteinander verbunden.



2.3.2. Bau des generischen Berechnungsservers

Der Berechnungsserver akzeptiert Aufgaben von Clients, führt die Berechnungen aus und liefert das Ergebnis an den Client zurück. Der Server besteht aus einem Interface und einer Klasse. Das Interface beschreibt die Methoden, die vom Client aufgerufen werden können. Die Klasse liefert die Implementation dazu.

Vorgehen:

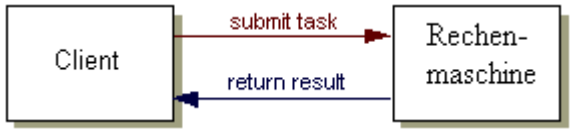
1. Design eines Remote Interface
   in diesem Abschnitt beschreiben wir das Interface, welches die Verbindung zwischen dem Client und dem Server darstellt.
2. Implementing eines Remote Interface
   in diesem Abschnitt definieren wir die Implementation des Interfaces, des remote Objekts. Die Klasse beschreibt den Server, ein Hauptprogramm, welches eine Instanz des remote Objekts kreiert, das Objekt registriert und allenfalls einen Security Manager aufsetzt.

2.3.3. Design eines Remote Interfaces

Herzstück der Berechnungsmaschine ist ein Protokoll, welches

- dem Client gestattet eine Aufgabe an den Server zu delegieren
- dem Berechnungsserver erlaubt die Aufgabe zu lösen und
- die Ergebnisse an den Client zurück liefern kann

Dieses Protokoll muss vom Server und den Clients unterstützt werden.



Jedes dieser Interfaces enthält genau eine Methode:

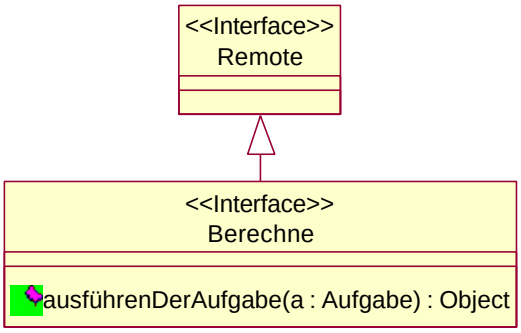
1. Das Berechnungsserver Interface Berechne gestattet Jobs an den Berechnungsserver (Compute Engine in der obigen Grafik) zu delegieren. Es beschreibt, wie die entfernten Teile des Berechnungsservers eingesetzt werden können.
2. das Client Interface Aufgabe definiert wie der Berechnungsserver die aufgegebene Aufgabe zu lösen hat.

Hier das Berechne *Interface:*

```
package berechne;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Berechne extends Remote {
    Object ausführenDerAufgabe(Aufgabe a)
        throws RemoteException;
}
```



Durch die Erweiterung des Interfaces java.rmi.Remote wird das Interface zu einem, dessen Methoden remote genutzt werden können. Jedes Interface, welches dieses Interface implementiert, wird ein remote Objekt.

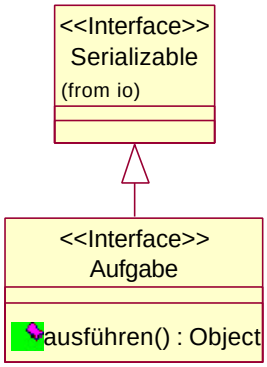
Als Teil eines remore Interfaces ist die Methode ausführenDerAufgabe ebenfalls eine remote Methode. Daher muss sie so definiert werden, dass sie eine java.rmi.RemoteException werfen kann. Diese Ausnahme wird vom RMI System immer dann geworfen, falls ein Kommunikationsausfall oder ein Protokollfehler auftritt.

Eine RemoteException ist eine (vom Compiler) geprüfte Ausnahme, muss also von jedem Programm, welches eine remote Methode implementiert, implementiert werden.

Als zweites definieren wir ein Interface für die Aufgabe. Diese Klasse wird als Argument für die Methode ausführenDerAufgabe() im Berechne Interface eingesetzt. Das Interface berechne.Aufgabe definiert eine Schnittstelle zwischen dem Berechnungsserver und der Aufgabe, die er zu lösen hat.

```
package berechne;
import java.io.Serializable;

public interface Aufgabe extends Serializable {
        Object ausführen();
}
```



Das Aufgabe Interface definiert eine einzige Methode, ausführen, welche ein Objekt zurück liefert, keine Parameter besitzt und auch keine Ausnahme wirft. Da dieses Interface nicht Remote erweitert, braucht die Methode auch nicht die java.rmi.RemoteException zu implementieren.

Der Rückgabewert der Methoden Berechne.ausführenDerAufgabe und Aufgabe.ausführen wurde als vom Typ Objekt deklariert. Das bedeutet, dass jede Aufgabe, die einen einfachen Datentyp zurück gibt, die Werte jeweils umwandeln muss ("wrappen"). Beispielsweise falls ein Typ Integer zurückgegeben wird, muss mit Hilfe der Integer Klasse eine int Zahl hergestellt werden, falls dies gwünscht wird.

Zu beachten ist, dass das Aufgabe Interface java.io.Serializable erweitert. RMI verwendet den Objekt Serialisierungsmechanismus, um Objekte "by-value" zwischen Java VMs zu transportieren.

Die Berechnungsmaschine kann unterschiedliche Aufgaben ausführen, solange sie das Aufgabe Interface implementieren. Natürlich kann die Implementationsklasse auch weitere Datenfelder und Methoden besitzen, die in der abstrakten Definition des Interfaces fehlen und die für die Lösung der Berechnungsazfgabe nötig sind.

Wie geschieht nun der gesamte Ablauf aus RMI Sicht?
Da RMI voraussetzen kann, dass die Aufgabe Objekte in Java geschrieben sind, kann RMI Objekte, die vorher dem Berechnungsserver unbekannt waren, einfach mittels RMI in den Berechnungsserver herunterladen. Damit hat der Client die Möglichkeit neue Aufgaben zu irgend einem Zeitpunkt zu definieren und so den Berechnungsserver dynamisch zu verändern.

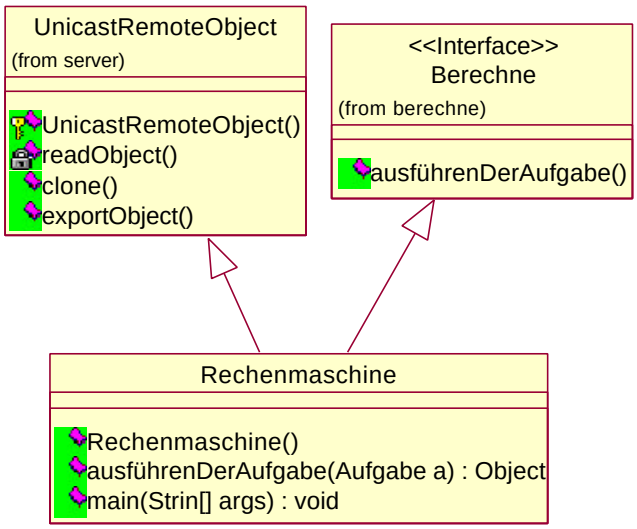
Da wir als Rückgabewert ein Java Objekt zulassen, also den Rückgabewert sehr offen lassen, können die Berechnungen ein allgemeines Java Objekt, nicht nur einen einfachen Datenwert zurück geben.

2.3.4. Implementation eines Remote Interfaces

Nun müssen wir den Berechnungsserver implementieren. Im Allgemeinen sollte die Implementationsklasse eines remote Interfaces mindestens:

- das remote Interface deklarieren, welches implementiert werden soll
- den Konstruktor für das remote Objekt deklarieren
- eine Implementation für jede remote Methode im remote Interface zur Verfügung stellen

Der Server muss das remote Objekt kreieren und installieren. Diese Aufgaben können im main Programm des remote Objekts oder in einer separaten Klasse erledigt werden. Im Einzelnen müssen folgende Aktivitäten erledigt werden:



- kreieren und installieren eines Security Managers
- kreieren einer oder mehrerer remote Objekte
- registrieren von mindestens einem remote Objekt in der RMI remote Objekt Registry oder einem andern Namensdienst , wie beispielsweise JNDI (Java Naming and Directory Interface), damit das System gestartet (gebootstrapped) werden kann.

Unser Berechnungsserver sieht somit folgendermassen aus:

```
package Berechnungsserver;

import java.rmi.*;
import java.rmi.server.*;
import berechne.*;
```

```
public class Rechenmaschine extends UnicastRemoteObject
                    implements Berechne
{
   public Rechenmaschine () throws RemoteException {
      super();
   }

   public Object ausführenDerAufgabe(Aufgabe a) {
      return a.ausführen();
   }

   public static void main(String[] args) {
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(new RMISecurityManager());
      }
      String name = "rmi://host/Berechne";
               // host muss richtig gesetzt werden!!
      try {
         Berechne maschine = new Rechenmaschine ();
         Naming.rebind(name, maschine);
         System.out.println("Rechenmaschine wurde registriert / gebunden");
      } catch (Exception e) {
         System.err.println("Rechenmaschine Exception: " +
                        e.getMessage());
         e.printStackTrace();
      }
   }
}
```

Schauen wir uns nun die einzelnen Komponenten der Implementation der Rechenmaschine an.

2.3.4.1. Deklaration der zu implementierenden remote Interfaces

Die Implementationsklasse für die Rechenmaschine wird deklariert als

```
public class Rechenmaschine extends UnicastRemoteObject
                    implements Berechne
```

Diese Deklaration besagt, dass die Klasse das remote Interface Berechne implementiert (und damit ein remote Objekt definiert) und die Klasse java.rmi.server.UnicastRemoteObject erweitert.

UnicastRemoteObject ist eine RMI Klasse, die als Superklasse für die Implementation von remote Objekten verwendet werden kann..Die Klasse liefert Implementationen für eine ganze Reihe von java.lang.Object Methoden (equals, hashCode, toString), so dass diese für remote Objekte korrekt definiert sind. Die Klasse UnicastRemoteObject umfasst auch Konstruktoren und statische Methoden, mit deren remote Objekte exportiert werden, also das remote Objekt für eintreffende Clientanfragen vorbereiten.

Sie können, wie schon früher erwähnt, darauf verzichten, das UnicastRemoteObject zu erweitern. Allerdings müssen Sie dann im Programm die nötigen Implementationen der java.lang.Object Methoden bereitstellen. Zudem muss dann die remote Objekt Implementation die Methode exportObject() der Klasse UnicastRemoteObject explizit aufrufen. Hier eine kurze Übersicht über die Klasse UnicastRemoteObject :

Konstruktor von UnicastRemoteObject()

protected UnicastRemoteObject()
: kreieren und exportieren eines neuen UnicastRemoteObject für einen anonymen PortCreate.

protected UnicastRemoteObject(int port)
: kreieren und exportieren eines neuen UnicastRemoteObject, welches einen bestimmten Port verwendet..

protected UnicastRemoteObject(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)
: kreieren und exportieren eines neuen UnicastRemoteObject, welches einen bestimmten Port und eine bestimmte Socket-Factory (siehe Beispiel unten) verwendet.

Methoden Zusammenfassung

Object clone()
: Liefert ein Clone Objekt

static RemoteStub exportObject(Remote obj)
: exportiert das remote Objekt obj und macht es damit für eintreffende Aufrufe von Clients verfügbar. Das Objekt verwendet einen anonymen Port.

static Remote exportObject(Remote obj, int port)
: exportiert das remote Objekt obj und macht es damit für eintreffende Aufrufe von Clients verfügbar. Das Objekt verwendet den Port int

static Remote exportObject(Remote obj, int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf) exportiert das remote Objekt obj und macht es damit für eintreffende Aufrufe von Clients verfügbar. Das Objekt verwendet den Port int, der Transport wird über die Socket Factory RMIClientSocketFactory RMI ServerSOcketFactory abgewickelt.static boolean unexportObject(Remote obj, boolean force)
: entfernt das remote Objekt obj aus dem RMI Laufzeitsystem

Methoden, welche von der Klasse java.rmi.server.RemoteServer *geerbt wurden:*
getClientHost, getLog, setLog

Methoden, welche von der Klasse java.rmi.server.RemoteObject *geerbt wurden:*
equals, getRef, hashCode, toString, toStub

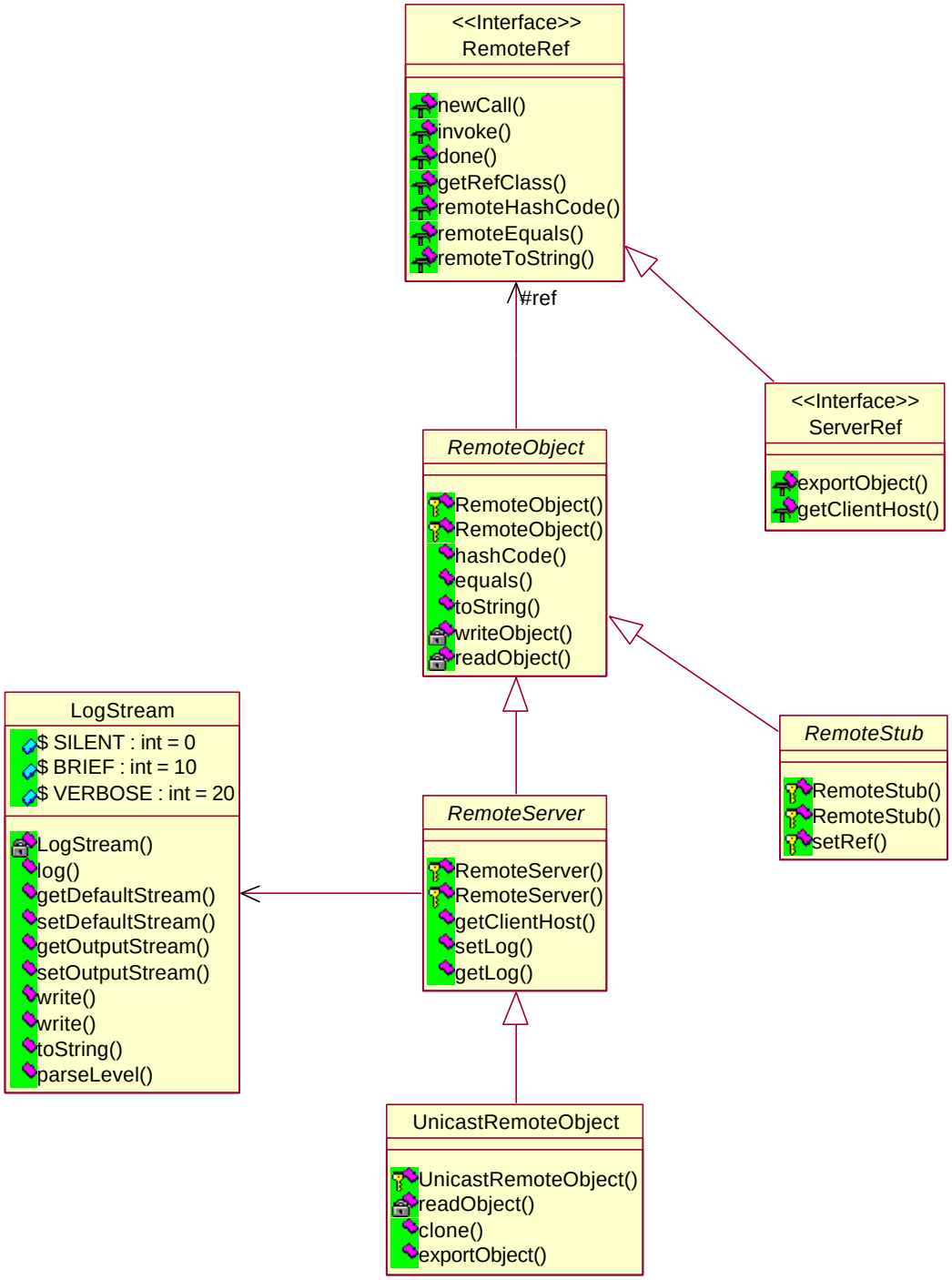
Methoden, welche von der Klasse java.lang.Object *geerbt wurden:*
finalize, getClass, notify, notifyAll, wait, wait, wait

Das Klassendiagramm, aus Rose, zeigt die Vererbungen (einen Ausschnitt) zwischen den einzelnen Klassen aus dem Package java.rmi.server.

REMOTE METHODE INVOCATION - PRAXIS

In unseren Beispiel erweitert die Rechenmaschine das UnicastRemoteObject, und ermöglicht damit die unicast (Punkt-zu-Punkt) remote Kommunikation mittels des RMI Socket-basierten Transports.

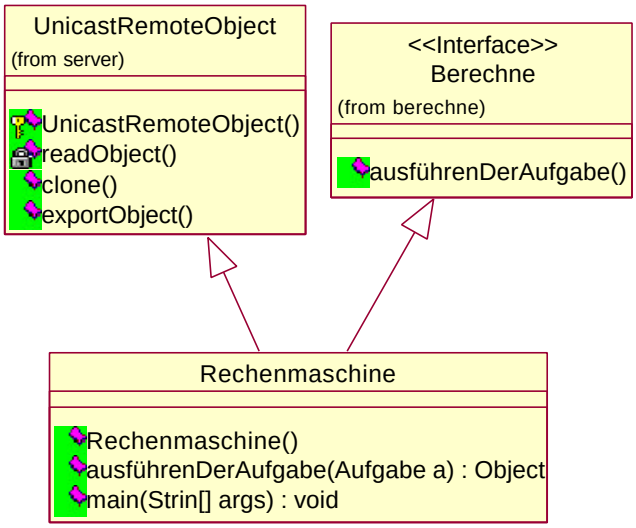
Alternativ dazu könnten wir auch die java.rmi.activation.Activatable Klasse erweitern. Auch diese Klasse besitzt eine exportObject() Methode.



Unsere Rechenmaschine definiert ein einziges remote Interface. Zudem werden einige Methoden definiert, die nur lokal genutzt werden können, beispielsweise der Konstruktor und die main Methode.

2.3.4.2. Definition des Konstruktors

Die Rechenmaschine Klasse besitzt nur einen Konstruktor ohne Argumente.




```
public Rechenmaschine() throws RemoteException {
   super();
}
```

Dieser Konstruktor ruft einfach den Superklasse Konstruktor auf, auch ohne Argumente. Dieser ist gemäss dem Klassendiagramm identisch mit dem UnicastRemoteObject Konstruktor ohne Argumente. Dieser Konstruktor würde ach aufgerufen, falls wir den Konstruktor für unsere Klasse nicht explizit definiert hätten. Die explizite Definition macht das Programm aber klarer.

Im Laufe der Konstruktion wird ein UnicastRemoteObject exportiert, dh. es wird für eintreffende Anfragen verfügbar, indem es an einem anonymen Port auf Anfragen wartet.

Der Konstruktor für das UnicastObject, ohne Argumente, deklariert die Ausnahme RemoteException, die somit auch vom Konstruktor der Rechenmaschine deklariert werden muss. Eine RemoteException kann immer dann während der Konstruktion eines remote Objekts auftreten, falls Kommunikationsresourcen nicht zur Verfügung stehen oder der passende Stub nicht gefunden wird.

2.3.4.3. Implementationen für jede Remote Methode

Die Klasse des remote Objekts muss alle remote Methoden, die im remote Interface spezifiziert wurden, implementieren. Das Interface Berechne enthält nur eine Methode, ausführenDerAufgabe() :

```
public Object ausführenDerAufgabe(Aufgabe a) {
   return a.ausführen();
}
```

Diese Methode implementiert das Protokoll zwischen der Rechenmaschine und ihren Clients. Clients liefern der Rechenmaschine Aufgabe Objekte ab. Diese besitzen eine Methode ausführen(). Die Rechenmaschine führt diese Aufgabe aus und liefert das Ergebnis der ausführen Methode an den Client zurück.

Die Methode ausführenDerAufgabe() benötigt keinerlei Kenntnisse über das Ergebnis der Berechnung, der ausführen() Methode, da dieses auf jeden Fall ein (generelles) Objekt Object zurück liefert. Dieses muss allenfalls im Client noch gecastet werden, also in einen entsprechenden Datentyp oder Objekttyp umgewandelt werden.

2.3.4.3.1. Übergabe von Objekten in RMI

Argumente für oder von remote Methoden können fast von beliebigem Typus sein:
lokale Objekte, remote Objekte und elementare Datentypen.

Präziser ausgedrückt:
Parameter (Eingabe oder Rückgabe) für remote Methoden können sein:

- Instanzen einfacher Datentypen
- remote Objekte
- serialisierbare Objekte, also Objekte, die java.io.serializable implementieren

Einige Objekte erfüllen diese Bedingungen nicht und können damit weder als Eingabe- noch als Rückgabe-Parameter einer remote Methode verwendet werden. Die meisten dieser Objekte, wie beispielsweise eine Dateibeschreibung (file Descriptor), kapseln Informationen in sich, die lediglich innerhalb eines Adressraumes Sinn machen. Viele Kernklassen, also Klassen aus den Packages java.lang und java.util implementieren das Serializable Interface.

Die Regeln, nach denen Argumente und Rückgabewerte übergeben werden sind:

1. *remote Objekte* werden "*by reference*" übergeben.
   Ein remote Objekt ist ein Stub, der Client-seitige Proxy, der alle remote Interfaces implementiert, die das remote Objekt implementiert.
2. *lokale Objekte* werden "by copy" übergeben.
   Dazu wird das Objekt *serialisiert*. Standardmässig werden alle Felder des Objekts kopiert, ausser jenen, die als static oder transient deklariert werden. Das Standard-Serialisieren kann klassenweise überschrieben werden.

Die Übergabe "by reference" von remote Objekten bewirkt, dass allfällige Änderungen am remote Objekt sich auf das ursprüngliche Objekt auswirken (im Gegensatz zur Methode: "by value", bei der eine Kopie angefertigt wird, deren Werte sich nach den Methodenaufrufen vom ursprünglichen Objekt unterscheiden können).

Falls ein Objekt "by reference" übergeben wird, also ein remote Objekt, stehen dem Empfänger lediglich die Methoden des Objekts zur Verfügung, die remote Interfaces sind. Alle lokalen, bei der Implementation hinzugefügten Methoden, stehen dem Empfänger *nicht* zur Verfügung.

In unserem Beispiel würde, falls wir eine Instanz der Rechenmaschine "by reference" weitergeben würden, der Empfänger lediglich Zugriff auf die ausführen() Methode haben, also den Konstruktor und die main() Methode oder die Methoden aus java.lang.Object nicht sehen.

In remote Methodenaufrufen werden Objekte - Parameter, Rückgabewerte und Ausnahmen - welche nicht explizit remote Objekte sind, "by value" übergeben. Dies hat zur Folge, dass eine Kopie des Objekts gemacht wird und auf die Zielmaschine transferiert wird. Jede Änderung des Objektzustands auf der Empfängerseite ändert lediglich den Objektzustand auf der Empfängermaschine, nicht im Original.

2.3.4.4. Implementation der Server main Methode

Die wichtigste Methode der Rechenmaschine ist die main Methode. Die main Methode wird benötigt, um die Rechenmaschine zu starten und alle erforderlichen Initialisierungen durchzuführen sowie allfällige benötigte Vorbereitungen zu treffen, damit die Maschine Kundenanfragen bearbeiten kann. Diese Methode ist sicher keine remote Methode! Daher kann sie auch nicht von einer entfernten JVM aus aufgerufen werden. Da die Methode static ist, gehört sie auch nicht zu einem bestimmten Objekt, einer Instanz, sondern zur Klasse Rechenmaschine.

2.3.4.4.1. Kreieren und Installieren eines Security Manager

Als erstes wird in der main Methode normalerweise der Security Manager installiert. Zum Testen könnten wir diesen Schritt weglassen; aber in der Praxis muss dies getan werden, insbesondere muss dann auch eine passendes Policy Datei kreiert werden (mittels des PolicyTools beispielsweise). Der Security Manager sorgt dafür, dass heruntergeladene Objekte keinen freien Zugriff auf lokale Systemresourcen besitzen. Der Secutity Manager bestimmt, ob ein heruntergeladenens Objekt Zugriff auf das lokale Dateisystem hat oder privilegierte Operationen ausführen darf.

Alle RMI Programme müssen (müssten) einen Security Manager installieren, sonst lädt RMI keine Klassen herunter (ausser sie sind bereits lokal: zum Testen können wir also auf den Security Manager verzichten). Diese Einschränkung garantiert, dass Operationen von heruntergeladenem Code Sicherheitsprüfungen durchlaufen.

Unsere Rechenmaschine verwendet den Security Manager, der mit dem RMI System mitgeliefert wird: RMISecurityManager. Dieser Security Manager funktioniert analog zum Security Manager für Applets: er ist sehr konservativ in Bezug auf das was erlaubt ist. Es ist aber kein Problem, mit Hilfe der Policy Datei entsprechende Zugriffsrechte zu definieren.

```
if (System.getSecurityManager() == null) {
   System.setSecurityManager(new RMISecurityManager());
}
```

2.3.4.4.2. Remote Objekte den Clients zur Verfügung stellen

Als nächstes kreiert die main Methode eine Instanz der Rechenmaschine:

```
Berechne maschine = new Rechenmaschine();
```

Wie bereits mehrfach erwähnt, ruft diese Methode direkt den parameterlosen Konstruktor der Superklasse UnicastRemoteObject auf. Dieser exportiert das neu kreierte Objekt an das RMI Laufzeitsystem. Nach dem Export steht das remote Objekt Rechenmaschine Clients zur Verfügung, an einem anonymen Port, der von RMI oder dem Betriebsystem ausgesucht wird.

Zu beachten:
Die Variable ist vom Typ Berechne, *nicht* Rechnungsmaschine. Das heisst, dass das dem Client zur Verfügung gestellte Interface das Berechne Interface und seine Methoden ist, *nicht* die Rechenmaschine Klasse mit ihren Methoden.

Bevor ein Anrufer eine Methode eines remote Objekts einsetzen kann, muss er zuerst eine Referenz auf das remote Objekt erhalten. Dies kann auf die selbe Art und Weise geschehen wie man eben eine Objektreferenz erhält, beispielsweise als Teil eines Rückgabewerts einer Methde oder als Teil einer Datenstruktur, welche eine solche Referenz enthält.

Das RMI System stellt ein ganz spezielles remote Objekt zur Verfügung, die RMI Registry, mit deren Hilfe Referenzen auf remote Objekte gefunden werden können. Die RMI Registry ist ein einfacher Namenserver, welcher es einem Client gestattet eine remote Referenz zu erhalten sofern der Name des remote Objekts bekannt ist. Mit Hilfe des ersten gefundenen remote Objekts können dann weitere gefunden werden.

Das java.rmi.Naming Interface wird als Front-End API benutzt, um Namen in der Registry zu binden, zu registrieren und zu finden. Sobald ein Objekt in die Registry auf dem lokalen Rechenr gebunden ist, können Clients (oder Server) von einem beliebigen Host die remote Objekte per Namen aufrufen, die Referenz auf das remote Objekt erhalten und dann dessen remote Methoden einsetzen.

Die Registry kann von allen Servern in einem Netzwerk gemeinsam genutzt werden, oder aber nur von einem individuellen Prozess, wie auch immer praktischer.

Die Rechenmaschine kreiert einen Namen mittels folgender Anweisung:

```
String name = "rmi://host/Berechne"; // rmi: kann auch fehlen
```

Der Name umfasst also den Rechnernamen, *auf dem die Registry (und das remote Objekt) läuft*, sowie ein Name.

Berechne identifiziert das remote Objekt in der Registry.

Zusätzlich muss der Name in der RMI Registratur / Registry eingetragen werden. Dies geschieht im try ... catch Block:

```
Naming.rebind(name, maschine);
```

Der Aufruf der (bind() oder) rebind() remote Methode betrifft die RMI Registry auf dem lokalen Host. Im schlimmsten Fall führt der Aufruf zu einer Ausnahme RemoteException. Daher muss diese abgefangen werden. Dies geschieht im try ... catch Block.

Die Naming.rebind() Methode hat zwei Parameter: die Lokation und das Objekt.

- der *erste* Parameter ist eine URL-formattierte java.lang.String Darstellung der Lokation und des Namen des remote Objekts.
  Sie müssen den Namen Ihres Rechners entsprechend setzen. Im Beispiel können Sie vermutlich mit localhost arbeiten. An Stelle eines Rechenrnamens können Sie auch eine IP Adresse eingeben. Falls der Rechenrname fehlt, wird localhost angenommen
  Sie brauchen auch kein Protokoll zu spezifizieren. Im Minimum müssen Sie in unserem

Beispiel also nur den Namen des remote Objekts angeben: maschine.
Optional können Sie auch noch eine Portnummer angeben. Falls diese fehlt, wird der Standardport für RMI, 1099, verwendet. Sollten Sie mehrere RMI Registries am laufen haben, müssen Sie natürlich unterschiedliche Ports verwenden, auch im Netzwerk (falls Ihr Nachbar gerade 1099 belegt):

[rmi:]//host:1122/objektname

- Das RMI Laufzeitsystem ersetzt den Objektnamen objektname durch eine Referenz auf den Stub (das lokale Proxy Objekt). Die remote Implementationen, wie zum Beispiel die Rechenmaschine, verlassen die JVM, auf der sie kreiert werden, **nie**! Falls also ein Client ein remote Objekt in einer Registry sucht, wird eine Referenz auf das Proxy Objekt, den Stub, zurückgegeben.
- Aus Sicherheitsgründen kann eine Applikation die Methoden bind(), rebind() und unbind() für remote Objekte nur für Registries verwenden, die lokal vorhanden sind, also auf dem mselben Rechner installiert sind. Lookups können dagegen natürlich von überall her ausgeführt werden.

In unserem Beispiel gibt der Server eine Meldung aus, sobald er das Objekt registriert hat und bereit ist, zu starten und Anrufe entgegenzunehmen. Damit ist die Arbeit für die main() Methode erledigt. *Sie benötigen also keinen Thread, der den Server am Leben erhält*!

Der Server wird weiter aktiv sein, solange irgend eine Referenz von einem remote oder lokalen Client darauf verweist. Sonst startet der Server herunter gefahren und der Garbage Collector räumt ihn weg. Das RMI System kümmert sich darum, dass der Server läuft, falls Anfragen eintreffen, solange der Eintrag in der Registry vorhanden ist.

Der Rest der main() Methode dürfte klar sein:es handelt sich um die Ausnahmebehandlungen! RemoteException kann vom Konstruktor oder der Rechenmaschine beim Binden an die Registry geworfen werden.

2.3.5. Kreieren des Client Programms

Der Berechnungsserver ist ein recht einfaches Programm: sie führt Aufgaben aus, die an sie gesandt werden. Der Client ist komplexer! Er muss die Aufgabe spezifizieren und an den Server zu senden.

Für unser Beispiel benötigen wir demnach zwei Klassen:
1. die erste beschreibt die Berechnung, als Beispiel: die Berechnung von π. Wir nennen diese Klasse, eigentlich in Verletzung der allgemein gehaltenen Architektur, BerechnePi. Diese Klasse arbeitet mit der Klasse Berechne zusammen.
2. de zweite Klasse, Pi (wieder in Verletzung der allgemein gehaltenen Architektur) implementiert das Interface Aufgabe und definiert, was die Rechenmaschine zu tun hat. Pi muss den Wert der Konstanten π bis auf eine bestimmte Anzahl Dezimalen berechnen.

Schauen wir uns nun die Implementation dieser Klassen an:

```
package berechne;
public interface Aufgabe extends java.io.Serializable {
  Object ausführen();        }
```

Das Interface Aufgabe erweitert java.io.Serializable, somit kann ein Objekt, welches dieses Interface implementiert, vom RMI Laufzeitsystem serialisiert und an eine entfernte JVM gesandt werden. Den selben Effekt würden wir erzielen, wenn unsere Implementationsklassen sowohl das Aufgabe als auch das java.io.Serializable Interface implementieren würden.

Allerdings ist die Hauptaufgabe des Aufgabe Interfaces, es zu gestatten, Implementationen dieses Interfaces an ein berechne Objekt zu übergeben. Daher macht es keinen Sinn, eine Klasse zu haben, die java.io.Serializable nicht implementiert. Daher verbinden wir diese zwei Interfaces explizit, um sicher zu sein, dass wir serialisieren können

Der Programmcode, der Berechne's Methoden aufruft, muss ein eine Referenz auf dieses Objekt beschaffen, ein Aufgabe Objekt kreieren und dann verlangen, dass die Aufgabe ausgeführt wird.

Die Definition der Aufgabe Pi schauen wir uns später an. Die Konstruktion eines Pi Objects erhält einen Parameter, nämlich die Genauigkeit der Berechnung. Als Ergebnis erhalten wir einen einfachen Datentyp java.math.BigDecimal, welcher π mit der gewünschten Genauigkeit darstellt.

Die Client Klasse Client.BerechnePi sieht folgendermassen aus:

```
package Client;
import java.rmi.*;
import java.math.*;
import berechne.*;
public class BerechnePi {
    public static void main(String args[]) {
        // Security Manager setzen
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new RMISecurityManager());
    }
    try {
        // Register und remote Objekt args[0] ist der Registry Host
    String name = "//" + args[0] + "/Berechne";
        // bestimmen der remote Referenz zum Objekt 'Berechne'
    Berechne berech = (Berechne) Naming.lookup(name);
        // Genauigkeit der Berechnung
    Pi aufgabe = new Pi(Integer.parseInt(args[1]));
    BigDecimal pi = (BigDecimal) (berech.ausführenDerAufgabe(aufgabe));
    System.out.println(pi);
    } catch (Exception e) {
      System.err.println("BerechnePi Exception: " +  e.getMessage());
      e.printStackTrace();
    }
    }
}
```

Wie bei der Rechenmaschine beginnt das Programm mit dem Setzen eines Security Managers. Auch hier ist es nötig, weil RMI den Code zum Client herunterladen kann. In diesem Beispiel kann beispielsweise der Stub der Rechenmaschine zum Client heruntergeladen werden.

Jedesmal wenn Programmcode zum Client heruntergeladen wird, muss ein Security Manager geladen sein. Also muss auch beim Client, wie beim Server ein Security Manager präsent sein.

Nach dem Installieren eines Security Managers konstruiert der Client einen Namen, mit dem er dann in der Registry das remote Objekt nachschaut.

Beim Starten des Clients werden auch zwei Parameter mitgegeben: args[0] und args[1]. Das erste Argument, args[0], ist der Name des remote Host, auf dem das Berechne Objekt läuft. Der Client benutzt die Naming.lookup Methode, um das remote Objekt zum Namen in der Registry des remote Hosts zu finden. Dazu bildet der Client eine URL (mit dem rmi Protokoll), welche angibt, wo der Berechnungsserver sich befindet. Die Syntax der URL ist und muss identisch sein, wie beim Binden an die Registry, beim Aufruf Naming.rebind.

```
// Register und remote Objekt args[0] ist der Registry Host
String name = "//" + args[0] + "/Berechne";

Berechne berech = (Berechne) Naming.lookup(name);
```

Als nächstes kreiert der Client ein neues Pi Objekt, wobei der zweite Client Parameter args[1] verwendet wird, um die Genauigkeit der Berechnung von π anzugeben.

```
Pi aufgabe = new Pi(Integer.parseInt(args[1]));
```

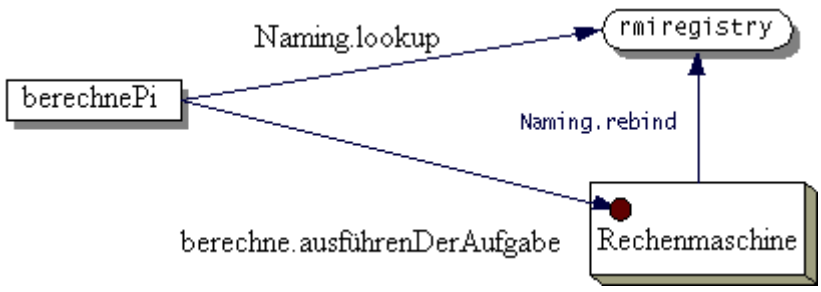
Schliesslich ruft der Client die Methode ausführenDerAufgabe() der Klasse Berechne und deren remote Objekt berech aus dem Namimg.lookup().

```
BigDecimal pi = (BigDecimal) (berech.ausführenDerAufgabe(aufgabe));
```

Als Argument wird der Methode ausführenDerAufgabe() ein Pi Objekt mitgegeben. Dieses liefert ein Objekt vom Typus java.math.BigDecimal. Auf diesen Datentyp muss das Ergebnis am Schluss wieder gecastet werden, mit (BigDecimal).

Schliesslich gibt der Client das Ergebnis aus.

Das folgende Diagramm zeigt schematisch, wie die einzelnen Meldungen / Methodenaufrufe zusammenhängen.



Nun müssen wir nur noch die Klasse Pi verstehen.

Diese Klasse implementiert das Interface Aufgabe und berechnet den Wert von π mit der verlangten Genauigkeit. Der eigentliche Algorithmus der Berechnung interessiert hier

eigentlich nicht. Einzig die Genauigkeit, die wir als Parameter vorgegeben haben, muss eingehalten werden. Für unser Beispiel ist wesentlich, dass die Berechnung einige Zeit in Anspruch nimmt, also die Aufgabe sinnvollerweise an einen Spezialrechner delegiert werden sollte.

Und so sieht das Programm aus:

```
package Client;

import berechne.*;
import java.math.*;

public class Pi implements Aufgabe {

   /** Konstanten zur Berechnung von Pi*/
   private static final BigDecimal ZERO = BigDecimal.valueOf(0);
   private static final BigDecimal  ONE = BigDecimal.valueOf(1);
   private static final BigDecimal FOUR = BigDecimal.valueOf(4);

   /** Rundung bei der Berechnung festlegen */
   private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;

   /** Präzision der Berechnung, nach dem Dezimalpunkt */
   private int digits;

   /**
   * Konstruktion einer Task, welche Pi mit der
   * verlangten Genauigkeit berechnet
   */
   public Pi(int digits) {
      this.digits = digits;
   }

   /**
    * Berechne pi.
    */
   public Object ausführen() {
      return berechnePi(digits);
   }

   /**
    * Berechne den Wert von Pi mit der angegebenen Anzahl
    * Stellen nach dem Dezimalpunkt.  Der Wert wird mit Hilfe der
    * Machin's Formel berechnet:
    *
    *         pi/4 = 4*arctan(1/5) - arctan(1/239)
    *
    * und einer Reihenentwicklung von arctan(x) bis zur
    * ausreichenden Genauigkeit.
    */
```

```
    public static BigDecimal berechnePi(int digits) {
        int scale = digits + 5;
        BigDecimal arctan1_5 = arctan(5, scale);
        BigDecimal arctan1_239 = arctan(239, scale);
        BigDecimal pi = arctan1_5.multiply(FOUR).subtract(
                            arctan1_239).multiply(FOUR);
        return pi.setScale(digits,
                        BigDecimal.ROUND_HALF_UP);
    }

    /**
     * Berechne den Wert des ArcTangens
     * Der wert wird mit Hilfe der Reihenentwicklung berechnet
     *
     * arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 +
     *     (x^9)/9 ...
     */

    public static BigDecimal arctan(int inverseX,
                            int scale)
    {
        BigDecimal result, numer, term;
        BigDecimal invX = BigDecimal.valueOf(inverseX);
        BigDecimal invX2 =
            BigDecimal.valueOf(inverseX * inverseX);

        numer = ONE.divide(invX, scale, roundingMode);

        result = numer;
        int i = 1;
        do {
            numer =
                numer.divide(invX2, scale, roundingMode);
            int denom = 2 * i + 1;
            term =
                numer.divide(BigDecimal.valueOf(denom),
                            scale, roundingMode);
            if ((i % 2) != 0) {
                result = result.subtract(term);
            } else {
                result = result.add(term);
            }
            i++;
        } while (term.compareTo(ZERO) != 0);
        return result;
    }
}
```

Das interessante am Beispiel ist, dass das Berechne Objekt die Klassendefinition von Pi's Klasse erst zum Zeitpunkt benötigt, wenn ein Pi Objekt als Argument in der ausführenDerAufgabe() verwendet wird.

Zu diesem Zeitpunkt wird die Klasse bzw. das Objekt von RMI in die JVM des Berechne Objekts geladen, die ausführen() Methode wird aufgerufen und der Programmcode ausgeführt..

An Stelle der Berechnung von π könnten wir auch speziell schwer zu berechnende Sicherheitsschlüssel oder Zufallszahlen berechnen.

2.3.6. Übersetzen und Ausführen des Beispiels

Nachdem wir nun den Programmcode gesehen und vermutlich zumgrossen Teil verstanden haben, müssen wir die Class Dateien generieren, mit javac und anschliessend die Programme ausführen, mit java und rmiregistry.

2.3.6.1. Übersetzen der Quellen der Beispielprogramme

Typische Vorgehensweise in RMI Projekten:
Ein Szenario für den praktischen Einsatz eines Berechnungsservers wäre, dass der Entwickler eine JAR Datei (ein Java ARchiv) kreiert, in der die Interfaces für die Aufgabe und Berechne enthalten sind. Diese Interfaces werden für die Implementation der Server Klassen benötigt, und werden zudem von Client Programmen genutzt.

Als nächstes würde der Entwickler die Interfaces implementieren und möglichen Clients auf einem Server zur Verfügung stellen.

Der Entwickler des Client Programms kann die Interfaces benutzen, um unabhängig von der Server Seite den Client entwickeln, welcher den Server benutzt.

In diesem Abschnitt erstellen wir übersetzen wir die Programme und erstellen die JAR Dateien. Die Klasse Pi, welche wir auf der Client Seite entwickeln, wird zur Laufzeit zum Server heruntergeladen. Der Stub wird zur Laufzeit vom Server zum Client herunter geladen.

Wir haben unser System, wie bereits aufgezeigt, in drei Packages aufgeteilt:
1. berechne:
   Berechne und Aufgabe Schnittstellen
2. Berechnungsserver:
   Rechenmaschine Implementation und Stub
3. Client:
   berechnePi und Pi Aufgabe Implementation

2.3.6.1.1. Erstellen der JAR Datei mit den Interface Klassen

Als erstes erstellen wir das Archiv mit den Interfaces. Diese werden für die Entwicklung des Clients und des Servers benötigt. Bevor wir archivieren können, müssen wir die Dateien im Verzeichnis berechne, die Dateien des Packages berechne, übersetzen. Aus den Class Dateien bilden wir dann das JAR Archiv.

Im Verzeichnis berechne finden Sie die compile_berechne.bat Datei:

```
@echo off
javac  *.java
```

pause

Als nächstes archivieren wir die generierten Class Dateien. Dazu befindet sich im berechne Verzeichnis die make_jar.bat Datei:

```
@echo off
cd ..
jar cvf berechne.jar berechne\*.class
cd berechne
pause
```

Die Optionen cvf des jar Befehls haben folgende Bedeutung:

1. -v : verbose Output
   Ausgabe der Aktivitäten
   added manifest
   adding: berechne/Berechne.class (in=281) (out=196)
   (deflated 30%)
   adding: berechne/Aufgabe.class (in=200) (out=164)
   (deflated 18%)
2. -c : create a new archive
   ein allfällig bereits vorhandenenes Archiv wird überschrieben
3. -f : Angabe der Archivdatei

Das Archiv wird im übergeordneten Verzeichnis von berechne generiert und abgespeichert. Es soll den Entwicklern der Client Applikation und der Server Implementation zur Verfügung stehen.

Die Class Dateien, serverseitig und clientseitig, müssen in ein Verzeichnis gestellt werden, aus dem sie mittels eines Webservers heruntergeladen werden können. In der Regel sind dies Verzeichnisse der Form:

unter Windows:
file:/c:/home/user/public_html/classes/ oder file:/c:/webroot/....
unter Unix:
file:/home/user/public_html/classes/

Wichtig ist einfach daran zu denken, dass die Klassen dynamisch herunter geladen werden können und müssen und daher dem Webserver zur Verfügung stehen müssen. Sun stellt einen minimalen Webserver zur Verfügung, welcher im wesentlichen nichts anderes kann, als Klassen herunterzuladen:

ftp://ftp.javasoft.com/pub/jdk1.1/rmi/class-server.zip

2.3.6.1.2. Übersetzen der Serverseite

Da der Server die Interface Klassen benötigt, müssen wir beim Übersetzen den CLASSPATH passend setzen. Generell geht man dazu über, den CLASSPATH auf der Kommandozeile anzugeben, da im andern Fall oft Probleme entstehen, weil Klassen aus dem CLASSPATH und nicht dynamisch über das Web geladen werden.

```
@echo off
```

```
javac -classpath .;.. *.java
pause
```

In unserem Fall geht es darum, das berechne.jar Archiv im CLASSPATH zu haben. Dies erreichen wir mit ";.." , da sich das Archiv im übergeordneten Verzeichnis befindet. Sie finden Hinweise zum Setzen der CLASSPATH Variable, speziell in Java 2, auf der Seite http://java.sun.com/products/jdk/1.3/docs/tooldocs/win32/classpath.html oder in Ihrer JDK Dokumentation.

Als nächstes müssen wir die Stubs generieren. Dies geschieht mit Hilfe des rmic Compilers. Diesem kann man verschiedene Parameter mitgeben, die beispielswiese die JDK Version berücksichtigen (ab JDK1.2 werden Skeletons nicht mehr benötigt), oder eine Verzeichnisangabe, um anzugeben wo die Class Dateien gespeichert werden sollen.

Stub und Skeleton müssen allgemein zugänglich sein, sollten am Schluss also in einem übergeordneten verzeichnis sein.

```
@echo off
cd ..
rmic -classpath .;Berechnungsserver Berechnungsserver.Rechenmaschine
pause
cd Berechnungsserver
```

rmic löst den Packagenamen auf, dh. falls wir rmic im Verzeichnis ....\Berechnungsserver starten, wird er die Klasse nicht finden:

```
\Berechnungsserver>rmic -classpath .  Rechenmaschine
error: File .\Rechenmaschine.class does not contain type Rechenmaschine as expected, but t
ype Berechnungsserver.Rechenmaschine. Please remove the file, or make sure it appears in
the correct subdirectory of the class path.
error: Class Rechenmaschine not found.
2 errors
```

Falls unsere Packages im Verzeichnis Berechnung sind, und dies das netzwerktaugliche Verzeichnis ist, müssen wir Stubs, Skeletons und die Interfaces in dieses Verzeichnis kopieren und (zur Sicherheit) die Interfaces aus dem Archiv entnehmen (in der Regel kann ein Webserver oder ein RMI Programm JAR Dateien lesen):

```
jar xvf berechne.jar

created: META-INF/
extracted: META-INF/MANIFEST.MF
extracted: berechne/Berechne.class
extracted: berechne/Aufgabe.class
```

2.3.6.1.3. Übersetzen der Client Klassen

Auch hier müssen wir den CLASSPATH so setzen, dass der Java Compiler die Klassen aus dem berechne Package findet.

```
@echo off
```

```
javac -classpath .;.. pi.java
Rem
Rem in . befindet sich pi.class
Rem in .. befindet sich das jar File des berechne Package
Rem
javac -classpath .;.. BerechnePi.java
pause
```

Die Pi Class Datei muss allgemein verfügbar sein, da sie von der Rechenmaschine benötigt wird.

2.3.6.2. Starten der Beispielprogramme

2.3.6.2.1. Eine Bemerkung betreffend Security

Da sich ab JDK 1.2 das Sicherheitsmodell wesentlich geändert hat, muss bei aktivem Security Manager jeweils eine Policy Datei erstellt werden. Dies geschieht am einfachsten mittels des Policytools.

Sie haben beim Beispiel zwei unterschiedliche Policy Dateien mit unterschiedlichen Privilegien

1. Verbindung zum Port 80 (Port für HTTP)
   Verbindung zu unprivilegierten Ports (Ports grösser als 1024)

   ```
   grant {
            permission java.net.SocketPermission "*:1024-65535","connect,accept";
            permission java.net.SocketPermission "*:80", "connect";
   };
   ```

   falls das Herunterladen der Klassen mit Hilfe eines HTTP Server ermöglicht wird, dann ist diese Policy die am besten geeignete.

2. falls Sie mit URLs arbeiten wollen, ist die folgende Variante günstiger (in Windows müssen Sie bei der Verzeichnisangabe zwei Backslash Zeichen verwenden

   ```
   grant {
          permission java.net.SocketPermission "*:1024-65535", "connect,accept";
       permission java.io.FilePermission "D:\\RMI\\Berechnen\\classes\\-", "read";
       permission java.io.FilePermission "D:\\RMI\\Berechnen\\classes\\-", "read";
   };
   ```

In diesen Beispielen wird vorausgesetzt, dass die Security Policies in einer Datei java.policy stehen. Falls Sie entgegen allen Vermutungen noch mit JDK 1.1 arbeiten, dann brauchen Sie sich in Bezug auf Security keine Sorgen zu machen, da das Security Modell damals noch viel einfacher war.

2.3.6.2.2. Starten des Servers

Bevor die Rechenmaschine gestartet werden kann, muss die RMI Registry gestartet werden. Die RMI Registry stellt, wie bereits erwähnt, einem einfachen Bootstrap Namensdienst zur Verfügung und muss auf dem Server laufen. Zudem darf in der Konsole, in der rmiregistry gestartet wird, der CLASSPATH nicht gesetzt werden oder dieser nicht auf Stubs oder sonstige Projektklassen verweist. Falls dies der Fall wäre, würde sich die RMI Registry nicht daran erinnern, wo diese Klassen sind und sie somit bei Anforderung nicht finden.

```
@echo off
set CLASSPATH=
start rmiregistry
```

Standardmässig startet dieser Dienst an Port 1099. Falls Sie dies nicht wollen, müssen Sie einen andern Port angeben:

```
@echo off
set CLASSPATH=
start rmiregistry 2001
```

Nachdem die Registry gestartet ist, müssen Sie sich um den Server kümmern.

```
@echo off
Rem
Rem Original
Rem java -
Djava.rmi.server.codebase=file://"D:\UnterrichtsUnterlagen\ParalleleUndVerteilteSysteme\K
apitel16 RMI\Programme\Berechnung"/ -Djava.rmi.server.hostname=localhost  -
Djava.security.policy=java.policy

Berechnungsserver.Rechenmaschine
Rem
Rem modifiziert
Rem
java -Djava.security.policy=java.policy Berechnungsserver.Rechenmaschine
Rem das reicht schon : jetzt müsste das remote Objekt gebunden werden
pause
```

Da der Server im localhost und im korrekten Verzeichnis gestartet wird, benötigen wir die Zusatzangaben, die oben auskommentiert sind, nicht.

Sonst muss mit der -Djava.rmi.server.codebase die Adresse / Codebase angegeben werden. Von dort werden dann Stubs und die Berechne und Aufgabe Interfaces heruntergeladen.

Falls Sie im Netzwerk arbeiten, müssen Sie auch noch die Property -Djava.rmi.server.hostname=... angeben.

2.3.6.2.3. Starten des Client

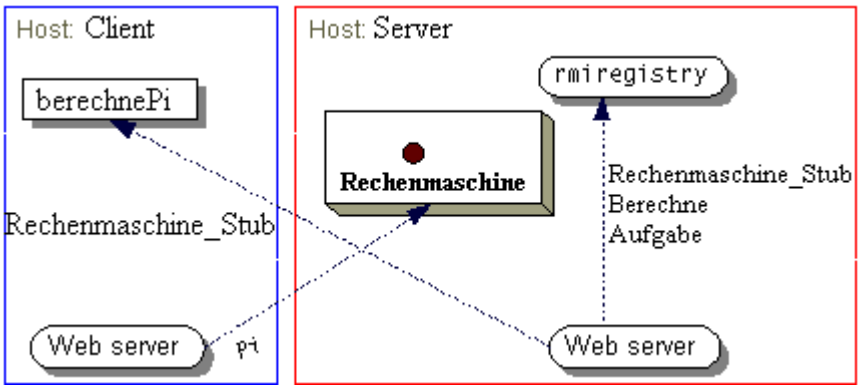
Nachdem die Registry und der Server gestartet wurden, bleibt nur noch der Client.

```
@echo off
Rem
Rem Original
Rem java -
RemDjava.rmi.server.codebase=file://D:\UnterrichtsUnterlagen\ParalleleUndVe
rteilteSystemRem e\Kapitel16RMI\Programme\Berechnung/ -
Djava.rmi.server.hostname=localhost
Rem - Djava.security.policy=java.policy Client.BerechnePi 15
Rem
Rem modifiziert
Rem
java -Djava.security.policy=java.policy Client.BerechnePi localhost 25
Rem : jetzt folgt die Ausgabe
pause
```

Da wir auch den Client im selben Verzeichnis starten wie den Server und auf dem selben Host, benötigen wir die codebase und hostname Properties nicht. Wie Sie sehen, erschein die Klasse Pi nirgends.

Falls wir den Client aus einem andern Verzeichnis starten, müssen wir diese und eventuell auch den CLASSPATH angeben. Der CLASSPATH muss so gesetzt sein, dass das JAR Archiv mit den Interfaces im Zugriff ist (bei uns ist dies sicher der Fall).

Die Angabe des Berechnungsservers und der Genauigkeit sind ebenfalls noch nötig.
Testen Sie den Server mit hohen Genauigkeiten, beispielsweise 5000. Dann dauert die Berechnung einige Zeit. Aber das Ganze funktioniert.
Ein mögliches Ergebnis:
3.14159265358979323846
Schauen wir uns nochmals das Ganze schematisch an:



beim Registrieren der remote Objektreferenz in der Registry, lädt die Registry den Stub und die Interfaces herunter.

Die Klasse berechnePi lädt den Rechenmaschine_Stub vom Webserver (oder Dateiserver, falls in der URL file://.. steht). Den Server findet der Client aufgrund des Lookups in der Registry.

Pi wird automatisch in die Rechenmaschine geladen, von Client Host.

Tolle Sache! Und funktionieren tut's auch. Es sind allerdings viele Fehler möglich. Viele gehen auf falsch oder überhaupt gesetzte CLASSPATH Variablen zurück. Vergessen Sie

nich: seit JDK 1.2 setzt man CLASSPATH in der Regel nur auf der Kommandozeile und nicht als Umgebungsvariable.

2.3.6.2.4. Mögliche Fehlermeldung

Falls Sie eine Fehlermeldung der folgenden Art erhalten (hier für das RMI Programm HelloWorld):

```
HelloImpl err: RemoteException occurred in server thread; nested exception
is:
        java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
        java.lang.ClassNotFoundException: HelloWorld.HelloImpl_Stub
java.rmi.ServerException: RemoteException occurred in server thread; nested
exception is:

        java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
        java.lang.ClassNotFoundException: HelloWorld.HelloImpl_Stub
java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
        java.lang.ClassNotFoundException: HelloWorld.HelloImpl_Stub
java.lang.ClassNotFoundException: HelloWorld.HelloImpl_Stub
        at
sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(Unknown
Source)
        at sun.rmi.transport.StreamRemoteCall.executeCall(Unknown Source)
        at sun.rmi.server.UnicastRef.invoke(Unknown Source)
        at sun.rmi.registry.RegistryImpl_Stub.rebind(Unknown Source)
        at java.rmi.Naming.rebind(Unknown Source)
        at HelloWorld.HelloImpl.main(HelloImpl.java, Compiled Code)
HelloImpl.main: Ende
```

dann haben Sie vermutlich die Registry nicht aus dem root Verzeichnis des Packages gestartet (hier also: nicht aus der Oberverzeichnis von ...\HelloWorld\: die Fehlermeldung stammt nicht aus diesem Beispiel sondern aus einem HelloWorld Demoprogramm).

2.4. Spezialthemen

Die folgenden Abschnitte behandeln Spezialthemen:

1. Socket Factory
   dies zeigt, wie Sie die Standardsockets von RMI durch eigene ersetzen können und damit auch gleich noch die Kommunikation sicherer / verschlüsselt abwickeln können. Als Beispiel besprechen wir, wie eine Verbindung mit Hilfe von SSL realisiert werden könnte.
2. Activation
   diese Thema ist besonders interessant, da Sie damit Objekte "schlafend" setzen können und zu einem späteren Zeitpunkt wieder aufwecken können.
3. Dynamic Code Downloading
   in diesem Abschnitt besprechen wir die codebase Property etwas genauer, da wir sie oben umgangen haben.
4. Factory Pattern
   ist eine kurze Besprechung über den Einsatz des Entwurfsmusters "Factory" im Rahmen von RMI Applikationen
5. RMI over IIOP
   diese relativ neue Art der RMI Kommunikation ist in einem heterogenen Umfeld wichtig und gestattet die Einbindung von RMI in ein CORBA Umfeld.

Soweit die Planung. Machen wir uns an die Arbeit!

2.4.1. Definition und Implementation einer eigenen SocketFactory

Schauen wir uns an, wie man die Standard SocketFactory von RMI durch eine eigene ersetzen kann. Diese Möglichkeit hatte man ursprünglich nicht vorgesehen; aber insbesondere für sichere Kommunikation (SSL und ähnliche) muss man solche Möglichkeiten haben.

Sie haben sogar die Möglichkeit nicht TCP basierte, oder eigene Transportprotokolle zu verwenden, statt TCP und java.net.Socket, welche RMI standardmässig einsetzt.

In einem früheren Release von JDK konnte man zwar eine eigene `RMISocketFactory` zu definieren, allerdings war dies nicht auf der Ebene einzelner Objekte möglich: man konnte nicht das eine Objekt mit einer SSL `RMISocketFactory`, ein anderes Objekt aber mit der Standard JRMP behandeln. Zudem musste jeweils pro Protokoll eine eigene Registry benutzt werden. Ab JDK1.2 ist es möglich flexibler vorzugehen: pro Objekt kann eine eigene Factory definiert werden.

Jetzt wollen wir drei Beispiele ansehen:

1. eine RMI Factory, welche lediglich einen Socket Typ generiert
2. eine RMI Factory, welche mehrere Socket Typen herstellen kann
3. ein Beispiel für den praktischen Einsatz einer RMI Socket Factory

Da eine sichere Kommunikation für viele Anwendungen wichtig ist, wurden speziell für die SSL Kommunikation entwickelt. Einige der Bibliotheken sind allgemein verfügbar, einige sind ausserhalb der USA offiziell nicht erhältlich, noch nicht.

RMI und SSL

In der RMI User List finden Sie beliebig viele Fragen zum Thema SSL und RMI. Falls Sie sich auf dem Laufenden halten möchten, sollten Sie sich in diese Liste eintragen. In diesem Einschub fassen wir lediglich einige der Standardfragen und deren Beantwortung in eben dieser User List zusammen. Sie finden weitere Antworten auf dem Java Site von Sun.

1. **Frage**: ist es möglich SSL mit RMI zu benutzen?
   Anwort: Ja, es ist möglich, wegen einigen Erweiterungen in Java2 ab der Version JDK1.2

2. **Frage**: und wie setzt man SSL konkret ein?
   Antwort: Die RMI Version in JDK1.2 oder neuer erlaubt es dem RMI Entwickler eigene Sockets einzusetzen. In den früheren Versionen konnte man lediglich eine Socket-Version pro JVM einsetzen. Ab Version JDK1.2 kann pro Objekt eine spezielle Socket-Version eingesetzt werden. Alles was man zu tun hat, ist eine RMI Socket Factory zu installieren. Diese Factory kann dann einen Socket kreieren, mit dessen Hilfe die gewünschte Kommunikation möglich wird. Wie man eine spezielle Socket Factory kreiert und in RMI einbindet, besprechen wir ausserhalb dieses Einschubs.
   Wie man konkret in Java SSL implementiert lernen Sie in den folgenden Beispielen kennen.

3. **Frage**: Gibt es eine Sun Implementation für SSL in Java?
   Antwort: Sun verwendet im HotJava, Java WebServer und JavaServer Toolkit eine Sun Implementation von SSL. Zudem gibt es eine Java Implementation von SSL von Sun, die Java Secure Socket Extension JSSE. Diese ist ausserhalb der USA nicht legal erhältlich. Sie müssen also per email eine Version von Bekannten in den USA bestellen, oder die Implementation der Universität Graz verwenden. Die ist gratis und erst noch äusserst professionell (sprich mindest so gut, wie die Sun Implementation). Wir kommen weiter unten darauf zurück!

4. **Frage**: empfiehlt Sun eine bestimmte 3rd Party Java Implementation von SSL?
   Antwort: Sun verwendet keine 3rd Party Implementation, stellt aber auf dem Java Web eine Liste der gängigen Implementationen zur Verfügung.

   Die meisten Implementtaionen verwenden den RSA Algorithms, der auch publiziert wurde, also frei erhältlich ist. Da die USA immer noch im kalten Krieg leben ist der Export von "war important" Software eingeschränkt. Die Liste der SSL Lösungsanbieter besteht daher aus zwei Teilen: einer USA und einer nicht-USA Liste.

   Zuerst die US Liste:
   Phaos Technology's SSLava Toolkit http://www.phaos.com/main.htm
   Im Kit ist ein Beispiel enthalten, welches RMI und SSL einsetzt.

   Und hier die Nicht-US Liste der SSL Implementationen:
   Protekt SSL Software from Forge Information Technology http://www.protekt.com
   Baltimore Technologies' J/SSL http://www.baltimore.ie/products/jssl/index.html
   IAIK's iSaSiLk http://jcewww.iaik.tu-graz.ac.at/iSaSiLk/isasilk.htm

2.4.1.1. Kreieren einer RMI Socket Factory, welche genau einen Socket-Typus erzeugt

Das Vorgehen, zur Implementation einer eigenen RMI Socket Factory, welche lediglich einen bestimmten Socket Typ kreiert, besteht aus folgenden 5 Schritten:

1. Entscheidung: welchen Socket Typus soll hergestellt werden?
2. Schreiben einer Client Socket Factory, welche `RMIClientSocketFactory` implementiert
3. Implementieren der `RMIClientSocketFactory` `createSocket` Methode
4. Schreiben einer Server Socket Factory, welche `RMIServerFactory` implementiert
5. Implementieren der `RMIServerSocketFactory` `createServerSocket` Methode

Schritt 1:
Entscheidung: welchen Socket Typus soll hergestellt werden?
Welcher Socket Typus hergestellt werden soll hängt von der Applikation ab:

- Falls der Server sensitive Daten behandeln muss, sollten Sie die Daten besser verschlüsseln.
- Falls der Server Videodaten übertragen muss, sollten Sie die Daten besser komprimieren

Im folgenden Beispiel komprimieren wir die Daten und bauen die entsprechende Socket Factory.

2.4.1.1.1. Kreieren eines eigenen Socket Typus

Oft müssen die Daten, welche von einem Socket von `java.net.Socket` gesendet oder empfangen werden, aufbereitet oder nachbereitet werden. In diesen Fällen bietet es sich an, eigene Sockets zu definieren. Oben haben wir zwei Beispiele angegeben, in denen dies eindeutig der Fall ist: Kompression und Verschlüsselung.

Im Folgenden lernen Sie, wie ein solcher eigener Socket Typ kreiert werden kann speziell für die Kompression: die Daten werden vor dem Übermitteln komprimiert.

Das Vorgehen zur Implementation dieser speziellen Sockets aus mehreren Schritten:

1. erweitern des `java.io.FilterOutputStream`, um einen passenden Output Stream für den Socket zu kreieren. Die Methoden der Klasse müssen eventuell passend überschritten werden.
2. erweitern des `java.io.FilterInputStream`, um einen passenden Input Stream für den Socket zu kreieren. Falls nötig müssen Methoden der Klasse angepasst werden.
3. kreieren einer Subklasse von `java.net.Socket`. Implementieren Sie die passenden Konstruktoren und überschreiben Sie die Methoden `getInputStream`, `getOutputStream` und `close`.
4. kreieren einer `java.netServerSocket` Unterklasse. Implementieren des Konstruktors und überladen der `accept` Methode, um Sockets des gewünschten Typus kreieren zu können.

Schritt 1:
erweitern des `java.io.FilterOutputStream`

Da wir eine Socket Klasse kreieren, die Daten komprimieren, nennen wir die Klasse `CompressionOutputStream`. Diese erweitert `FilterOutputStream`. Es ist nicht immer ratsam `FilterOutputStream` zu erweitern. Sie sollten jenen Output Stream erweitern, der

bereits am dichtesten an jenen herankommt, den Sie implementieren möchten. In unserem Fall ist `FilterOutputStream` am besten geeignet.

Der Einfachheit halber verwenden wir einen Algorithmus, den man in der Praxis nicht einsetzen würde. Dieser verschlüsselt bis zu 62 Zeichen mit 6 Bits und 18 Bit Verschlüsselung für alle andern Zeichen. Java verwendet per Standard eine 16 Bit Darstellung.

Wir verwenden eine Lookup Tabelle, welche die 62 gängigen Zeichen verschlüsselt (o...61). Damit wir wieder entschlüsseln können, müssen wir eine Konstante voran setzen, aus der ersichtlich ist, ob es sich um eine 6 Bit oder eine 18 Bit Verschlüsselung handelt. Es wird vorausgesetzt, dass alle Zeichen, die wir im Eingabe / Ausgabe Strom verschlüsseln, ASCII Zeichen sind und das obere Byte nicht benutzt wird.

Als erstes schreiben wir ein Interface, welches alle Konstanten enthält, die wir für Input und Output Stream benötigen. In unserem Beispiel ist dies sinnvoll, da wir eine Lookup Tabelle definieren müssen.

```
package kompression;

interface CompressionConstants {

    /** Konstanten:  6-Bit Verschlüsselung. */

    /** No Operation */
    static final int NOP     = 0;
    /** Standard Byte Format. */
    static final int RAW     = 1;
    /** Zeichen ist in Lookup Tabelle. */
    static final int BASE    = 2;

    /** Der Zeichencode ist die Position in der Lookup Tabelle. */
    static final String codeTable =
        "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ ,.!?\"'()";
}
```

Jetzt müssen wir den 1.Schritt der Konstruktion abschliessen, also die Klasse `java.io.FilterOutputStream` erweitern und die Methoden der Klasse entsprechend überschreiben. Die Klasse nennen wir `CompressionOutputStream`.

Der Algorithmus wird im Kommentar der Klasse beschrieben, ist aber, wie bereits erwähnt, eher unwichtig und für die Praxis nicht geeignet.

```
package kompression;

import java.io.*;

class CompressionOutputStream extends FilterOutputStream
    implements CompressionConstants
{

    /*
     * Konstruktor ruft den Konstruktor der Oberklasse auf.
     */
    public CompressionOutputStream(OutputStream out) {
        super(out);
```

```
}

/*
 * Buffer der 6-Bit Codes in die das 32-Bit Wort gepackt wird.
 * Fünf 6-Bit Codes passen in 4 Worte.
 */
int buf[] = new int[5];

/*
 * Index der gültigen Codes in buf.
 */
int bufPos = 0;

/*
 * Diese Methode schreibt ein Byte in den Socket Stream.
 */
public void write(int b) throws IOException {
    //
    b &= 0xFF;

    // Look Position in der Code Tabelle codeTable,
    // um die Verschlüsselung
    // zu erhalten.
    int pos = codeTable.indexOf((char)b);

    if (pos != -1){
        // Falls pos in the codeTable ist, schreib BASE + pos in buf.
        // BASE zeigt, dass pos Zeichen in
        // der codeTable enthalten ist und einen Code zwischen
        //  2 und 63
        // inklusive besitzt.
        // RAW ist gleich 1.

        writeCode(BASE + pos);
    } else {
        // schreib RAW in buf:
        // das Zeichen wird in 12 Bits gesendet.
        writeCode(RAW);

        // schreib 4 Bits von b in buf.
        writeCode(b >> 4);

        // b modifizieren, so dass nur die ersten 4 Bits enthalten
        // sind,
        // schreib die ersten 4 Bits von b in buf.
        writeCode(b & 0xF);
    }
}

/*
 * Diese Methode schreibt bis zu len Bytes in den Socket Stream.
 */
public void write(byte b[], int off, int len) throws IOException {
    /*
     * Diese Implementation ist recht ineffizient, da sie
     * die write Methode für jedes Byte im Array aufrufen muss.
     */
    for (int i = 0; i < len; i++)
        write(b[off + i]);
}
```

```
   /*
    * Rücksetzen des Buffers.
    */
  public void flush() throws IOException {
    while (bufPos > 0)
      writeCode(NOP);
  }

    /*
     * Diese Methode schreibt die Daten in den Output Stream nachdem
     * die Daten aller 5 Bytes in buf in ein Wort geschrieben sind.
     */
    private void writeCode(int c) throws IOException {
        buf[bufPos++] = c;
        if (bufPos == 5) {      // 5 Codes
            int pack = (buf[0] << 24) | (buf[1] << 18) | (buf[2] << 12) |
                       (buf[3] << 6) | buf[4];
            out.write((pack >>> 24) & 0xFF);
            out.write((pack >>> 16) & 0xFF);
            out.write((pack >>> 8)  & 0xFF);
            out.write((pack >>> 0)  & 0xFF);
            bufPos = 0;
        }
    }
}
```

Als erstes wird die Klasse `FilterOutputStream` erweitert und das Interface `CompressionConstants` implementiert. Damit steht die Lookup Tabelle und die Konstanten zur Verfügung.

Um die Daten zu komprimieren, wird die `write` Methode überschrieben:

`public void write(int b)`
Diese Methode schreibt pro Aufruf ein Zeichen und fügt dem Zeichen eine Kompressionskonstante hinzu: RAW oder BASE, und teilt das Zeichen in zwei 4 Bit Teile auf.

Um mehrere Zeichen zu schreiben, wird die Methode

`public void write(byte b[], int off, int len)`
eingesetzt. Sie schreibt `len` Zeichen und verwendet die `write` Methode,welche ein Zeichen pro Aufruf schreibt.

Schliesslich steht eine Methode zur Verfügung

`private void writeCode(int c)`
packt fünf 6 Bit Codes in ein Wort, welches bis zu fünf Zeichen verschlüsseln könnte, und schreibt das Wort in den Ausgabestrom.

Schritt 2:
erweitern des `java.io.FilterInputStream`

Nachdem wir einen Ausgabestrom haben müssen wir uns um den Eingabestrom kümmern. Diese Aufgabe ist der obigen sehr nahe, wie man dem Quellcode entnehmen kann.

```
package kompression;
```

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

class CompressionInputStream extends FilterInputStream
    implements CompressionConstants
{
    /*
     * Konstruktor: ruft den Oberklassen-Konstruktor auf
     */
    public CompressionInputStream(InputStream in) {
        super(in);
    }

    /*
     * Buffer : entpackte 6-Bit Codes
     * der letzten 32 gelesenen Bits.
     */
    int buf[] = new int[5];

    /*
     * Position des nächsten Codes im Buffer (5 : Ende).
     */
    int bufPos = 5;

    /*
     * Lesen der Format Code und Dekompression der Zeichen.
     */

    public int read() throws IOException {
        try {
            int code;

            // Lesen und ignorieren der leeren (NOP's)
            do {
            code = readCode();
          } while (code == NOP);

            if (code >= BASE) {
                // Index des Zeichens in der codeTable
                // falls der Code im Bereich liegt.
                return codeTable.charAt(code - BASE);
            } else if (code == RAW) {
                // lesen der unteren 4 Bits und der oberen 4 Bits,
                // Rückgabe des rekonstruierten Zeichens
                int high = readCode();
                int low = readCode();
                return (high << 4) | low;
            } else
            throw new IOException("unbekannter Kompressionscode: " + code);
        } catch (EOFException e) {
            // EOF Code
            return -1;
        }
    }

    /*
     * Method liest bis zu len Bytes vom Input Stream.
     * Abbruch, falls read vor dem Lesen von len Bytes blockiert.
     */
    public int read(byte b[], int off, int len) throws IOException {
```

```
      if (len <= 0) {
          return 0;
      }

      int c = read();
      if (c == -1) {
          return -1;
      }
      b[off] = (byte)c;

      int i = 1;
        // Versuche len Bytes zu lesen oder bis
        // keine Bytes ohne Blockung gelesen werden können.
          try {
          for (; (i < len) && (in.available() > 0); i++) {
          c = read();
          if (c == -1) {
              break;
          }
          if (b != null) {
              b[off + i] = (byte)c;
          }
        }
      } catch (IOException ee) {
      }
      return i;
    }

    /*
     * Falls in buf nichts mehr zu entschlüsseln ist,
     * lies die nächsten vier Bytes.
     * Speichere jede Gruppe von 6 Bits in einem buf Element.
     */
    private int readCode() throws IOException {
        // Falls alle Daten in buf gelesen wurden,
        // (bufPos == 5) lies weitere 4 Bytes.
        if (bufPos == 5) {
            int b1 = in.read();
            int b2 = in.read();
            int b3 = in.read();
            int b4 = in.read();

            // EOF Test
            if ((b1 | b2 | b3 | b4) < 0) {
                throw new EOFException();
            }
            // 6 Bits an ein Element in
            // buf zuweisen
            int pack = (b1 << 24) | (b2 << 16) | (b3 << 8) | b4;
            buf[0] = (pack >>> 24) & 0x3F;
            buf[1] = (pack >>> 18) & 0x3F;
            buf[2] = (pack >>> 12) & 0x3F;
            buf[3] = (pack >>>  6) & 0x3F;
            buf[4] = (pack >>>  0) & 0x3F;
            bufPos = 0;
        }
        return buf[bufPos++];
    }
}
```

Falls Sie einen eigenen Eingabestrom schreiben müssen Sie Methoden definieren, mit denen die Daten aus dem Strom gelesen (oder im Falle von Output, geschrieben) werden können.

Daher müssen wir die Lesemethoden überschreiben:

`public int read()`
Diese Methode liest pro Aufruf ein Zeichen und entschlüsselt die 6 Bit Codes.

Die Methode

`public int read(byte b[], int off, int len)`
liest bis zu `len` Bytes und stellt diese in ein Array `b`. Dies geschieht mittels Aufrufen der `read` Methode, welche jeweils ein Zeichen pro Aufruf liest oder das Ende der Datei erreicht.

Die Methode `readCode`, welche der Methode `writeCode` im `CompressionOutputStream` entspricht, liest Daten aus dem Strom, entpackt jede Gruppe von vier Bytes in fünf 6-Bit Codes, welche dann von `read` entschlüsselt werden.

Schritt 3:
kreieren einer Subklasse von `java.net.Socket`

Um diese Unterklasse zu implementieren müssen wir die entsprechenden Methoden und den Konstruktor überschreiben, anpassen: `getInputStream`, `getOutputStream` und die `close` Methoden.

Nachdem wir die CompressionInputStream und CompressionOutputStream Klassen implementiert haben, können wir die Sockets, welche mit diesen Strömen kommunizieren, implementieren. Unsere Klasse muss also `java.net.Socket` erweitern.

```
package kompression;

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

class CompressionSocket extends Socket {

    /* InputStream  */
    private InputStream in;
    /* OutputStream  */
    private OutputStream out;
    /*
     * Standard Konstruktor für CompressionSocket
     */
    public CompressionSocket() { super(); }

    /*
     * Konstruktor für CompressionSocket
     */
    public CompressionSocket(String host, int port) throws IOException {
        super(host, port);
    }
    /*
     * Rückgabe eines CompressionInputStream Stromes
     */
    public InputStream getInputStream() throws IOException {
        if (in == null) {
```

```
            in = new CompressionInputStream(super.getInputStream());
        }
        return in;
    }
    /*
     * Rückgabe eines CompressionOutputStream Stromes
     */
    public OutputStream getOutputStream() throws IOException {
        if (out == null) {
            out = new CompressionOutputStream(super.getOutputStream());
        }
        return out;
    }
    /*
     * Flush (leeren) des CompressionOutputStream vor
     * dem Schliessen des Socket.
     */
    public synchronized void close() throws IOException {
        OutputStream o = getOutputStream();
        o.flush();
      super.close();
    }
}
```

Da unser neuer Socket mittels Datenkomprimierung kommuniziert, müssen wir die Methoden der Klasse Socket überschreiben und die Eingabe- und Ausgabe- Ströme direkt manipulieren sowie einen passenden Konstruktor definieren.

`public CompressionSocket()`
`public CompressionSocket(String host, int port)`
Der Konstruktor ruft den Konstruktor der übergeordneten Klasse `java.net.Socket` auf, sonst passiert nichts.

`public InputStream getInputStream()`
kreiert einen CompressionInputStream für den Socket, falls noch keiner existiert, und liefert einen eine Referenz auf den Strom.

`public OutputStream getOutputStream() throws IOException {`
liefert einen OutputStream, falls nötig. Dieser OutputStream ist jener, der vom Socket verwendet wird: `super.getOutputStream()`.

`public synchronized void close()`
Die `close`() Methode leert den Buffer `flush`() damit alle Daten gesendet werden, bevor der Socket geschlossen wird.

Schritt 4:
kreieren einer `java.net.ServerSocket` Unterklasse

Auch hier müssen wir die entsprechenden Methoden und Konstruktoren überschreiben. Die Unterklasse der ServerSocket Klasse nennen wir `CompressionServerSocket`.

Hier zur Übersicht der Quellcode. Die Beschreibung finden Sie nach dem Programm.

```
package Komprimierung;


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

```
class CompressionServerSocket extends ServerSocket {

    public CompressionServerSocket(int port) throws IOException {
        super(port);
    }

    public Socket accept() throws IOException {
        Socket s = new CompressionSocket();
        implAccept(s);
        return s;
    }
}
```

Wie im Falls der CompressionSocket müssen wir den Konstruktor und einige Methoden, hier die accept() Methode überschreiben.

`public CompressionServerSocket(int port)`
Den Konstruktor implementieren einfach indem wir den Konstruktor der übergeordneten Klasse aufrufen, also einen Server Socket kreieren.

`public Socket accept()`
Die einzige `ServerSocket` Methode, die wir überschreiben müssen, ist die Methode `accept()`. Sie wird einfach so überschrieben, dass sie einen `CompressionSocket` statt einen normalen `Socket` kreiert.

Die recht einfache Implementation eines Komprimierungssockets ist nur deswegen möglich, weil das Protokoll oberhalb von TCP (und erst recht oberhalb IP) implementiert wird. `java.net.Socket` und `java.net.ServerSocket` verwenden TCP als Standardprotokoll.

Nach diesem Einschub über Sockets und Socketmodifikationen kommen wir zurück zu unserer eigentlichen Aufgabe:

1. Entscheidung: welchen Socket Typus soll hergestellt werden? : *siehe oben*
2. Schreiben einer Client Socket Factory, welche `RMIClientSocketFactory` implementiert
3. Implementieren der `RMIClientSocketFactory` `createSocket` Methode
4. Schreiben einer Server Socket Factory, welche `RMIServerFactory` implementiert
5. Implementieren der `RMIServerSocketFactory` `createServerSocket` Methode

Schritt 2:
Schreiben einer Client Socket Factory, welche `RMIClientSocketFactory` implementiert

Wir beginnen die Implementation einer Client RMI Socket Factory, indem wir das Interface `RMIClientSocketFactory` implementieren. Die eigentliche Socket Factory nennen wir `CompressionClientSocketFactory`.

```
package Komprimierung;

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;
    }

}
```

Diese Klasse implementiert als einzige Methode:
`public Socket createSocket(String host, int port)`

Schritt 3:
Implementieren der `RMIClientSocketFactory` `createSocket` Methode

`public Socket createSocket(String host, int port)`
Die Methode kreiert einen neuen `CompressionSocket` und liefert diesen zurück.

Schritt 4:
Schreiben einer Server Socket Factory, welche `RMIServerFactory` implementiert

Wir nennen unsere Factory CompressionServerSocketFactory, eine Implementation des Interfaces RMISocketFactory.

Diese Klasse lässt sich analog zu oben sehr einfach implementieren:

```
package Komprimierung;

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;
    }
}
```

Schritt 5:
Implementieren der `RMIServerSocketFactory` `createServerSocket` Methode

Die Implementation der Methode `createServerSocket` in der RMI Socket Factory ist fast identisch mit der Implementation der `createSocket` Methode, ausser dass die Methode `createServerSocket` einen Socket vom Typus `CompressionServerSocket` liefern muss.

Nachdem wir wissen, wie man eine RMI Socket Factory kreieren muss, die einen bestimmten Socket Typ kreiert, wollen wir gleich die Erweiterung auf mehrere Sockettypen ansehen. Anschliessend werden wir ein `HelloWorld` Beispiel mit unterschiedlichen RMI Socket Factories ausprogrammieren.

2.4.1.2. Kreieren einer RMI Socket Factory, welche mehr als einen Socket Typus liefert

Un eine Socket Factory zu kreieren, welche mehrere Typen von Sockets kreiert, geht man analog vor wie im obigen Beispiel:

1. Entscheidung: welchen Socket Typus soll hergestellt werden?
2. Schreiben einer Client Socket Factory, welche `RMIClientSocketFactory` implementiert
3. Implementieren der `RMIClientSocketFactory` `createSocket` Methode
4. Schreiben einer Server Socket Factory, welche `RMIServerFactory` implementiert
5. Implementieren der `RMIServerSocketFactory` `createServerSocket` Methode

Einzig die "Wrapper" Klasse, die wir oben definiert haben, muss fähig sein, unterschiedliche Socket Typen zu liefern.

Die Socket Factory für dieses Beispiel werden wir MultiClientSocketFactory nennen. Die Socket Factories haben jeweils einen Konstruktor. Dieser muss in der Lage sein, unterschiedliche Sockets zu liefern. Wir verwenden zwei Socket Typen: "compression" und "xor" neben den Standard Socket Typen.

Schritt 1:
Entscheidung: welchen Socket Typus soll hergestellt werden?

Wie oben erwähnt werden wir drei Socket Typen verwenden: `XorSocket`, `CompressionSocket` und `java.net.Socket`.

Die Implementation der `XorSocket` geschieht analog zu den `CompressionSocket`: wir definieren Eingabe und Ausgabe Ströme und damit dann die Sockets.

```
package MultiSocket;

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

class XorSocket extends Socket {

    /*
     * Pattern zum verschlüsseln und entschlüsseln:
     * alle Daten, die von / an den Socket gesandt werden.
     */
    private byte pattern;

    /* InputStream, der vom Socket verwendet wird. */
    private InputStream in = null;

    /* OutputStream, der vom Socket verwendet wird. */
    private OutputStream out = null;

    /*
     * Konstruktor für die Klasse XorSocket.
     */
    public XorSocket(byte pattern)
        throws IOException
    {
```

```
        super();
        this.pattern = pattern;
    }

    /*
     * Konstruktor für die Klasse XorSocket.
     */
    public XorSocket(String host, int port, byte pattern)
        throws IOException
    {
        super(host, port);
        this.pattern = pattern;
    }

    /*
     * Liefert einen Stream vom Typus XorInputStream.
     */
    public InputStream getInputStream() throws IOException {
        if (in == null) {
            in = new XorInputStream(super.getInputStream(), pattern);
        }
        return in;
    }

    /*
     *Liefert einen Stream vom Typus XorOutputStream.
     */
    public OutputStream getOutputStream() throws IOException {
        if (out == null) {
            out = new XorOutputStream(super.getOutputStream(), pattern);
        }
        return out;
    }
}
```

Und daraus können wir die Server Sockets bauen:

```
package MultiSocket;

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

class XorServerSocket extends ServerSocket {

    /*
     * Pattern mit dem entschlüsselt und verschlüsselt wird:
     * jedes Byte, welches gelesen oder geschrieben wird.
     */
    private byte pattern;

    /*
     * Konstruktor für die Klasse XorServerSocket.
     */
    public XorServerSocket(int port, byte pattern) throws IOException {
        super(port);
        this.pattern = pattern;
    }

    /*
     * Kreieren eines Sockets vom Typus XorSocket und anschliessend Aufruf
     * von implAccept und warten auf Client Anfragen.
```

```
     */
    public Socket accept() throws IOException {
        Socket s = new XorSocket(pattern);
        implAccept(s);
        return s;
    }
}
```

Damit ist der erste Schritt abgeschlossen.

Den Quellcode von Schritte 2-5 finden Sie unten und daran anschlliessend eine Erläuterung der einzelnen Schritte.

```
package MultiSocket;

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

public class MultiClientSocketFactory
    implements RMIClientSocketFactory, Serializable{
    /* Standard RMISocketFactory    */
    private static RMISocketFactory defaultFactory =
      RMISocketFactory.getDefaultSocketFactory();

    private String protocol;
    private byte[] data;

    public MultiClientSocketFactory(String protocol, byte[] data) {
      this.protocol = protocol;
      this.data = data;
    }
    /*
     * überschreiben der createSocket Methode der
     * RMISocketFactory Klasse.
     * Falls man weder "Compression" oder "XOR" angibt,
     * wird eine Standard TCP Connection aufgebaut.
     */
    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("ungueltiges Argument für das XOR Protokoll");
          return new XorSocket(host, port, data[0]);
      }
      return defaultFactory.createSocket(host, port);
    }
}
```

Schritt 2:
Schreiben einer Client Socket Factory, welche `RMIClientSocketFactory` implementiert

Oben sehen Sie den Quellcode der MultiClientSocketFactory. Diese implementiert das RMISocketFactory Interface.

Schritt 3:
Implementieren der `RMIClientSocketFactory` `createSocket` Methode

```
public Socket createSocket(String host, int port)
      throws IOException
    {
      /*
      *     Abfrage des Protokolltyps:
      */

      /* Komprimierung */

      if (protocol.equals("compression")) {
          return new CompressionSocket(host, port);

      /* XOR           */

      } else if (protocol.equals("xor")) {
          if (data == null || data.length != 1)
      throw new IOException("ungueltiges Argument für das XOR Protokoll");
          return new XorSocket(host, port, data[0]);
      }

      /* sonst eben Standard Sockets */

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

Das Protokoll wird als Parameter an den Konstruktor der Klasse mitgegeben.

Schritt 4:
Schreiben einer Server Socket Factory, welche `RMIServerFactory` implementiert

```
package MultiSocket;

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")) {
            System.out.println("Es werden Compression Sockets verwendet");
          return new CompressionServerSocket(port);
```

```
      } else if (protocol.equals("xor")) {
            System.out.println("Es werden XOR Sockets verwendet");
          if (data == null || data.length != 1)
      throw new IOException("ungueltiges Argument für das XOR Protokoll");
          return new XorServerSocket(port, data[0]);
      }
   System.out.println("Es wwird das Standard Socket Protokoll verwendet!");
      return defaultFactory.createServerSocket(port);
    }
}
```

Schritt 5:
Implementieren der `RMIServerSocketFactory` `createServerSocket` *Methode*

```
public ServerSocket createServerSocket(int port) throws IOException
    {
      if (protocol.equals("compression")) {
      …
```

Das Überschreiben der createSocket Methode ist analog wie bei der RMIClientSocketFactory möglich.

2.4.1.3. Anwendung der selbstkonstruierten Socket Factory in einer Applikation

Um die RMI Socket Factory einsetzen zu können, müssen wir nun eine einfache Testapplikation schreiben. Dies geschieht in zwei Schritten:

1. schreiben eines Konstruktor für das remote Objekt, welcher den `UnicastRemoteObject` oder `Activatable` Konstruktor aufruft und die `RMIServerSocketFactory` und RMIClientSocketFactory als Parameter verwendet.
2. definieren der Zugriffsrechte in einer Policy Datei

In diesem Beispiel geben wir einmal mehr alle Privilegien. Falls Sie herausfinden wollen, welche Rechte benötigt werden, dann starten Sie die Applikation einfach ohne Policy Datei. Sie sehen dann welche Privilegien benötigt werden.

Schritt 1:
schreiben eines Konstruktor für das remote Objekt, welcher den `UnicastRemoteObject` *oder* `Activatable` *Konstruktor aufruft und die* `RMIServerSocketFactory` *und* `RMIClientSocketFactory` *als Parameter verwendet.*

Sie müssen dem RMI Laufzeitsystem mitteilen, welche Socket Factory verwendet werden soll. Falls Unser remote Objekt das `UnicastRemoteObject` erweitert, sieht dieser Aufruf folgendermassen aus:

```
protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,
        RMIServerSocketFactory ssf)
```

Schauen wir uns eine Implementation in einem Hello World Programm an (Programmskizze):

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

Der Aufruf des Konstruktors der Oberklasse impliziert den Aufruf von:

```
protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,
         RMIServerSocketFactory ssf)
```

Schritt 2:
definieren der Zugriffsrechte in einer Policy Datei

Unser Beispiel verwendet eine einfache Policy:

```
grant {
 // jeglicher Zugriff ist erlaubt
 permission java.security.AllPermission;
};
```

Und hier die vollständige Version des HelloWorld mit "XOR" oder "Compression" Protokoll.

```
package HelloWorld;

public interface Hallo extends java.rmi.Remote {
    String sagHallo() throws java.rmi.RemoteException;
}
```

Und so sieht der Client aus, wobei wir gleich zwei Protokollvarianten ausprobieren:

```
package HelloWorld;

import java.rmi.*;

public class HalloClient {
    private static String message = "";
    public static void main(String args[]) {
        System.out.println("HalloClient: Start");
      //kreieren und instanzieren des Security Manager
      if (System.getSecurityManager() == null)
          System.setSecurityManager(new RMISecurityManager());
        System.out.println("HalloClient: SecurityManager wurde
                            installiert");
        try {
            System.out.println("HalloClient: Lesen der Registry");
          // zum Testen : Abfrage der Registry
             for (int i=0; i< Naming.list("rmi://localhost/").length; i++)
                              System.out.println("HalloClient:
                                   "+Naming.list("rmi://localhost/")[i]);
            System.out.println("HalloClient: es wurden
                   "+Naming.list("rmi://localhost/").length + " Eintraege
                   in der Registry gefunden!");
            System.out.println("HalloClient: Lesen der HalloXORServer
                   Referenz aus der Registry");
            Hallo objXOR = (Hallo) Naming.lookup("/HalloXORServer");
            System.out.println("HalloClient: Aufruf der remote Methode
                   Hallo.sagHallo()");
            message = objXOR.sagHallo();
            System.out.println(message);

            System.out.println("HalloClient: Lesen der
                    HalloCompressionServer Referenz aus der Registry");
    Hallo objCompressed = (Hallo) Naming.lookup("/HalloCompressionServer");
```

```
    System.out.println("HalloClient: Aufruf der remote Methode
       Hallo.sagHallo()");
            message = objCompressed.sagHallo();
            System.out.println(message);

        } catch (Exception e) {
          System.out.println("HalloClient Exception: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }

}
```

Nun müssen wir noch den Server implementieren:

```
package HelloWorld;

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

public class HalloImpl extends UnicastRemoteObject implements Hallo {

    /*
     * Aufruf der Oberklasse mit dem Protokoll:
     * Client und Server Socket Factory.
     */
    public HalloImpl(String protocol, byte [] pattern) throws
                                          RemoteException {
      super(0, new MultiClientSocketFactory(protocol, pattern),
            new MultiServerSocketFactory(protocol, pattern));
    }

    /*
     * Remote Methode, die einen String "Hallo Du da!"
     * zurück liefert, falls sie aufgerufen wird.
     */
    public String sagHallo() throws RemoteException {
        System.out.println("HalloServer: Methode sagHallo wurde
                            aufgerufen!");
        return  "Hallo Du da!";
    }

    public static void main(String args[]) {
        System.out.println("HalloServer: Start");
      //Kreieren und installieren des Security Managers
      if (System.getSecurityManager() == null)
          System.setSecurityManager(new RMISecurityManager());
        System.out.println("HalloServer: SecurityManager wurde
                           installiert");
        byte [] aPattern = { (byte)1011 };
      try {
            // XOR Hallo Server
          HalloImpl objXOR = new HalloImpl("xor", aPattern);

            // Compressed Hallo Server
          HalloImpl objCompression = new HalloImpl("compression", null);

          Naming.rebind("/HalloXORServer", objXOR);
          System.out.println("HalloXORServer: in die Registry gebunden");
```

```
            Naming.rebind("/HalloCompressionServer", objXOR);
            System.out.println("HalloCompressionServer: in die Registry
                                gebunden");
        } catch (Exception e) {
            System.out.println("HalloImpl err: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Wir können also mehrere Server jeweils mit unterschiedlichen Socket Factories aufbauen. Server und Client können Sie mittels .bat Dateien starten. Testen Sie die Applikation und verschieben Sie anschliessend den Client in ein anderes Verzeichnis oder einen andern und testen Sie erneut, mit geänderten .bat Dateien.

Wenden wir uns nun dem zweiten Thema aus dem Themenkatalog "Spezialthemen RMI" zu:

1. Socket Factory : *siehe oben*
   dies zeigt, wie Sie die Standardsockets von RMI durch eigene ersetzen können und damit auch gleich noch die Kommunikation sicherer / verschlüsselt abwickeln können. Als Beispiel besprechen wir, wie eine Verbindung mit Hilfe von SSL realisiert werden könnte.
2. Activation
   diese Thema ist besonders interessant, da Sie damit Objekte "schlafend" setzen können und zu einem späteren Zeitpunkt wieder aufwecken können.
3. Dynamic Code Downloading
   in diesem Abschnitt besprechen wir die codebase Property etwas genauer, da wir sie oben umgangen haben.
4. Factory Pattern
   ist eine kurze Besprechung über den Einsatz des Entwurfsmusters "Factory" im Rahmen von RMI Applikationen
5. RMI over IIOP
   diese relativ neue Art der RMI Kommunikation ist in einem heterogenen Umfeld wichtig und gestattet die Einbindung von RMI in ein CORBA Umfeld.

2.4.2. Activation: Remote Objekt Aktivierung

Vor Java 2 konnte auf ein UnicastRemoteObjekt zugegriffen werden, falls
1. das Programm eine Instanz eines remote Objekts kreierte und
2. die ganze Zeit am Laufen war.

Seit der Einführung der Aktivierungsklassen `java.rmi.activation.Activatable` und dem RMI Daemon `rmid` kann man Programme schreiben, welche Informationen über remote Objekte enthalten kann, die auf Anweisung, "on demand", kreiert und ausgeführt werden können, statt dauernd aktiv sein zu müssen.

Der RMI Daemon rmid stellt eine Java Virtual Machine zur Verfügung, welche weitere JVM Instanzen erzeugen kann.Wie das Ganze geschieht wollen wir im Folgenden kennen lernen. Dabei gehen wir schrittweise vor.

2.4.2.1. Sich mit Aktivierung vertraut machen

Sie sollten sich bereits mit RMI etwas auskennen, bevor Sie sich mit remote Aktivierung beschäftigen. Um sich mit der Aktivierung vertraut zu machen, besprechen wir:

1. Kreieren eines aktivierbaren Objekts
   Dieser Teil der Einführung befasst sich mit dem Prozess des Kreierens einer neuen Klasse, welche `java.rmi.activation.Activatable` erweitert.

2. aktivierbar machen eines UnicastRemoteObject
   In diesem Teil werden wir sehen, wie eine Implementation eines remote Objekts, welches `UnicastRemoteObject` und `java.rmi.activation.Activatable` erweitert.

3. Aktivierung eines Objekts, welches java.rmi.activation.Activatable nicht erweitert
   Dabei geht es um den Einsatz der statischen Methode `Activatable.exportObject` im Konstruktor für eine Klasse, welche Activatable nicht erweitert. Zudem lernen wir, wie ein remote Interface kreiert werden kann.

4. Vertiefte Kenntnisse über Aktivierung
   In diesem Abschnitt werden wir, nachdem wir die obigen Themen bereits behandelt haben, uns mit der Frage beschäftigen, wie mit Hilfe eines `MarshalledObject` Daten persistent gespeichert werden können.

2.4.2.2. Kreieren eines aktivierbaren Objekts

Dieser Abschnitt zeigt Ihnen Schritt für Schritt, wie ein aktivierbares Objekt kreiert werden kann, durch Erweiterung von `java.rmi.activation.Activatable`. Vor Java 2 konnte eine Instanz der `UnicastRemoteObject` Klasse von einem Server Programm eingesetzt werden, welches

1. eine Instanz eines remote Objekts kreierte
2. die ganze Zeit am Laufen gehalten wurde.

Nun kann mit Hilfe der Klasse `java.rmi.activation.Activatable` und des RMI Daemon, `rmid`, Programme geschrieben werden, welche remote Objekte kreieren können, die "auf Anfrage" aktiviert werden können. Der RMI Daemon `rmid` stellt eine Java Virtual Machine zur Verfügung.

Das weitere Vorgehen:

1. Schrittweises kreieren der Implementationsklasse
2. Schrittweises kreieren der Setup Klassen
3. Schritte zum Übersetzen und Ausführen der Programmbeispiele

Für dieses Beispiel benötigen wur folgende Dateien:

1. `Client.java` :
   die Klasse, welche eine Methode des aktivierbaren Objekts aufruft

2. `MeinRemoteInterface.java`
   ein Interface, welches `java.rmi.Remote` erweitert und von

3. `ActivatableImplementation.java`
   implementiert wird. Diese Klasse wird aktiviert werden.

4. `Setup.java`,
   die Klasse, welche Informationen über die aktivierbare Klasse in die RMI Registry und beim Daemon einträgt.

Der Client unterscheidet sich nicht von einem Client für ein Standard RMI System: die Aktivierung ist eine strikt serverseitige Angelegenheit.

2.4.2.2.1. Kreieren der Implementationsklasse

Die Erstellung der Implementationsklasse geschieht in mehreren, vier Schritten:

1. importieren der benötigten Klassen
2. erweitern der `java.rmi.activation.Activatable` Klasse
3. deklarieren eines Konstruktors mit zwei Argumenten
4. implementieren der Methoden des remote Interfaces

Schritt 1:
importieren der benötigten Klassen

```
import java.rmi.*;
import java.rmi.activation.*;
```

Schritt 2:

erweitern der `java.rmi.activation.Activatable` *Klasse*

```
public class ActivatableImplementation extends Activatable
     implements Beispiel1.MeinRemoteInterface {
```

Schritt 3:
deklarieren eines Konstruktors mit zwei Argumenten

```
public ActivatableImplementation(ActivationID id, MarshalledObject data)
      throws RemoteException {
      // Registrieren des Objekts, welches das Aktivierungssystem
      // an einem anonymen Port anbietet (expportiert)
      //
  super(id, 0);
  System.out.println("ActivatableImplementation: Aufruf des Konstruktors");
    }
```

Schritt 4:
implementieren der Methoden des remote Interfaces

```
// Implementation der Methode, welche in MeinRemoteInterface
// deklariert wurde
//
    public Object rufMichRemoteAuf() throws RemoteException {

      return "ActivatableImplementation: Der Aufruf war erfolgreich!";
    }
```

2.4.2.2.2. Kreieren der "setup" Klasse

Die Setup Klasse soll alle nötigen Informationen kreieren, welche von der aktivierbaren Klasse benötigt werden, ohne dass eine Instanz eines remote Objekts kreiert wird.

Die Setup Klasse liefert die Informationen über aktivierbare Klassen an rmid, registriert eine Referenz (eine Instanz der Stub Klasse zur aktivierbaren Klasse) und registriert einen Namen in der RMI Registry. Nach diesen Arbeiten endet die Setup Klasse.

In diesem Programm durchlaufen wir sieben Schritte bis zur Fertigstellung der Klasse(n):
1. importieren aller relevanten Klassen
2. installieren eines Security Managers
3. kreieren einer Instanz der ActivationGroup
4. kreieren einer Instanz der ActivationDesc Klasse
5. deklarieren einer Instanz des remote Interfaces und registrieren bei `rmid`
6. binden des Stubs an einen Namen in der RMI Registry
7. verlassen der Setup Applikation

Schritt 1:
importieren aller relevanten Klassen

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
```

Schritt 2:
installieren eines Security Managers

```
System.out.println("Setup: Setzen des Security Managers");
System.setSecurityManager(new RMISecurityManager());
```

In diesem Beispiel verwenden wir eine einfache Policy Datei:

```
grant {
      // Allow everything for now
      permission java.security.AllPermission;
};
```

Diese Policy ist in einem produktiven Umfeld nicht gerade sinnvoll! Wie eine Policy sinnvollerweise aufgesetzt werden kann, können Sie in den Unterlagen zur Security nachlesen. Diese Policies wurden ab Java 2 nötig; dort wurde das Security Modell wesentlich erweitert. Sie können die Policy Datei mit dem PolicyTool erstellen und verwalten, sofern Sie etwas üben!

```
// Ab Java 2 muss eine Security Policy definiert werden
// für die ActivationGroup VM
// Das erste Argument von "put" ist der Schlüssel (aus der
// Hashtabelle),
// das zweite Argument ist dessen Wert, der Dateiname
//
Properties props = new Properties();
System.out.println("Setup: Setzen der Property 'java.security.policy'");
props.put("java.security.policy","java.policy");
```

Das Setzen des Policy Verzeichnisses, und der Policy Datei, muss mit absoluten Pfaden geschehen, damit jedes Programm darauf zugreifen kann. Eine andere Variante finden Sie im Bootstrap Beispiel: dort wird die Lokation der Policy Datei mittels einer URL angegeben.

Schritt 3:
kreieren einer Instanz der ActivationGroup

```
System.out.println("Setup: ActivationGroup");
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc beispielGruppe = new ActivationGroupDesc(props, ace);

// Registrieren der ActivationGroupDesc
// Dies liefert eine ID
//
ActivationGroupID agi =
      ActivationGroup.getSystem().registerGroup(beispielGruppe);

// kreieren der Gruppe
//
ActivationGroup.createGroup(agi, beispielGruppe, 0);
```

Schritt 4:
kreieren einer Instanz der `ActivationDesc` *Klasse*

Der `ActivationDesc`, der Activation Deskiptor liefert alle Informationen, welche rmid benötigt, um eine Instanz der Implementationsklasse zu kreieren.

Sie müssen eventuell einige der Pfad und URL Abgaben Ihrer Umgebung anpassen.

```
// die Zeichenkette "location" spezifiziert eine URL
// von der die Klassendefinition geladen werden kann
// falls das Objekt aktiviert werden muss.
// Am Schluss der URL
// steht ein /.
//
String location =
"file:/D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapitel16_RMIIn
vocation/Programme/Aktivierung/";

System.out.println("Setup: Lokation (URL) - "+location);

// kreieren des Rests der Parameter
// für den ActivationDesc Konstruktor
//
MarshalledObject data = null;

// Das zweite Argument des ActivationDesc Konstruktors wird benötigt,
// um diese Klasse eindeutig zu identifizieren;
// die Lokation bezieht sich relativ auf die
// URL-formatierte Zeichenkette "location".
//
ActivationDesc desc = new
ActivationDesc("Beispiel1.ActivatableImplementation", location, data);
```

Schritt 5:
deklarieren einer Instanz des remote Interfaces und registrieren bei `rmid`

```
// Registrieren bei rmid
//
MeinRemoteInterface mri = (MeinRemoteInterface)Activatable.register(desc);
System.out.println("Setup: der Stub fuer die ActivatableImplementation
wurde gefunden" );
```

Schritt 6:
binden des Stubs an einen Namen in der RMI Registry

```
//Binden des Stub an einen Namen in der Registry an Port 1099
//
Naming.rebind("ActivatableImplementation", mri);
System.out.println("Setup: die ActivatableImplementation wurde exportiert"
);
```

Schritt 7:
verlassen der Setup Applikation

```
System.out.println("Setup: Ende");
System.exit(0);
```

Damit steht der Quellcode der Klasse Setup zur Verfügung. Was verbleibt ist die Übersetzung und die Erstellung aller weiteren Klassen.

2.4.2.2.3. Übersetzen und ausführen des Codes

Wir gehen auch hier in (sechs) Schritten vor:

1. übersetzen des remote Interface, Implementation, Client und Setup Klassen
2. starten von `rmic` für die Implementationsklasse
3. starten der `rmiregistry`
4. Aktivierung des RMI Aktivierungsdaemons `rmid`
5. starten des Setup Programms
6. starten des Clients

Schritt 1:
übersetzen des remote Interface, Implementation, Client und Setup Klassen

Dafür steht Ihnen eine .bat Datei zur Verfügung.

```
@echo off
echo Uebersetzen aller Java Quelldateien
echo Bitte warten ...
javac -classpath . *.java
pause
```

Konkret werden folgende Dateien übersetzt:

```
ActivatableImplementation.java
Client.java
MeinRemoteInterface.java
Setup.java
```

Schritt 2:
starten von `rmic` *für die Implementationsklasse*

```
@echo off
echo Generieren von Stubs und Skeletons
echo Bitte warten ...
cd ..
rmic -classpath . Beispiel1.ActivatableImplementation
cd Beispiel1
pause
```

Wegen des Package Namens wechseln wir auf das übergeordnete Verzeichnis.

Schritt 3:
starten der `rmiregistry`

```
@echo off
echo RMIServer
echo Abbruch mit CTRL/C
set CLASSPATH=
cd ..
rmiregistry
cd Beispiel1
```

Beachten Sie, dass in diesem Fenster der CLASSPATH gelöscht werden muss, damit bei Anfragen die Klasseninformationen noch zur Verfügung stehen. Falls rmiregistry die Klassen im CLASSPATH findet, nimmt rmiregistry an, dass dies allgemein der Fall ist und "verliert" die gespeicherte Informationen, die java.rmi.server.codebase Information. Die Clients sind damit nicht mehr in der Lage die Class (Stubs) Dateien herunter zu laden.

Schritt 4:
Aktivierung des RMI Aktivierungsdaemons `rmid`

```
@echo off
echo RMI Daemon
echo Anhalten mit stop_rmid
set CLASSPATH=
cd ..
rmid -J-Djava.security.policy=rmid.policy
cd Beispiel1
pause
```

Die `rmid.policy` Datei lassen wir der Einfachheit halber auch sehr einfach: wir geben auch hier alle Privilegien (mit PolicyTool kreiert):

```
grant {
      // Allow everything for now
      permission java.security.AllPermission;
};
```

rmid benötigt eine Policy Datei, um feststellen zu können, ob die Activation Group eine JVM starten darf. Wie `rmid` funktioniert, können Sie im folgenden Einschub über `rmid` nachlesen.

Schritt 5:
starten des Setup Programms

Mit dem Setup Programm wird die Codebase Property gesetzt. Damit können die Stubs gefunden werden. Auf der Kommandozeile sollten vier Angaben stehen:

1. Der `java` Befehl
2. die Angabe der Property, mit der angegeben wird, wo die Policy Datei steht
3. die Angabe der Property, mit der angegeben wird, wo der Stub Code zu finden ist, wobei kein Leerzeichen im Namen vorkommen darf, von "-D" bis zum abschliessenden "/".
4. der vollqualifizierte Namen des Setup Programms

Beispiel:

```
1. java  -Djava.security.policy=java.policy
         -Djava.rmi.server.codebase=file://d:\\RMIKlassen/
          Aktivierungsbespiel1.Setup
2. @echo off
   echo Java Activation
   echo Setup
   Rem
   cd ..
   java  -Djava.security.policy=rmid.policy -
   Djava.rmi.server.codebase=file:/d:/UnterrichtsUnterlagen/ParalleleUndVer
   teilteSysteme/Kapitel16_RMInvocation/Programme/Activierung/Beispiel1/
   Beispiel1.Setup
   cd Beispiel1
   pause
```

Dateiangaben in den Properties sind URLs, wir könnten also auch einen HTTP Server einsetzen. Beachten Sie den Bindestrich in der Verzeichnisangabe oben: ohne den tritt ein Fehler auf.

Falls beispielsweise der abschliessende "/" fehlt, kann eine `java.lang.ClassNotFoundException` geworfen werden. Der selbe Fehler kann auftreten, falls der CLASSPATH der RMI Registry bekannt ist.

Schritt 6:
starten des Clients

Der Client wird mit dem Rechnernamen als Parameter gestartet.

```
@echo off
Rem
Rem RMI Aktivierung - Client
echo RMI Aktivierung - Client
Rem
cd ..
java -Djava.security.policy=java.policy Beispiel1.Client localhost
cd Beispiel1
pause
```

Ausgaben
Die Programme wurden mit Ausgaben erweitert, damit man besser erkennen kann, was in dieser verteilten Applikation abläuft. Sie können nicht einfach den Server mit dem Debugger laufen lassen, da der Server laufen muss bevor der Client gestartet wird.

Setup:

```
Java Activation
Setup
Setup: Start
Setup: Setzen des Security Managers
Setup: Setzen der Property 'java.security.policy'
Setup: ActivationGroup
Setup: Lokation (URL) -
file:/D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapite
l16_RMInvocation/Programme/Aktivierung/
Setup: der Stub fuer die ActivatableImplementation wurde gefunden
Setup: die ActivatableImplementation wurde exportiert
Setup: Ende
Taste drücken, um fortzusetzen . . .
```

Client:

```
RMI Aktivierung
Client
Client: Start
Client: setzen des SecurityManagers
Client: Lookup
Client: remote Referenz auf ein Objekt, welches Activatable implementiert.
Client: Aufruf der remote Methode auf dem Server
Client: Result des Aufrufs - ActivatableImplementation: Der Aufruf war
erfolgreich!
Client: Ende
Taste drücken, um fortzusetzen . . .
```

RMI Daemon

```
Anhalten mit stop_rmid
Fri Oct 13 21:56:20 GMT+02:00 2000:ExecGroup-
0:out:ActivatableImplementation: Aufruf des Konstruktors
```

2.4.2.2.4. rmid - Java RMI Activation System Daemon

`rmid` startet den Activation System Daemon, mit dessen Hilfe Objekte registriert und in einer Java Virtual Machine (JVM) aktiviert werden.

SYNOPSIS
`rmid [options]`

BESCHREIBUNG
Das rmid Tool startet den Daemon des Aktivierungssystem. Der Daemon muss gestartet werden, bevor aktivierbare Objekte im Aktivierungssystem registriert oder in einer JVM aktiviert werden können.

Der Daemon kann durh Ausführen des rmid Befehls gestartet werden, zusammen mit der Angabe einer Policy Datei:

```
rmid -J-Djava.security.policy=rmid.policy
```

Falls man die Sun Implementation des Tools verwendet muss eine Policy Datei engegeben werden, um Feststellen zu können, ob Informationen aus den `ActivationGroupDesc` benutzt werden dürfen, um eine JVM zu starten.

Der Wert von `sun.rmi.activation.execPolicy` bestimmt diese Policy.

Das Tool startet standardmässig den Activator an Port 1098 und bindet ein `ActivationSystem` an die interne Registry. Sie können jedoch alternative Ports angeben:

```
rmid -J-Djava.security.policy=rmid.policy -port 1099
```

OPTIONEN

`-C<Zeilen auf der Kommandozeile>`
spezifiziert eine Option, die als Parameter auf der Kommandozeile an jeden Child Prozess (Aktivierungsgruppe) von rmid weiter gegeben wird, sobald dieser Child Prozess kreiert wird.

Beispiel:
`rmid -C-Dsome.property=value`

Diese Parameter sind beispielsweise für das Testen einer Applikation sehr nützlich:
`rmid -C-Djava.rmi.server.logCalls=true`

Damit werden alle Cild JVMs geloggt!

`-J<someCommandLineOption>`
spezifiziert eine Option für den Java Interpreter von `rmid`.

Beispiel:
`rmid -J-Djava.security.policy=rmid.policy`

Legt die Datei fest, in der die Policies stehen.

`-J-Dsun.rmi.activation.execPolicy=<policy>`

spezifiziert die Policy, welche `rmid` verwendet, um Kommandos der Befehlszeile zu überprüfen. Dies funktioniert nur mit der Sun Implementation von `rmid`.
Falls die Angabe fehlt, wird `-J-Dsun.rmi.activation.execPolicy=default` verwendet.

Zusätzlich können Properties gesetzt werden, wie beim Java Interpreter:
`-D<property>=<value>`

Dabei unterscheidet man zwei Gruppen:
ExecPermission und ExecOptionPermission.

ExecPermissons:
Machen wir erst einmal ein Beispiel
```
permission com.sun.rmi.rmid.ExecPermission
"c:\\jdk1.3\\bin\\java";
```

Diese Permissions werden beim Ausführen eines bestimmten Kommandos beachtet.

Syntax
Die Pfadangabe zeigt auf die Datei, welche ausgeführt werden darf.
Erlaubt sind auch allgemeinere Pfadangaben, wie "*****", bei der die Ausführung aller Programme im entsprechenden Verzeichnis erlaubt wird;
bei der Angabe eines Pfades "**\-**" zeigt an, dass auch alle Unterverzeichnisse zugelassen sind.
schliesslich erlaubt die Angabe "**<<ALL FILES>>**" das Ausführen aller Programme.

Beispiel:
"*" bezeichnet alle Dateien des aktuellen Verzeichnisses
"-" bezeichnet alle Dateien des aktuellen und aller Unter-Verzeichnisse.

ExecOptionPermission
Ein Beispiel dafür wäre:
permission com.sun.rmi.rmid.ExecOptionPermission
"-Djava.security.debug=*";

Mit der ExecOptionPermisson werden `rmid` bestimmte Optionen bei der Aktivierung einer Aktivierungsgruppe verwendet.

Syntax
Wie oben werden Wildcards unterstützt: "*".

Policy Datei für `rmid`
Machen wir einfach ein kompexeres Beispiel:
```
grant {
       permission com.sun.rmi.rmid.ExecPermission
       "c:\\jdk1.3\\bin\\java";

       permission com.sun.rmi.rmid.ExecPermission
       "c:\\jdk1.3\\bin\\java_g";

       permission com.sun.rmi.rmid.ExecPermission
       "c:\\jdk1.3\\bin\\rmidcmds\\*";
```

```
        permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.policy=c:\\policies\\group.policy";

        permission com.sun.rmi.rmid.ExecOptionPermission
        "-Djava.security.debug=*";

        permission com.sun.rmi.rmid.ExecOptionPermission
        "-Dsun.rmi.*";
};
```

Die ersten zwei Permissons gestatten rmid die Java\bin Programme auszuführen.
Diese Angabe ist überflüssig, weil die Befehle in Java\bin in JAVA_HOME stehen.

Die dritte Permisson gestattet die Ausführung eines beliebigen Befehls im Verzeichnis `c:\jdk1.3\bin\rmidcmds`

Die vierte Permission definiert eine Policy Datei.

Die letzte Permission gestattet es irgend eine Property aus der sun.rmi Hierarchie zu verwenden.

Zum Starten von rmid mit einer Policy muss der Aufruf
`rmid -J-Djava.security.policy=rmid.policy`

ausgeführt werden.

Weitere Parameter:

`-log dir`
Spezifiziert das Verzeichnis, in welches der Daemon ein Log schreibt.

`-port port`
Spezifiziert den Port, welcher von `rmid` verwendet wird.

`-stop`
stoppt die Ausführung des Aktivierungs Daemons, gegebenenfalls an einem bestimmten Port.

2.4.2.3. Aktivierbarmachen eines Unicast Remote Objekts

In diesem Teil werden wir sehen, wie eine Implementation eines remote Objekts, welches `UnicastRemoteObject` und `java.rmi.activation.Activatable` erweitert.

Wir bauen also eine Klasse um von einer Klasse, die ein `UnicastRemoteObject` ist in ein aktivierbares Objekt, welches `java.rmi.activation.Activatable` erweitert.

Das Thema ist also eigentlich nur für Leute wichtig, die eine bestehende remote Applikation in eine aktivierbare Applikation umwandeln wollen.

In diesem Beispiel besprechen wir der Reihe nach folgende Themen:
1. Schrittweises modifizieren der Implementationsklasse
2. Schrittweises modifizieren der Serverklasse
3. Übersetzen und starten des Beispiels

Sie werden folgende Dateien benötigen:

- `Client2.java`
  Der Client ruft Methoden eines aktivierbaren Objekts auf.
  Änderungen an dieser Klasse im Vergleich zum vorhergehenden sind minimal bis null, da wir ja lediglich den Server modifizieren, aktivierbar machen möchten. Aktivierbarkeit ist eine rein serverseitige Angelegenheit.
- `MeinRemoteInterface.java`
  Das Interface erweitert java.rmi.Remote und wird implementiert durch:
- `MeinRemoteInterfaceImpl.java`
  Diese Klasse wird aktiviert werden
- Setup2.java
  Die Setup Klasse registriert Informationen über die aktivierbare Klasse bei der RMI Registry und dem RMI Daemon

2.4.2.3.1. Schrittweises modifizieren der Implementationsklasse

Unsere Beispielanwendung ist `MeinRemoteInterfaceImpl.java`. Die Migration einer bestehenden Implementationsklasse, welche `UnicastRemoteObject` erweitert, geschieht in folgende vier Schritten:

1. Importieren der relevanten Bibliotheken und Klassen in die Implementationsklasse
2. Modifikation der Klassendefinition, so dass die Klasse neu `java.rmi.activation.Activatable` erweitert.
3. Entfernen oder auskommentieren des alten Standard-Konstruktors.
4. Definieren eines Konstruktors der Implementationsklasse mit zwei Argumenten (`ActivationID id, MarshalledObject data`).

Schritt 1:
Importieren der relevanten Bibliotheken und Klassen in die Implementationsklasse

```
import java.rmi.*;
import java.rmi.activation.*;
```

Schritt 2:
Modifikation der Klassendefinition, so dass die Klasse neu `java.rmi.activation.Activatable` *erweitert.*

```
public class MeinRemoteInterfaceImpl extends Activatable
    implements MeinRemoteInterface {
```

Schritt 3:
Entfernen oder auskommentieren des alten Standard-Konstruktors.

```
/*

public MeinRemoteInterfaceImpl()
      throws RemoteException {
...

}

*/
```

Schritt 4:
Definieren eines Konstruktors der Implementationsklasse mit zwei Argumenten (`ActivationID id, MarshalledObject data`).

```
public MeinRemoteInterfaceImpl(ActivationID id, MarshalledObject data)
      throws RemoteException {

      // Registrieren des Objekts beim Aktivierungssystem
      // und export an einen anonymen Port.
      //
      super(id, 0);
    }
```

2.4.2.3.2. Schrittweises modifizieren der Serverklasse

Entgegen einer klassischen RMI Serverklasse, die dauernd aktiv im Server auf Kundenanfragen warten muss, muss die Setup Klasse lediglich alle Informationen über die aktivierbare definieren und registrieren, ohne dass eine Instanz eines remote Objekts kreiert werden muss. Unsere Klasse ist in der Datei `Setup2.java` beschrieben.

Die Setup Klasse liefert dem RMI Daemon die Informationen über die aktivierbare Klasse, registriert eine remote Referenz (eine Instanz der aktivierbaren Stub Klasse) und einen Bezeichner, einen Namen bei der RMI Registry. Dann ist der Setup abgeschlossen und die Klasse kann beendet werden.

Das Kreieren der **Setup** Klasse geschieht in sieben Schritten:

1. Importieren der relevanten Klassen und Bibliotheken
2. Installieren des Security Managers
3. Kreieren einer Instanz der Activation Group
4. Kreieren einer Instanz der Activation Description
5. Entfernen der Referenz auf die Konstruktion der Implementationsklasse und registrieren beim RMI Daemon
6. Binden des Stubs an einen Namen in der RMI Registry
7. Abschliessen der Setup Applikation

Schritt 1:
Importieren der relevanten Klassen und Bibliotheken.

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
```

Schritt 2:
Installieren des Security Managers

```
System.out.println("Setup: Setzen des Security Managers");
System.setSecurityManager(new RMISecurityManager());
```

Auch in diesem Beispiel verwenden wir eine sehr einfache Sicherheits-Policy: wir gestatten den freien Zugriff: diese stehen in `java.policy`.

Die Lokation der Policy Datei muss dem System als Property mitgeteilt werden, oder repliziert werden.

```
// Ab Java 2 muss eine Security Policy definiert werden
// für die ActivationGroup VM
// Das erste Argument von "put" ist der Schlüssel (aus der
// Hashtabelle),
// das zweite Argument ist dessen Wert, der Dateiname
//
Properties props = new Properties();
System.out.println("Setup: Setzen der Property 'java.security.policy'");
props.put("java.security.policy", "java.policy");
```

Schritt 3:
Kreieren einer Instanz der Activation Group

Die Aufgabe der Activation Group Beschreibung ist es, dem Activation Daemon alle Informationen zu liefern, die er benötigt, um die passende JVM zu kontaktieren oder dem aktivierbaren Objekt eine neue JVM zur Verfügung zu stellen.

```
System.out.println("Setup: ActivationGroup");
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc beispielGruppe = new ActivationGroupDesc(props, ace);

// Registrieren der ActivationGroupDesc
// Dies liefert eine ID
//
ActivationGroupID agi =
ActivationGroup.getSystem().registerGroup(beispielGruppe);

// kreieren der Gruppe
//
ActivationGroup.createGroup(agi, beispielGruppe, 0);
```

Schritt 4:
Kreieren einer Instanz der Activation Description

Die Aufgabe der Aktivierungsbeschreibung ist es, alle Informationen, welche der RMI Daemon benötigt, um eine neue Instanz der Implementationsklasse zu erzeugen, zur Verfügung zu stellen.

Die Lokation zeigt dabei auf die Class Dateien, da wir diese in diesem Beispiel nicht dynamisch herunterladen. Aber im Prinzip könnten wir auch mit URLs aebeiten.

```
// die Zeichenkette "location" spezifiziert eine URL
// von der die Klassendefinition geladen werden kann
// falls das Objekt aktiviert werden muss.
// Am Schluss der URL
// steht ein "/".
//
String location =
"file:/D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapitel16_RMIIn
vocation/Programme/Aktivierung/";
System.out.println("Setup: Lokation (URL) - "+location);

// kreieren des Rests der Parameter
// für den ActivationDesc Konstruktor
```

```
//
MarshalledObject data = null;

// Das zweite Argument des ActivationDesc Konstruktors wird benötigt,
// um diese Klasse eindeutig zu identifizieren;
// die Lokation bezieht sich relativ auf die
// URL-formatierte Zeichenkette "location".
//
ActivationDesc desc = new
            ActivationDesc("Beispiel1.ActivatableImplementation", location,
data);
```

Schritt 5:
Entfernen der Referenz auf die Konstruktion der Implementationsklasse und registrieren beim RMI Daemon

```
// MeinRemoteInterfaceImpl mri = MeinRemoteInterfaceImpl();
// wird ersetzt durch:
//
// Registrieren bei rmid
//
MeinRemoteInterface mri = (MeinRemoteInterface)Activatable.register(desc);
System.out.println("Setup: der Stub fuer die ActivatableImplementation
wurde gefunden" );
```

Schritt 6:
Binden des Stubs an einen Namen in der RMI Registry

Beim Stub handelt es sich um jenen, der von der `Activatable.register(desc)` zurück geliefert wird und wie oben ersichtlich gecastet wird.

Das Binden geschieht mittels folgender Programmzeilen:

```
// Binden des Stub an einen Namen in der Registry an Port 1099
//
Naming.rebind("ActivatableImplementation", mri);
System.out.println("Setup: die ActivatableImplementation wurde exportiert"
);
```

Schritt 7:
Abschliessen der Setup Applikation

```
System.out.println("Setup: Ende");
System.exit(0);
```

2.4.2.3.3. Übersetzen und starten des Beispiels

Das Übersetzen und schrittweise Starten der Beispielanwendung geschieht in sechs Schritten:

1. Übersetzen der Java Quellen für das remote Interface, die Implementationsklasse, den Client und die Setup Klasse.
2. Starten von `rmic` zur Generierung der Stubs
3. Starten der RMI Registry
4. Starten des RMI Aktivierungsdaemons
5. Starten des Setup Programms
6. Starten des Client

Schritt 1:
Übersetzen der Java Quellen für das remote Interface, die Implementationsklasse, den Client und die Setup Klasse.

```
@echo off
Rem
Rem Alles uebersetzen
echo RMI Aktivierung
echo alle Java Dateien uebersetzen
Rem
javac -classpath . *.java
pause
```

Im Einzelnen werden folgende Java Klassen übersetzt:

```
ActivatableImplementation.java
Client2.java
MeinRemoteInterface.java
MeinRemoteInterfaceImpl.java
Setup2.java
```

Schritt 2:
Starten von `rmic` *zur Generierung der Stubs.*

```
@echo off
Rem
echo rmic - generieren von Stubs und Skeletons
echo Bitte warten ...
cd ..
rmic -classpath . Beispiel2.ActivatableImplementation
rmic -classpath . Beispiel2.MeinRemoteInterfaceImpl
cd Beispiel2
pause
```

Schritt 3:
Starten der RMI Registry.

```
@echo off
echo RMIServer
echo Abbruch mit CTRL/C
set CLASSPATH=
cd ..
rmiregistry
cd Beispiel2
pause
```

Wie aus dem obigen .bat Skript ersichtlich, muss der CLASSPATH in jenem Fenster, in dem die RMI Registry gestartet wird, gelöscht werden, oder darf auf keinen Fall auf die Klassen zeigen, welche zum Client herunter geladen werden müssen, inklusive der Stubs für die remote Objekte.

Falls Sie dies nicht beachten, kann die RMI Registry die Stub Klassen im CLASSPATH finden. Falls dies der Fall ist, *ignoriert* die Registry die `java.rmi.server.codebase` Property. Damit kann aber auch der Client die Stubs nicht finden und sie somit nicht herunter laden können.

Schritt 4:
Starten des RMI Aktivierungsdaemons

```
@echo off
echo RMI Daemon
echo Anhalten mit rmid -stop oder rmid_stop.bat
set CLASSPATH=
cd ..
rmid -J-Djava.security.policy=rmid.policy
cd Beispiel2
pause
```

Wir haben die Möglichkeit eine Policy Datei für den RMI Daemon zu definieren und diese auf der Kommandozeile einzugeben.

Ohne Policy kann das Aktivierungssystem keine JVM starten, es kann kein Objekt aktiviert werden. In der Regel dauert der Start des Daemons eine Weile: Sie sollten nach dem Starten des Daemons nicht gleich den Client starten: das kann schief gehen. Sie müssen in diesem Fall den Client erneut starten, da ein Stack-Dump generiert wird.

Schritt 5:
Starten des Setup Programms

```
@echo off
echo Java Activation
echo Setup
Rem
cd ..
java  -Djava.security.policy=rmid.policy -
Djava.rmi.server.codebase=file:/d:/UnterrichtsUnterlagen/ParalleleUndVertei
lteSysteme/Kapitel16_RMInvocation/Programme/Activierung/Beispiel2
Beispiel2.Setup2
cd Beispiel2
pause
```

Wie aus dem obigen Beispiel ersichtlich besteht der Start des Setup Programms aus mehreren Teilen:

1. dem java Befehl, also dem Starten der JVM
2. der Angabe der Policy Datei: -Djava.security.policy=<rmid.policy>
3. der Angabe der Codebase, also des Verzeichnisses, in dem der Stub Code zu finden ist. Diese Angabe darf ab dem "-D" kein Leerzeichen enthalten. Sie sehen dies oben daran, dass das Verzeichnis "...Kapitel16 RMI..." in "...Kapitel16_RMI..." umbenannt wurde, werden musste!
   Die Angabe geschieht in Form einer URL. Wir könnten also auch einen HTTP oder FTP Server verwenden, ab dem die Klassen herunter geladen werden können. Falls man eine Datei mit "file:" angibt, muss man beachten, dass es Betriebsysteme gibt, die entweder einen "/", zwei "//" oder drei "///" verlangen.
   Oft wird auch ein abschliessender "/" verlangt.
4. der voll qualifizierten Angabe des Package Namens des Setup Programms. Im Beispiel oben: `Beispiel2.Setup2`

Falls die CODEBASE falsch eingegeben wird, meldet sich das Programm normalerweise mit einer `java.lang.ClassNotFoundException`. Daneben haben Sie, falls die Registry an einem

unbekannten Ort gestartet wird, vermutlich das Problem, dass die Klassen überhaupt nicht gefunden werden. Ausgaben sehen Sie weiter unten.

Schritt 6:
Starten des Client

```
@echo off
Rem
Rem RMI Aktivierung - Client
echo RMI Aktivierung
echo Client
Rem
cd ..
java -Djava.security.policy=java.policy Beispiel2.Client2 localhost
cd Beispiel2
pause
```

Wie aus dem obigen Beispiel ersichtlich besteht der Start des Client Programms aus mehreren Teilen:

1. dem java Befehl, also dem Starten der JVM
2. der Angabe der Policy Datei: -Djava.security.policy=<rmid.policy>
3. der voll qualifizierten Angabe des Package Namens des Setup Programms. Im Beispiel oben: `Beispiel2.Client2`
4. dem Rechnernamen, in unserem Beispiel `localhost`

Ausgaben
Die Programme wurden mit Ausgaben erweitert, damit man besser erkennen kann, was in dieser verteilten Applikation abläuft. Sie können nicht einfach den Server mit dem Debugger laufen lassen, da der Server laufen muss bevor der Client gestartet wird.

Setup:

```
Java Activation
Setup
Setup2: Start
Setup2: Setzen des Security Managers
Setup2: Setzen der Property 'java.security.policy'
Setup2: ActivationGroup
Setup2: Lokation (URL) -
file:/D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapit
el16_RMInvocation/Programme/Aktivierung/
Setup2: ActivationDesc
Setup2: Casting
Setup2: der Stub fuer die ActivatableImplementation wurde gefunden
Setup2: die ActivatableImplementation wurde exportiert
Setup2: Ende
Taste drücken, um fortzusetzen . . .
```

Client:

```
RMI Aktivierung
Client
Client: Start
Client: setzen des SecurityManagers
Client: Lookup
Client: remote Referenz auf ein Objekt, welches Activatable implementiert.
Client: Aufruf der remote Methode auf dem Server
```

```
Client: Result des Aufrufs - ActivatableImplementation: Der Aufruf war
erfolgreich!
Taste drücken, um fortzusetzen . . .
```

RMI Daemon

```
RMI Daemon
Anhalten mit stop_rmid
Sat Oct 14 13:22:51 GMT+02:00 2000:ExecGroup-
      0:out:ActivatableImplementation: Aufruf des Konstruktors
Sat Oct 14 13:26:17 GMT+02:00 2000:ExecGroup-
      1:out:ActivatableImplementation: Aufruf des Konstruktors
```

Pro Client Aufruf wird eine Zeile ausgegeben. Im Log werden weitere Daten erfasst.

2.4.2.4. Aktivierung eines Objekts, welches java.rmi.activation.Activatable *nicht* erweitert

In diesem Abschnitt wollen wir untersuchen und aufzeichnen, wie eine bestehende Klasse aktivierbar gemacht werden kann.

Auch in diesem Abschnitt gehen wir schrittweise vor, wir wollen auch hier eine Art Checkliste für das Vorgehen aufstellen und gleich an einem Beispiel zeigen.

1. Schrittweises kreieren eines remote Interfaces

2. Schrittweises modifizieren der Implementationsklasse

3. Schrittweises entwickeln der Setup Klasse

4. Übersetzen und starten der Beispielprogramme

Das Beispiel besteht aus folgenden Java Dateien:

1. Client3.java
   Diese Klasse wird die remote Methode aufrufen

2. MeineKlasse.java
   Diese Klasse wird aktivierbar gemacht

3. NochEinRemoteInterface.java
   Diese Schnittstelle erweitert java.rmi.Remote und wird implementiert von:

4. Setup3.java
   Diese Klasse registriert die Informationen über die aktivierbare Klasse bei der RMI Registry und beim RMI Aktivierungsdaemon

Den Client brauchen wir nicht im Detail zu besprechen, da er sich kaum von den andern Clients unterscheidet; das sollte auch sein, da der Client nichts zur Aktivierung beiträgt: Aktivierung ist eine serverseitige Angelegenheit.

2.4.2.4.1. Schrittweises kreieren eines remote Interfaces

Im Interface werden alle Methoden, die man remote aufrufen möchte, definiert. In unserem Beispiel nennen wir diese Schnittstelle `Beispiel3.NochEinRemoteInterface.`

Das Kreieren des remote Interfaces **NochEinRemoteInterface.** geschieht in drei Schritten:

1. Importieren der relevanten Klassen und Bibliotheken
2. Erweiterung von java.rmi.Remote
3. Deklarieren aller Methoden, die remote aufgeruft werden können

Schritt 1:
Importieren der relevanten Klassen und Bibliotheken

```
import java.rmi.*;
```

Schritt 2:
Erweiterung von java.rmi.Remote

```
public interface NochEinRemoteInterface extends Remote
```

Schritt 3:
Deklarieren aller Methoden, die remote aufgeruft werden können

```
public String rufDenServer(String s) throws RemoteException;
```

2.4.2.4.2. Schrittweises modifizieren der Implementationsklasse

In diesem Beispiel nennen wir die Implementationsklasse `MeineKlasse`. Die Migration einer bereits definierten Klasse, die weder `Activatable` noch `UnicastRemoteObject` erweitert, zu einer Klasse, die aktivierbar ist, geschieht in drei Schritten:

1. Importieren der relevanten Klassen und Bibliotheken
2. Modifizieren der Klassendefinition, so dass die Klasse ein Interface implementiert, welches java.rmi.Remote erweitert.
3. Deklarieren eines Konstruktors mit zwei Argumenten `(ActivationID id, MarshalledObject data)`

Schritt 1:
Importieren der relevanten Klassen und Bibliotheken

```
import java.rmi.*;
import java.rmi.activation.*;
```

Schritt 2:
Modifizieren der Klassendefinition, so dass die Klasse ein Interface implementiert, welches java.rmi.Remote *erweitert.*

```
public class MeineKlasse implements NochEinRemoteInterface {
```

und

```
public interface NochEinRemoteInterface extends Remote
```

Schritt 3:
Deklarieren eines Konstruktors mit zwei Argumenten (`ActivationID id, MarshalledObject data`)

```
// Konstruktor für Aktivierung und Export; dieser Konstruktor wird
// von der Methode ActivationInstantiator.newInstance während der
// Aktivierung aufgerufen, um das Objekt zu konstruieren.
//
public MeineKlasse(ActivationID id, MarshalledObject data)
      throws RemoteException {

// Registrieren des Objekts beim Aktivierungssystem und
      // anschliessender Export an einen anonymen Port
      //
 System.out.println("MeineKlasse: Aufruf des Aktivierungs-Konstruktors");
      Activatable.exportObject(this, id, 0);
    }
```

2.4.2.4.3. Schrittweises entwickeln der Setup Klasse

In diesem Beispiel nennen wir die Setupklasse `Setup3`, wie sonst. Die Aufgabe dieser Klasse ist es, alle Informationen, die nötig sind, um das aktivierbare Objekt zu kreieren, ohne dass eine Instanz kreiert wird, dem Aktivierungssystem also dem RMI Daemon und der RMI Registry zur Verfügung zu stellen. Danach endet die Setup Klasse und braucht auch nicht mehr gestartet werden. Sie sehen dies an der Ausgabe des Daemons: im Fenster des Daemons sehen Sie, dass das remote Objekt einmal aktiviert wird (eine remote Instanz kreiert wird) und anschliessend nur noch die remote Methoden ausgeführt werden.

Der Bau der **`Setup3`** Klasse geschieht in sieben Schritten
1. Importieren der relevanten Klassen und Bibliotheken
2. Installieren des Security Managers
3. Kreieren einer Instanz der `ActivationGroup` Klasse
4. Kreieren einer Instanz der `ActivationDesc` Klasse
5. Entfernen allfälliger Referenzen auf die Implementationsklasse und registrieren am RMI Daemon
6. Binden des Stubs an einen Namen in der RMI Registry
7. verlassen der Setup Applikation

Schritt 1:
Importieren der relevanten Klassen und Bibliotheken

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
```

Schritt 2:
Installieren des Security Managers

```
System.out.println("Setup: Setzen des Security Managers");
System.setSecurityManager(new RMISecurityManager());

// Ab Java 2 muss eine Security Policy definiert werden
// für die ActivationGroup VM
// Das erste Argument von "put" ist der Schlüssel (aus der
// Hashtabelle),
// das zweite Argument ist dessen Wert, der Dateiname
```

```
//
Properties props = new Properties();
System.out.println("Setup: Setzen der Property 'java.security.policy'");
props.put("java.security.policy", "java.policy");
```

Schritt 3:
Kreieren einer Instanz der `ActivationGroup` *Klasse*

```
System.out.println("Setup: ActivationGroup");
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc beispielGruppe = new ActivationGroupDesc(props, ace);

// Registrieren der ActivationGroupDesc
// Dies liefert eine ID
//
ActivationGroupID agi =
           ActivationGroup.getSystem().registerGroup(beispielGruppe);

// Registrieren der ActivationGroupDesc
// Dies liefert eine ID
//
ActivationGroupID agi =
ActivationGroup.getSystem().registerGroup(beispielGruppe);

// kreieren der Gruppe
//
ActivationGroup.createGroup(agi, beispielGruppe, 0);
```

Schritt 4:
Kreieren einer Instanz der `ActivationDesc` *Klasse*

```
// die Zeichenkette "location" spezifiziert eine URL
// von der die Klassendefinition geladen werden kann
// falls das Objekt aktiviert werden muss.
// Am Schluss der URL
// steht ein "/".
//

String location =
"file://D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapitel16_RMI
nvocation/Programme/Aktivierung/";

System.out.println("Setup: Lokation (URL) - "+location);

// kreieren des Rests der Parameter
// für den ActivationDesc Konstruktor
//
MarshalledObject data = null;

// Das zweite Argument des ActivationDesc Konstruktors wird benötigt,
// um diese Klasse eindeutig zu identifizieren;
// die Lokation bezieht sich relativ auf die
// URL-formatierte Zeichenkette "location".
//
ActivationDesc desc = new ActivationDesc("Beispiel3.MeineKlasse", location,
                                         data);
```

Schritt 5:
Entfernen allfälliger Referenzen auf die Implementationsklasse und registrieren am RMI Daemon

```
// Registrieren bei rmid
//
NochEinRemoteInterface neri =
                    (NochEinRemoteInterface)Activatable.register(desc);
System.out.println("Setup: der Stub fuer die ActivatableImplementation
                        wurde gefunden" );
```

Schritt 6:
Binden des Stubs an einen Namen in der RMI Registry

```
// Binden des Stub an einen Namen in der Registry an Port 1099
//
Naming.rebind("MeineKlasse", neri);
System.out.println("Setup:die ActivatableImplementation wurde exportiert");
```

Schritt 7:
verlassen der Setup Applikation

```
System.out.println("Setup: Ende");
System.exit(0);
```

2.4.2.4.4. Übersetzen und starten der Beispielprogramme

Das Übersetzen und Starten geschieht in sechs Schritten:

1. Übersetzen der remote Interface-, Implementations- , Client- und Setup- Klassen
2. Generieren der Stubs (Skeletons werden seit Java 2 nicht mehr benötigt)
3. Starten der RMI Registry
4. Starten des RMI Daemons für die Aktivierung
5. Starten des Setup Programms
6. Starten des Clients

Schritt 1:
Übersetzen der remote Interface-, Implementations- , Client- und Setup- Klassen

```
@echo off
Rem
Rem Alles uebersetzen
echo RMI Aktivierung
echo alle Java Dateien uebersetzen
Rem
javac -classpath . *.java
pause
```

Schritt 2:
Generieren der Stubs (Skeletons werden seit Java 2 nicht mehr benötigt)

```
@echo off
Rem
echo rmic - generieren von Stubs und Skeletons
echo Bitte warten ...
cd ..
rmic -classpath . Beispiel3.MeinKlasse
```

```
cd Beispiel3
pause
```

Schritt 3:
Starten der RMI Registry

```
@echo off
echo RMIServer
echo Abbruch mit CTRL/C
set CLASSPATH=
cd ..
rmiregistry
cd Beispiel3
pause
```

Wie immer wird zuerst der CLASSPATH gelöscht, damit die Klassen und Stubs in der Registry gefunden werden können. Falls die Registry die Klassen im CLASSPATH findet, werden sie nicht registriert.

Schritt 4:
Starten des RMI Daemons für die Aktivierung

```
@echo off
echo RMI Daemon
echo Anhalten mit stop_rmid
set CLASSPATH=
cd ..
rmid -J-Djava.security.policy=rmid.policy
cd Beispiel2
pause
```

Der Daemon benötigt eine Security Policy, sonst kann er keine Klassen laden oder neue JVMs kreieren.

Schritt 5:
Starten des Setup Programms

```
@echo off
echo Java Activation
echo Setup
Rem
cd ..
java  -Djava.security.policy=rmid.policy -
Djava.rmi.server.codebase=file:/d:/UnterrichtsUnterlagen/ParalleleUndVertei
lteSysteme/Kapitel16_RMInvocation/Programme/Activierung/Beispiel3
Beispiel3.Setup3
cd Beispiel3
pause
```

Das Starten des Setup Programms besteht aus vier Bestandteilen:
1. dem java Befehl, mit dem die JVM gestartet wird
2. die Spezifikation verschiedener Properties in der Form `-D<Property>=<Wert>`
   In unserem Fall geben wir die Security Policy bekannt:
   `-Djava.security.policy=rmid.policy`
3. die Spezifikation der Codebase
   `-Djava.rmi.server.codebase=file:/...`
   Dabei darf der Dateiname bzw. die URL keine Leerstelle enthalten, wie Sie oben sehen.

4. der voll qualifizierte Package Namen des Setup Programms
   `Beispiel3.Setup3`

Schritt 5:
Starten des Setup Programms

```
@echo off
Rem
Rem RMI Aktivierung - Client
echo RMI Aktivierung
echo Client
Rem
cd ..
java -Djava.security.policy=java.policy Beispiel3.Client3 localhost
cd Beispiel3
pause
```

Als Parameter müsen wir beim Aufruf des Clients den Host angeben, auf dem sich unsere Registry und der Daemon befinden.

Ausgaben
Die Programme wurden mit Ausgaben erweitert, damit man besser erkennen kann, was in dieser verteilten Applikation abläuft. In diesem Beispiel müssen Sie etwas lange darauf warten, bis der Daemon bereit ist. Falls Sie den Client zu früh starten, erhalten Sie eine Fehlermeldung.

Setup:

```
Java Activation
Setup
Setup: Start
Setup: Setzen des Security Managers
Setup: Setzen der Property 'java.security.policy'
Setup: ActivationGroup
Setup: Lokation (URL) -
file://D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapit
el16_RMInvocation/Programme/Aktivierung/
Setup: der Stub fuer die ActivatableImplementation wurde gefunden
Setup: die ActivatableImplementation wurde exportiert
Setup: Ende
Taste drücken, um fortzusetzen . . .
```

Client:

```
RMI Aktivierung
Client
Client: setzen des SecurityManagers
Client: Lookup
Client: remote Referenz auf ein Objekt,
        welches Activatable implementiert.
        remoteRef=Beispiel3.MeineKlasse_Stub[RemoteStub [ref: null]]
Client: Aufruf der remote Methode'rufDenServer(Harry Potter bist Du
        hier?)';
Client: Ergebnis des Aufrufs
        Harry Potter bist Du hier?  - Ja, ich bin hier!
Taste drücken, um fortzusetzen . . .
```

RMI Daemon

```
RMI Daemon
Anhalten mit stop_rmid
Sat Oct 14 16:25:52 GMT+02:00 2000:
      ExecGroup-1:out:MeineKlasse:
                                Aufruf des Aktivierungs-Konstruktors
Sat Oct 14 16:25:54 GMT+02:00 2000:
      ExecGroup-1:out:MeineKlasse:
                                Methode 'rufDenServer()'
Sat Oct 14 16:27:07 GMT+02:00 2000:
      ExecGroup-1:out:MeineKlasse:
                                Methode 'rufDenServer()'
```

Pro Client Aufruf wird eine Zeile ausgegeben. Im Log werden weitere Daten erfasst.

2.4.3. Der Einsatz des Adapter Entwurfsmusters

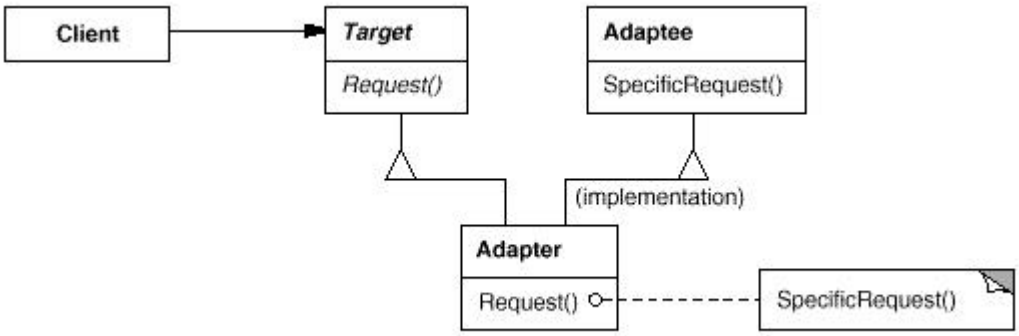
Machen wir uns zuerst mit dem allgemeinen Muster bekannt, gemäss dem Buch / der CD der Gang Of Four, "GoF":
Design Patterns: Elements of Reusable Object-Oriented Software
http://www.awl.com/cseng/titles/0-201-63361-2
Erich Gamma, Richard Helm, Ralph Johnson, und John Vlissides

Aufgabe des Entwurfsmusters
Konversion eines Interfaces einer Klasse in ein Interface, welches von einem Client benutzt werden kann. Adapter lassen Klassen zusammenarbeiten, die sonst wegen inkompatiblen Schnittstellen nicht zusammenarbeiten könnten.

Auch bekannt als

Wrapper

Motivation
Gelegentlich kann eine Klasse eines Werkzeugkastens nicht mehrfach eingesetzt werden, weil ihr Interface die spezifischen Anforderungen der Applikationen nicht erfüllt.



In unserem Fall wollen wir eine Adapter Klasse kreieren, welches das remote Interface implementiert, beim Aktivierungsdaemon und der RMI Registry vom Setup Programm registriert wird, und welche dann die Objektinstanz kreiert und die remote Methode an diese Instanz weiterleitet.

Der Vorteil dieses Musters ist, dass damit an der ursprünglichen nicht remote Klasse keinerlei Änderungen gemacht werden müssen. Das ist oft auch nötig, weil der Quellcode dieser Klasse nicht vorliegt oder aus irgend welchen Gründen nicht geändert werden darf.

2.4.3.1. Definition der Klassen

Schauen wir uns nun die einzelnen Klassendefinitionen an.
Sei unsere lokale (nicht remote) Klasse folgendermassen definiert:

```
package Adapter;

public class MeineNichtRemoteKlasse {
```

```
    private String resultat = null;
    // Verknüpfung zweier Zeichenketten
    //
    public String rufDenServer(String nochEineZeichenkette) {
        System.out.println("NichtRemote: rufDenServer()
                           Eingabe="+nochEineZeichenkette);
        resultat = nochEineZeichenkette + " Ja, ich bin hier!";
        System.out.println("NichtRemote: rufDenServer()
                           Rueckgabe="+resultat);
        return resultat;
    }
}
```

Jetzt schreiben wir ein remote Interface, welches die gewünschte Methode aus der obigen Klasse implementiert, aber eben remote:

```
package Adapter;

import java.rmi.*;

public interface NochEinRemoteInterface extends Remote
{
      public String rufDenServer(String s) throws RemoteException;
}
```

Damit wir diese Definition in unser System einbinden können, müssen wir entweder unsere nicht remote Klasse modifizieren, oder aber einen Adapter konstruieren. Da der Quellcode der Klasse `MeineNichtRemoteKlasse` eventuell nicht vorliegt, ist der Umweg über den Adapter universeller, wenn unter Umständen auch etwas langsamer und komplexer.

Unsere Adapter Klasse heisse `MeinNichtRemoteKlasseAdapter`. Dieser Adapter kreiert eine Instanz von `MeineNichtRemoteKlasse` im Konstruktor und ruft die Methode `MeineNichtRemoteKlasse.rufDenServer` auf.

```
package Adapter;

import java.rmi.*;
import java.rmi.activation.*;

public class MeinNichtRemoteKlasseAdapter implements NochEinRemoteInterface
{
    private String resultat = null;
    private MeineNichtRemoteKlasse mnrk;

    // Konstruktor für Aktivierung und Export; dieser Konstruktor
    // wird von der Aktivierungsmethode ActivationInstantiator.newInstance
    // während der Aktivierung aufgerufen, um das Objekt zu konstruieren.
    //
public MeinNichtRemoteKlasseAdapter(ActivationID id, MarshalledObject data)
        throws RemoteException {

        // Registrieren des Objekts beim Aktivierungssystem
        // und EXport an einen anonymen Port
        //
        System.out.println("Adapter: Aufruf des Aktivierungskonstruktors");
        Activatable.exportObject(this, id, 0);
        System.out.println("Adapter: Aktivierungskonstruktors - Objekt
                           wurde exportiert");
```

```
        try {
          System.out.println("Adapter: Aktivierungskonstruktors -
                                ClientHost="+Activatable.getClientHost() );
        } catch(java.rmi.server.ServerNotActiveException snae)
        {
          System.out.println("Adapter: Aktivierungskonstruktors -
                                ClientHost wurde nicht gefunden!");
        }

        // Instanz der Klasse 'MeineNichtRemoteKlasse' kreieren
        //
        mnrk = new MeineNichtRemoteKlasse();
        System.out.println("Adapter: Aktivierungskonstruktors -
                              Instanz="+mnrk.toString());
    }

    // Definieren der Methoden aus dem Interface NochEinRemoteInterface
    // Diese akzeptiert eine Zeichenkette, modifiziert sie und
    // liefert die modifizierte dem Client zurück
    //
    public String rufDenServer(String strAdapterZeichenkette)
           throws RemoteException {

      // Hier wird die Zeichenkette nicht modifiziert:
      // statt sie zu modifizieren, wird sie an die
      // Implementation in der Klasse 'MeineNichtRemoteKlasse'
      // weitergegeben, dort bearbeitet und zurück geliefert.
      //
        resultat = mnrk.rufDenServer(strAdapterZeichenkette);
        System.out.println("Adapter: rufDenServer()
                              Eingabe="+strAdapterZeichenkette);
        System.out.println("Adapter: rufDenServer() Rueckgabe="+resultat);
        return resultat;
    }
}
```

Die Adapter Klasse liefert einen Stub, wir müssen also dafür sorgen, dass dieser generiert wird.

Für die Definition aller Angaben zum aktivierbaren Objekt definieren wir eine Setup Klasse. Die Funktion dieser Klasse ist analog den früheren Beispielen. Einzig die remote Klasse muss mit korrekt angegeben werden, konkret die Adapter Klasse:

```
package Adapter;

import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class SetupMitAdapter {

    // Diese Klasse registriert Informationen über 'MeineKlasse'
    // bei der RMI Registry und dem RMI Daemon
    //
    public static void main(String[] args) throws Exception {
        System.out.println("Setup: Start");
      System.setSecurityManager(new RMISecurityManager());

        System.out.println("Setup: Setzen des Security Managers");
        // Ab Java 2 muss eine Security Policy definiert werden
        // für die ActivationGroup VM
```

```
        // Das erste Argument von "put" ist der Schlüssel (aus der
        // Hashtabelle),
        // das zweite Argument ist dessen Wert, der Dateiname
        //
      Properties props = new Properties();
        System.out.println("Setup: Setzen der Property
                           'java.security.policy'");
        props.put("java.security.policy", "java.policy");

        System.out.println("Setup: ActivationGroup");
        ActivationGroupDesc.CommandEnvironment ace = null;
        ActivationGroupDesc beispielGruppe = new ActivationGroupDesc(props,
                                                                     ace);

        // Registrieren der ActivationGroupDesc
        // Dies liefert eine ID
        //
        ActivationGroupID agi =
           ActivationGroup.getSystem().registerGroup(beispielGruppe);

        // kreieren der Gruppe
        //
        ActivationGroup.createGroup(agi, beispielGruppe, 0);

        // die Zeichenkette "location" spezifiziert eine URL
        // von der die Klassendefinition geladen werden kann
        // falls das Objekt aktiviert werden muss.
        // Am Schluss der URL
        // steht ein "/".
        //
        String location =
"file://D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapitel16_RMI
nvocation/Programme/Aktivierung/";
        System.out.println("Setup: Lokation (URL) - "+location);

        // kreieren des Rests der Parameter
        // für den ActivationDesc Konstruktor
        //
        MarshalledObject data = null;

        // Das zweite Argument des ActivationDesc Konstruktors wird
        // benötigt,
        // um diese Klasse eindeutig zu identifizieren;
        // die Lokation bezieht sich relativ auf die
        // URL-formatierte Zeichenkette "location".
        //
        ActivationDesc desc = new ActivationDesc
          ("Adapter.MeinNichtRemoteKlasseAdapter", location, data);

        // Registrieren bei rmid
        //
        NochEinRemoteInterface neri =
          (NochEinRemoteInterface)Activatable.register(desc);
        System.out.println("Setup: der Stub fuer die
                           ActivatableImplementation wurde gefunden" );

        // Binden des Stub an einen Namen in der Registry an Port 1099
        //
        Naming.rebind("MeinNichtRemoteKlasseAdapter", neri);
        System.out.println("Setup: die ActivatableImplementation wurde
                           exportiert" );
        System.out.println("Setup: Ende");
```

```
        System.exit(0);
    }
}
```

Die Client Klasse können wir weitestegehend übernehmen. Zu beachten ist das Einbinden des Adapters:

```
package Adapter;

import java.rmi.*;

public class Client{

    public static void main(String args[]) {

      String server = "localhost";
      if (args.length < 1) {
          System.out.println ("Usage: java Client <rmihost>");
          // System.exit(1);
            // server ist 'localhost'
      } else {
          server = args[0];
      }

      // Setzen eines Security Manager, damit der Client
      // den aktivierbaren Stub des Objekts heruntergeladen werden können.
      //

        System.out.println("Client: setzen des SecurityManagers");
      System.setSecurityManager(new RMISecurityManager());

      try {
          String location = "rmi://" + server +
                            "/MeinNichtRemoteKlasseAdapter";

        // Da mein Client keine Instanz eines Interfaces kreieren kann,
        // erhalten wir beim Lookup eine remote Referenz auf ein Objekt
        // welches MeinRemoteInterface implementiert.
        //
        // Diese remote Referenz von Naming.lookup(...) casten wir dann
        // (serialisierte Stub Instanz)
        // zu einem "MeinRemoteInterface", damit wir die Methoden des
        // Interfaces einsetzen können.
        //

        System.out.println("Client: Lookup");
        NochEinRemoteInterface neri =
            (NochEinRemoteInterface)Naming.lookup(location);

          System.out.println("Client: remote Referenz auf ein Objekt,");
          System.out.println("        welches Activatable implementiert.");
          System.out.println("        remoteRef="+neri.toString() );

        // der remote Methodenaufruf ergänzt die Zeichenkette 'resultat'
        //
        String resultat = "Harry Potter bist Du hier? ";

        System.out.println("Client: Aufruf der remote Methode
                                  'rufDenServer("+resultat+")';");
        resultat = (String)neri.rufDenServer(resultat);
        System.out.println("Client: Ergebnis des Aufrufs");
```

```
        System.out.println("        "+resultat);

      } catch (Exception e) {
        System.out.println("Client: Eine Exception wurde geworfen: " + e);
        e.printStackTrace();
      }
    }
}
```

2.4.3.2. Definition der Batch Dateien

Die Batch Dateien sind weitestgehend identisch mit jenen aus den früheren Beispielen.

Schritt 1:
übersetzen er Quelldateien

```
@echo off
Rem
Rem Alles uebersetzen
echo RMI Aktivierung
echo alle Java Dateien uebersetzen
Rem
javac -classpath . *.java
pause
```

Schritt 2:
Generieren der Stubs

```
@echo off
Rem
echo rmic - generieren von Stubs und Skeletons
echo Bitte warten ...
cd ..
rmic -classpath . Adapter.MeinNichtRemoteKlasseAdapter
cd Adapter
pause
```

Schritt 3:
Starten der RMI Registry

```
@echo off
echo RMIServer
echo Abbruch mit CTRL/C
set CLASSPATH=
cd ..
rmiregistry
cd Adapter
pause
```

Schritt 4:
Starten des RMI Aktivierungsdaemons

```
@echo off
echo RMI Daemon
echo Anhalten mit stop_rmid
set CLASSPATH=
cd ..
rmid -J-Djava.security.policy=rmid.policy
```

```
cd Adapter
pause
```

Schritt 5:
Starten des Servers, des Setup Programms

```
@echo off
echo Java Activation
echo Setup
Rem
cd ..
java  -Djava.security.policy=rmid.policy -
Djava.rmi.server.codebase=file:/d:/UnterrichtsUnterlagen/ParalleleUndVertei
lteSysteme/Kapitel16_RMInvocation/Programme/Activierung/Adapter
Adapter.SetupMitAdapter
cd Adapter
pause
```

Schritt 6:
Starten des Client Programms

```
@echo off
Rem
Rem RMI Aktivierung - Client
echo RMI Aktivierung
echo Client
Rem
cd ..
java -Djava.security.policy=java.policy Adapter.Client localhost
cd Adapter
pause
```

2.4.3.3. Ausgaben

Die Ausgaben sehen ähnlich aus, wie in den vorherigen Beispielen. Interessant ist der Aufbau des remote Objekts. Dies ist aus dem Log des Daemons ersichtlich.

Ausgabe 1:
Server / Setup

```
Java Activation
Setup
Setup: Start
Setup: Setzen des Security Managers
Setup: Setzen der Property 'java.security.policy'
Setup: ActivationGroup
Setup: Lokation (URL) -
file://D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapit
el16_RMInvocation/Programme/Aktivierung/
Setup: der Stub fuer die ActivatableImplementation wurde gefunden
Setup: die ActivatableImplementation wurde exportiert
Setup: Ende
Taste drücken, um fortzusetzen . . .
```

Ausgabe 2:
Client

```
RMI Aktivierung
Client
```

```
Client: setzen des SecurityManagers
Client: Lookup
Client: remote Referenz auf ein Objekt,
        welches Activatable implementiert.
        remoteRef=Adapter.MeinNichtRemoteKlasseAdapter_Stub[RemoteStub
                          [ref: null]]
Client: Aufruf der remote Methode 'rufDenServer(Harry Potter bist Du hier?
                                                )';
Client: Ergebnis des Aufrufs
        Harry Potter bist Du hier?  Ja, ich bin hier!
Taste drücken, um fortzusetzen . . .
```

Ausgabe 3:
RMI Daemon

```
RMI Daemon
Anhalten mit stop_rmid
```

Bemerkung: Konstruktion des remote Objekts

```
Sun Oct 15 14:07:31 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: Aufruf des Aktivierungskonstruktors
Sun Oct 15 14:07:35 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: Aktivierungskonstruktors - Objekt wurde exportiert
Sun Oct 15 14:07:35 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: Aktivierungskonstruktors - ClientHost=127.0.0.1
```

Bemerkung: Konstruktion des nicht remote Objekts

```
Sun Oct 15 14:07:35 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: Aktivierungskonstruktors -
                          Instanz=Adapter.MeineNichtRemoteKlasse@7934ad
```

Bemerkung: mehrfacher Aufruf der remote Methode

1. Aufruf

```
Sun Oct 15 14:07:35 GMT+02:00 2000:ExecGroup-0:out:
      NichtRemote: rufDenServer() Eingabe=Harry Potter bist Du hier?
Sun Oct 15 14:07:35 GMT+02:00 2000:ExecGroup-0:out:
      NichtRemote: rufDenServer() Rueckgabe=Harry Potter bist Du hier?
                                          Ja, ich bin hier!
Sun Oct 15 14:07:35 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: rufDenServer() Eingabe=Harry Potter bist Du hier?
Sun Oct 15 14:07:35 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: rufDenServer() Rueckgabe=Harry Potter bist Du hier?
                                          Ja, ich bin hier!
```

2. Aufruf

```
Sun Oct 15 14:08:19 GMT+02:00 2000:ExecGroup-0:out:
      NichtRemote: rufDenServer() Eingabe=Harry Potter bist Du hier?
Sun Oct 15 14:08:19 GMT+02:00 2000:ExecGroup-0:out:
      NichtRemote: rufDenServer() Rueckgabe=Harry Potter bist Du hier?
                                          Ja, ich bin hier!
Sun Oct 15 14:08:19 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: rufDenServer() Eingabe=Harry Potter bist Du hier?
Sun Oct 15 14:08:19 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: rufDenServer() Rueckgabe=Harry Potter bist Du hier?
                                          Ja, ich bin hier!
```

3. Aufruf

```
Sun Oct 15 14:08:42 GMT+02:00 2000:ExecGroup-0:out:
      NichtRemote: rufDenServer() Eingabe=Harry Potter bist Du hier?
Sun Oct 15 14:08:42 GMT+02:00 2000:ExecGroup-0:out:
      NichtRemote: rufDenServer() Rueckgabe=Harry Potter bist Du hier?
                                          Ja, ich bin hier!
Sun Oct 15 14:08:42 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: rufDenServer() Eingabe=Harry Potter bist Du hier?
Sun Oct 15 14:08:42 GMT+02:00 2000:ExecGroup-0:out:
      Adapter: rufDenServer() Rueckgabe=Harry Potter bist Du hier?
                                          Ja, ich bin hier!
```

2.4.4. Kreieren persistenter Daten mit Hilfe eines Marshalled Objekts

Dieser Abschnitt baut auf den vorhergehenden Abschnitten auf und sollte erst durchgearbeitet werden, nachdem Sie die vorherigen Abschnitte bearbeitet haben.

2.4.4.1. MarshalledObject

Die Klasse dient der Serialisierung von Objekten, erweitert die Klasse `Object` und implementiert `Serializable`:

```
public final class MarshalledObject
      extends Object
      implements Serializable
```

Ein `MarshalledObject` enthält einen Byte Stream mit der serialisierten Darstellung eines Objekts. Die `get()` Methode liefert eine neue Kopie des Originalobjekts, deserialisiert vom Byte Stream. Die Semantik der Serialisierung und Deserialisierung stammt aus dem RMI System.

In ihrer serialisierten Form werden die Klassen durch URL Informationen ergänzt. Diese URL Angabe gibt an, von wo die Klasse geladen werden kann, falls möglich.

Die `get()` Methode kann, falls die Klasse lokal nicht vorhanden ist, von der URL geladen werden.

2.4.4.2. ActivationDesc Klasse

Die MarshalledObject Klasse liefert einen flexiblen Mechanismus, um Persistenz- oder Initialisierungs-Daten zu liefern. Konkret geschieht dies mit Hilfe der **ActivationDesc**:

```
public final class ActivationDesc
extends Object
implements Serializable
```

Ein Activation Deskriptor enthält alle Informationen, die benötigt werden, um ein Objekt zu aktivieren:

- den Identifier der Objektgruppe,
- den voll qualifizierten Namen der Klasse
- die Lokation des Codes (der Klasse), ein Codebase URL Pfad
- den Restart "Modus" des Objekts und
- ein "marshalled" Objekt, welches die Objekt spezifischen Initialisierungsdaten enthält

Ein Deskripter, der beim Aktivierungssystem angemeldet ist, kann eingesetzt werden, um das vom Deskriptor spezifizierte Objekt wieder zu kreieren oder zu aktivieren.
Das MarshalledObject wird in der Objektbeschreibung als zweites Argument im remote Objekt Konstruktor eingesetzt, um das Objekt zu reinitialisieren / aktivieren.

2.4.4.3. Motivation

Im Falle eines UnicastRemoteObject ist es sehr einfach Informationen über die Kommandozeile an die Implementationsklasse weiter zu geben, da das Server Programm dauernd läuft und diese Parameter auswerten kann.

Im Falle der aktivierbaren Objekte sieht die Situation wesentlich anders aus: das Setup Programm endet in der Regel nachdem es alle Angaben an das Aktivierungssystem weitergeleitet hat.

In diesem Abschnitt lernen Sie einen Mechanismus kennen, wie
1. ein `java.util.Properties` Objekt konstruiert wird, mit dessen Hilfe die Lokation der `java.security.properties` Datei an den Konstruktor eines `ActivationGroupDescriptor` weiter gegeben wird, welcher seinerseits an den Konstruktor der `ActivationDesc` übergeben wird.
2. das `MarshalledObject`, welches an den `ActivationDesc` Konstruktor übergeben wird, ein `java.io.File` Objekt speichert, welches die Lokation des persistenten Datenspeichers repräsentiert.

Falls die persistenten Daten bereits existieren, in der Datei `persistentesObjekt.ser` vorhanden sind, werden diese gelesen und neue Transaktionsdaten hinzugefügt. Falls die Datei nicht existiert, wird die Datei neu angelegt und die Transaktionsdaten aus einem Vector darin als Objekt serialisiert.

Das Vorgehen:
1. Schrittweises kreieren des remote Interface
2. Schrittweises kreieren der Implementationsklasse
3. Schrittweises kreieren der Setup Klasse
4. Übersetzen und starten des Beispiels

Die Dateien:
1. Client.java
   Diese Klasse ruft eine Methode eines aktivierbaren Objekts auf
2. MeinePersistenteKlasse.java
   Diese Klasse ist aktivierbar, sie erweitert:
3. RemoteInterface.java
   Diese Klasse erweitert java.rmi.Remote und wird implementiert durch MeinePersistenteKlasse
4. Setup.java
   Die Klasse meldet das aktivierbare Objekt am RMI Aktivierungssystem und der Registry an

2.4.4.4. Schrittweises kreieren des remote Interface

Die Definition und Implementation des remote Interfaces `RemoteInterface` für unser Beispiel geschieht in drei Schritten:
1. importieren aller relevanten Klassen und Bibliotheken
2. erweitern des Remote Interfaces
3. deklarieren aller Methoden, welche remote aufgerufen werden können

Schritt 1 :
importieren aller relevanten Klassen und Bibliotheken

```
import java.rmi.*;
import java.util.Vector;
```

Schritt 2:
erweitern des Remote Interfaces

```
public interface RemoteInterface extends Remote {
```

Schritt 3:
erweitern des Remote Interfaces

```
public Vector rufDenServer(Vector v) throws RemoteException;
```

2.4.4.5. Schrittweises kreieren der Implementationsklasse

Unser Beispiel umfasst eine Implementationsklasse, `MeinePersistenteKlasse,` die wir in fünf Schritten kreieren wollen:

1. importieren aller relevanten Klassen und Bibliotheken
2. erweitern der `java.rmi.activation.Activatable` Klasse
3. deklarieren des zwei Argumente Konstruktors
4. implementieren der Methoden der Persistenzklasse
5. implementieren der remote Methoden aus dem Remote Interface

Schritt 1:
importieren aller relevanten Klassen und Bibliotheken

```
import java.io.*;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Vector;
```

Schritt 2:
erweitern der `java.rmi.activation.Activatable` *Klasse*

```
public class MeinePersistenteKlasse extends Activatable
    implements RemoteInterface {
```

Schritt 3:
deklarieren des zwei Argumente Konstruktors

Im Gegensatz zum üblichen Konstruktor müssen wir in diesem Fall die Persistenz einbauen. Die Behandlung der Persistenz ist recht einfach und ist aus dem Programmcode ersichtlich. Im Wesentlichen wird geprüft, ob bereits ein serialisiertes Objekt vorhanden ist.

Falls dies der Fall ist, wird es gelesen und durch die neuen Transaktionen ergänzt.
Falls kein serialisiertes Objekt gefinden wird, werden die Transaktionsdaten in ein neu kreiertes serialisiertes Objekt kreiert.

```
private Vector  transaktionen;
private File    speicher;

// Konstruktor für Aktivierung und Export
// Dieser wird von der Methode ActivationInstantiator.newInstance
// in der Aktivierung aufgerufen, um das Objekt zu konstruieren.
```

```
//
public MeinePersistenteKlasse(ActivationID id, MarshalledObject data)
      throws RemoteException, ClassNotFoundException, java.io.IOException {

      super(id, 0);
      System.out.println("Persistenz: Aktivierungs- und Export-
            Konstruktor");
      // Extrahieren  des File Objekts aus dem MarshalledObject
      // aus dem Konstruktor
      //
       System.out.println("Persistenz: Aktivierungs- und Export-
                        Konstruktor-Bestimmen der Datei");
      speicher = (File)data.get();

      if (speicher.exists()) {

            // falls das MarshalledObject existiert:
            // restore des Objektzustands
            //

            System.out.println("Persistenz: Konstruktor - Serialisiertes
                              Objekt wird gelesen");

            System.out.println("Persistenz: Konstruktor - restoreState()");

            this.restoreState();
      } else {

            System.out.println("Persistenz: Konstruktor - Init
                              Transaktionsvektors");
            transaktionen = new Vector(1,1);
            transaktionen.addElement("Persistenz: Init
                                    Transaktionsvektors");
      }
    }
```

Schritt 4:
implementieren der Methoden der Persistenzklasse

```
public class MeinePersistenteKlasse extends Activatable
public Vector liesTransaktionen() {
        System.out.println("Persistenz: liesTransaktionen");
      return transaktionen;
    }

    // Falls das MarshalledObject, welches dem Konstruktor übergeben wurde
    // eine Datei ist, wird es benutzt um die Transaktionsdaten zu
    // rekonstruieren.
    //
    private void restoreState() throws IOException, ClassNotFoundException
{
      System.out.println("Persistenz: Lesen des Serialisierten Objekts");
      System.out.println("Persistenz: restoreState()");
      File f = speicher;
      FileInputStream fis = new FileInputStream(f);
      ObjectInputStream ois = new ObjectInputStream(fis);
      transaktionen = (Vector)ois.readObject();
      ois.close();
    }
```

```
    private void saveState() {
      System.out.println("Persistenz: saveState()");
      FileOutputStream fos = null;
      ObjectOutputStream oos = null;

      try {
        File f  = speicher;
        try {
                fos = new FileOutputStream(f);
            } catch (IOException e1) {
                e1.printStackTrace();
                throw new RuntimeException("Persistenz: FileOutputStream
                                        konnte nicht kreiert werden");
            }
            try {
                  oos = new ObjectOutputStream(fos);
            } catch (IOException e2) {
                  throw new RuntimeException("Persistenz:
                         ObjectOutputStream konnte nicht kreiert werden");
            }
            try {
                  oos.writeObject(liesTransaktionen());
            } catch (IOException e3) {
                  throw new RuntimeException("Persistenz: Fehler beim
                                          Schreiben des Vectors");
            }
            try {
                  oos.close();
            } catch (IOException e3) {
                  throw new RuntimeException("Persistenz: Fehler beim
                                           Schliessen des Stream");
            }

      } catch (SecurityException e4) {
          throw new RuntimeException("Persistenz: Security Problem");
      } catch (Exception e) {
          throw new RuntimeException("Persistenz: Fehler beim Speichern des
                                                          Datenvektors");
      }
    }
```

Schritt 5:
implementieren der remote Methoden aus dem Remote Interface

```
// Definion der im RemoteInterface deklarierten Methoden
    //
    public Vector rufDenServer(Vector v) throws RemoteException {

      System.out.println("Persistenz: rufDenServer()");
      int limit = v.size();
      for (int i = 0; i < limit; i++) {
          transaktionen.addElement(v.elementAt(i));
      }

      // Speichern der Objektdaten in der Datei 'file'
      //
      System.out.println("Persistenz: rufDenServer() - Speichern des
                              Zustands");
      this.saveState();
      return transaktionen;
```

```
}
```

2.4.4.6. Schrittweises kreieren der Setup Klasse

Die Aufgabe der Setup Klasse bleibt weitestegehend unverändert: wir wollen damit dem Aktivierungssystem alle Informationen über die aktivierbare Klasse liefern, so dass ein remote Objekt kreiert werden kann.

Die Implementation der Setup Klasse geschieht in sieben Schritten:

1. importieren aller relevanten Klassen und Bibliotheken
2. installieren eines Security Managers
3. kreieren einer Instanz der ActivationGroup
4. kreieren einer Instanz der ActivationDesc
5. registrieren beim RMI Daemon
6. binden des Stubs an eien Namen der RMI Registry
7. verlassen der Applikation

Schritt 1 :
importieren aller relevanten Klassen und Bibliotheken

```
import java.io.File;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
```

Schritt 2 :
installieren eines Security Managers

```
System.setSecurityManager(new RMISecurityManager());
System.out.println("Setup: Setzen des Security Managers");
// Ab Java 2 muss eine Security Policy definiert werden
// für die ActivationGroup VM
// Das erste Argument von "put" ist der Schlüssel (aus der
// Hashtabelle),
// das zweite Argument ist dessen Wert, der Dateiname
//
Properties props = new Properties();
System.out.println("Setup: Setzen der Property 'java.security.policy'");
props.put("java.security.policy", "java.policy");
```

Schritt 3 :
kreieren einer Instanz der ActivationGroup

```
System.out.println("Setup: ActivationGroup");
ActivationGroupDesc.CommandEnvironment ace = null;
ActivationGroupDesc beispielGruppe = new ActivationGroupDesc(props, ace);

// Registrieren der ActivationGroupDesc
// Dies liefert eine ID
//
ActivationGroupID agi =
ActivationGroup.getSystem().registerGroup(beispielGruppe);

// kreieren der Gruppe
//
ActivationGroup.createGroup(agi, beispielGruppe, 0);
```

Schritt 4 :
kreieren einer Instanz der ActivationGroup

```
// die Zeichenkette "location" spezifiziert eine URL
// von der die Klassendefinition geladen werden kann
// falls das Objekt aktiviert werden muss.
// Am Schluss der URL
// steht ein "/".
//
String location =
"file:/D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapitel16_RMIIn
vocation/Programme/Aktivierung/MarshalledObjectPersistenz/";
System.out.println("Setup: Lokation (URL) - "+location);

// kreieren des Rests der Parameter
// für den ActivationDesc Konstruktor
//
// Angabe der Datei in der die Daten als Marshalled
// Objekt gespeichert werden sollen
System.out.println("Setup: MarshalledObject");
String strTemp =
"D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapitel16_RMInvocati
on/Programme/Aktivierung/MarshalledObjectPersistenz/persistentesObjekt.ser"
;
MarshalledObject data =  new MarshalledObject (new File(strTemp));

// Das zweite Argument des ActivationDesc Konstruktors wird benötigt,
// um diese Klasse eindeutig zu identifizieren;
// die Lokation bezieht sich relativ auf die
// URL-formatierte Zeichenkette "location".
//
System.out.println("Setup: ActivationDesc");
ActivationDesc desc = new ActivationDesc
  ("MarshalledObjectPersistenz.MeinePersistenteKlasse", location, data);
```

Schritt 5 :
registrieren beim RMI Daemon

```
// Registrieren bei rmid
//
RemoteInterface ri =
          (RemoteInterface)Activatable.register(desc);
System.out.println("Setup: der Stub fuer die ActivatableImplementation
                  wurde gefunden" );
```

Schritt 6 :
binden des Stubs an eien Namen der RMI Registry

```
// Binden des Stub an einen Namen in der Registry an Port 1099
//
Naming.rebind("MeinePersistenteKlasse", ri);
System.out.println("Setup: die ActivatableImplementation wurde exportiert"
);
```

Schritt 7 :
verlassen der Applikation

```
System.out.println("Setup: Ende");
```

```
System.exit(0);
```

2.4.4.7. Übersetzen und starten des Beispiels

Die Übersetzung und das Starten des Beispiels geschieht wie in den bereits bekannten Beispielen.

Vorgehensweise:

1. übersetzen des remote Interface, Implementation, Client und Setup Klassen
2. generieren des Stubs

3. starten der RMI Registry
4. starten des RMI Aktivierungs-Daemons
5. starten des Setup Programms
6. starten des Clients

Schritt 1:
übersetzen des remote Interface, Implementation, Client und Setup Klassen

```
@echo off
Rem
Rem Alles uebersetzen
echo RMI Aktivierung
echo alle Java Dateien uebersetzen
Rem
javac -classpath . *.java
pause
```

Alle Java Dateien werden übersetzt.

Schritt 2:
generieren des Stubs

```
@echo off
Rem
echo rmic - generieren von Stubs und Skeletons
echo Bitte warten ...
cd ..
rmic -classpath . MarshalledObjectPersistenz.MeinePersistenteKlasse
cd MarshalledObjectPersistenz
pause
```

Schritt 3:
starten der RMI Registry

```
@echo off
echo RMIServer
echo Abbruch mit CTRL/C
set CLASSPATH=
cd ..
rmiregistry
cd MarshalledObjectPersistenz
pause
```

Schritt 4:
starten des RMI Aktivierungs-Daemons

```
@echo off
echo RMI Daemon
echo Anhalten mit stop_rmid
set CLASSPATH=
cd ..
rmid -J-Djava.security.policy=rmid.policy
cd MarshalledObjectPersistenz
pause
```

Schritt 5:
starten des Setup Programms

```
@echo off
echo Java Activation
echo Setup
Rem
cd ..
java -Djava.security.policy=rmid.policy -
Djava.rmi.server.codebase=file:/d:/UnterrichtsUnterlagen/ParalleleUndVertei
lteSysteme/Kapitel16_RMInvocation/Programme/Activierung/MarshalledObjectPer
sistenz MarshalledObjectPersistenz.Setup
cd MarshalledObjectPersistenz
pause
```

Schritt 6:
starten des Clients

```
@echo off
Rem
Rem RMI Aktivierung - Client
echo RMI Aktivierung
echo Client
Rem
cd ..
java -Djava.security.policy=java.policy MarshalledObjectPersistenz.Client
localhost
cd MarshalledObjectPersistenz
pause
```

Ausgaben:

Setup:

```
Java Activation
Setup
Setup: Start
Setup: Setzen des Security Managers
Setup: Setzen der Property 'java.security.policy'
Setup: ActivationGroup
Setup: Lokation (URL) -
file:/D:/UnterrichtsUnterlagen/ParalleleUndVerteilteSysteme/Kapite
l16_RMInvocation/Programme/Aktivierung/MarshalledObjectPersistenz/
Setup: MarshalledObject
Setup: ActivationDesc
Setup: der Stub fuer die ActivatableImplementation wurde gefunden
Setup: die ActivatableImplementation wurde exportiert
Setup: Ende
```

```
Taste drücken, um fortzusetzen . . .
```

Client:

```
RMI Aktivierung
Client
Client: setzen des SecurityManagers
Client: Lookup
Client: remote Referenz auf ein Objekt,
        welches Activatable implementiert.

remoteRef=MarshalledObjectPersistenz.MeinePersistenteKlasse_Stub[RemoteStub
[ref:
null]]
Client: Aufbau des Resultat Vektors
Client: Aufruf der remote Methode 'rufDenServer()'
Client: Resultat:
Client: Ergebnis 0 Persistenz: Init Transaktionsvektors
Client: Ergebnis 1 Betrag einzahlen
Client: Ergebnis 2 Betrag abheben
Client: Ergebnis 3 Transferierter Betrag
Client: Ergebnis 4 Eingezogener Check
Client: Ergebnis 5 Einkauf im CooP
Taste drücken, um fortzusetzen . . .
```

RMI Daemon:

```
RMI Daemon
Anhalten mit stop_rmid
```

Bemerkung: Konstruieren und Aktivieren

```
Sun Oct 15 19:11:51 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: Aktivierungs- und Export-Konstruktor
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: Aktivierungs- und Export-Konstruktor-Bestimmen derDatei
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: Konstruktor - Serialisiertes Objekt wird gelesen
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: Konstruktor - restoreState()
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: Lesen des Serialisierten Objekts
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: restoreState()
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: rufDenServer()
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: rufDenServer() - Speichern des Zustands
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: saveState()
```

Bemerkung: Konstruieren und Aktivieren

```
Sun Oct 15 19:11:56 GMT+02:00 2000:ExecGroup-0:out:
      Persistenz: liesTransaktionen
Sun Oct 15 19:20:39 GMT+02:00 2000:ExecGroup-1:out:
      Persistenz: Aktivierungs- und Export-Konstruktor
Sun Oct 15 19:20:42 GMT+02:00 2000:ExecGroup-1:out:
      Persistenz: Aktivierungs- und Export-Konstruktor-Bestimmen derDatei
Sun Oct 15 19:20:42 GMT+02:00 2000:ExecGroup-1:out:
      Persistenz: Konstruktor - Init Transaktionsvektors
```

```
Sun Oct 15 19:20:42 GMT+02:00 2000:ExecGroup-1:out:
      Persistenz: rufDenServer()
Sun Oct 15 19:20:42 GMT+02:00 2000:ExecGroup-1:out:
      Persistenz: rufDenServer() - Speichern des Zustands
Sun Oct 15 19:20:42 GMT+02:00 2000:ExecGroup-1:out:
      Persistenz: saveState()
Sun Oct 15 19:20:42 GMT+02:00 2000:ExecGroup-1:out:
      Persistenz: liesTransaktionen
```

Bei jedem Aufruf des Client wird die Transaktionsliste ergänzt, also alle Transaktionen aus dem Vector hinten angefügt.

Sie sehen dies im Client Fenster und auch an der Grösse der serialisierten Objektdatei.

2.4.5. Dynamisches Herunterladen von Code mit RMI - Einsatz von `java.rmi.server.codebase` Property

Wir haben wiederholt die Codebase Property verwendet. Jetzt geht es darum, diese Eigenschaft etwas genauer unter die Augen zu nehmen und die Möglichkeiten eines Einsatzes besser zu verstehen.

Vorgehensweise:
1. Einleitung
2. Was ist die Codebase?
3. Wie funktioniert die Codebase?
4. Einsatz der Codebase in RMI zum herunterladen von Stubs und vielem mehr
5. Beispiele
6. Troubleshooting

2.4.5.1. Einleitung

Eine der wichtigsten Fähigkeiten von Java Systemen ist es, dynamisch Programme von beliebigen URLs herunter laden zu können, auch auf JVMs auf physikalisch unterschiedlichen Systemen.

Beispiele dafür kennen Sie bereits aus typischen Web Applikationen, die Applets. Bei Applets ist die Codebase der "Mutterserver", also der Server von dem das Applet stammt. Hier wollen wir nun Codebase aus Sicht von RMI betrachten.

Java RMI nutzt die Fähigkeit dynamisch Code zu laden, der nie auf der lokalen Maschine verfügbar ist oder war, im Speziellen können also Stubs (und in der alten Version Skeletons, die ab Java 2 nicht mehr benötigt werden) herunter geladen werden.

Der Begriff Codebase stammt aus der Java Programmiersprache, speziell dem ClassLoader. Der ClassLoader muss wissen, wo sich die Klassen befinden, die geladen werden müssen. Falls der Class Loader mit einem HTTP zusammen arbeitet, dann handelt es sich typischerweise um ein Applet mit einer Codebase, einem Tag im HTML Code.

```
<applet height=100 width=100
                codebase="meineKlassen/"
                code="MeineKlasse.class">
        <param name="ticker">
</applet>
```

2.4.5.2. Was ist eine Codebase?

Die Codebase definiert die Quelle oder einen Platz ab dem Klassen in die Java Virtuelle Maschine geladen werden können. Die Codebase ist also eine 'Wegleitung' an die JVM, in der spezifiziert wird, wo die Klassen zu finden sind.

CLASSPATH ist gewissermassen die 'lokale Codebasis': aus dem CLASSPATH ist ersichtlich, wo die Klassen lokal gefunden werden können. Der CLASSPATH kann relative oder absolute Adressen enthalten. Analog verhält es sich mit der Codebase.

2.4.5.3. Wie funktioniert die Codebase?

Um mit einem Applet kommunizieren zu können, muss der remote Client in der Lage sein mit ihm zu kommunizieren. Auf Applets kann man mit FTP (ftp://...) oder lokal ("file://...) zugreifen, obschon dies in der Regel mittels eines HTTP Servers geschieht:

1. der Client Browser fordert das Applet an, die Applet Class Datei, falls diese nicht im Client CLASSPATH gefunden wird.
2. die Class Definition des Applets und aller damit verbundenen Klassen wird vom Server über HTTP zum Client herunter geladen.
3. das Applet wird auf dem Client ausgeführt

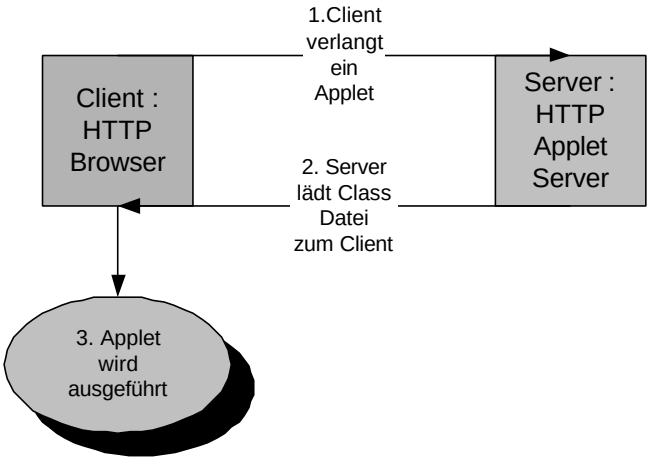


Abbildung 1 Herunterladen von Applets

Die Applet Codebase bezieht sich immer relativ zur URL der HTML Seite, in der sich der <Applet> Tag befindet.

2.4.5.4. Einsatz von Codebase in RMI

Da wir im Rahmen von RMI remote Objekte kreieren können, welche Methodenaufrufe von Clients in einer andern JVM empfangen kann, muss der Client eine Möglichkeit haben mit dem remote Objekt über das RMI Wire Protokoll (oder IIOP: siehe weiter unten) zu kommunizieren. Dabei muss der Client selbst sich nicht um das Protokoll kümmern. RMI verwendet spezielle Klassen, die sogenannten Stubs, *welche zum Client herunter geladen werden können*. Diese, die Stubs, können mit dem remote Objekt kommunizieren. Früher geschah diese Kommunikation zwischen dem Stub und dem Skeleton, den clientseitigen und serverseitigen 'Proxy-Objekten'.

Die `java.rmi.server.codebase` Property repräsentiert eine oder mehrere URLs, von denen diese Stubs und Klassen, die von den Stubs benötigt werden, heruntergeladen werden können.

Wie bei den Applets können Klassen, die benötigt werden, als 'file://... ' URLs spezifiziert werden. Allerdings können damit lediglich Klassen geladen werden, die sich auf dem selben Server befinden. Da dies eher einschränkend ist, werden in der Codebase, wie beispeilsweise bei Applets, in der Regel 'HTTP://...' oder 'FTP://...' URLs spezifiziert werden.

Daher sollten die Class Dateien mit Hilfe eines HTTP oder eines FTP Servers zur Verfügung gestellt werden. Das folgende Diagramm zeigt schematisch, was konkret passiert.

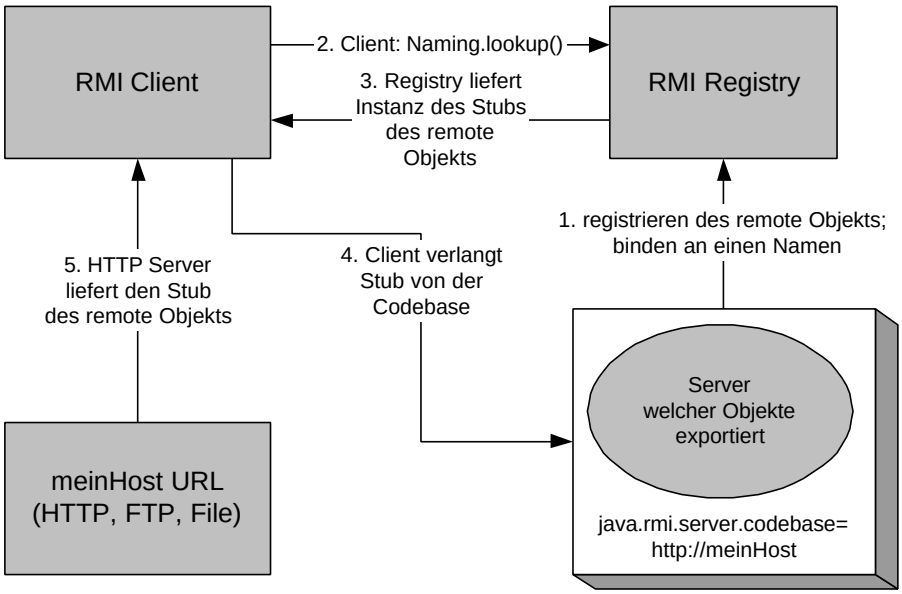


Abbildung 2 Herunterladen der RMI Stubs

1. Die Codebase des remote Objekts wird vom remote Objekt Server gesetzt, als `java.rmi.server.codebase` Property gesetzt. Das remote Objekt wird an einen Namen gebunden, in der RMI Registry. *Die Codebase, die beim Server spezifiziert wird, wird auch in die Registry eingetragen.*
2. Der RMI Client verlangt eine Referenz zu einem Namen eines remote Objekts. Die Referenz, *die Instanz des Stubs des remote Objekts*, wird vom Client benutzt, um remote Methoden aufzurufen.
3. Die RMI Registry liefert eine Referenz, *eine Instanz eines Stubs,* auf die benötigte Klasse.

   Falls die Klassendefinition für die Stub Instanz lokal, im CLASSPATH des Clients, der immer als erstes durchsucht wird, gefunden werden kann, dann lädt der Client die Klasse lokal.

   Falls die Definition für den Stub lokal im CLASSPATH des Clients nicht gefunden werden kann, versucht der Client die Klassendefinition mittels Codebase, remote zu erhalten.

4. Der Client verlangt die Klassendefinition von der Codebase. Die Codebase, die der Client benutzt, ist die URL, die der Stub Instanz hinzugefügt wurde, als die Stub Klasse von der Registry geladen wurde (wie in Schritt 1 erwähnt).

5. Die Klassendefinition für den Stub (und andere vom Stub benötigte Klassen) werden vom Server zum Client herunter geladen.

   Schritt 4 und 5 werden auch von der RMI Registry ausgeführt, um die Stubs zu laden.

6. Nun hat der Client alle Informationen, die benötigt werden, um die remote Methode aufzurufen. Die Stubs Instanz agiert als Proxy für das remote Objekt auf dem Server.

   Im Gegensatz dazu wird bei Applets der Code auf der lokalen JVM ausgeführt.

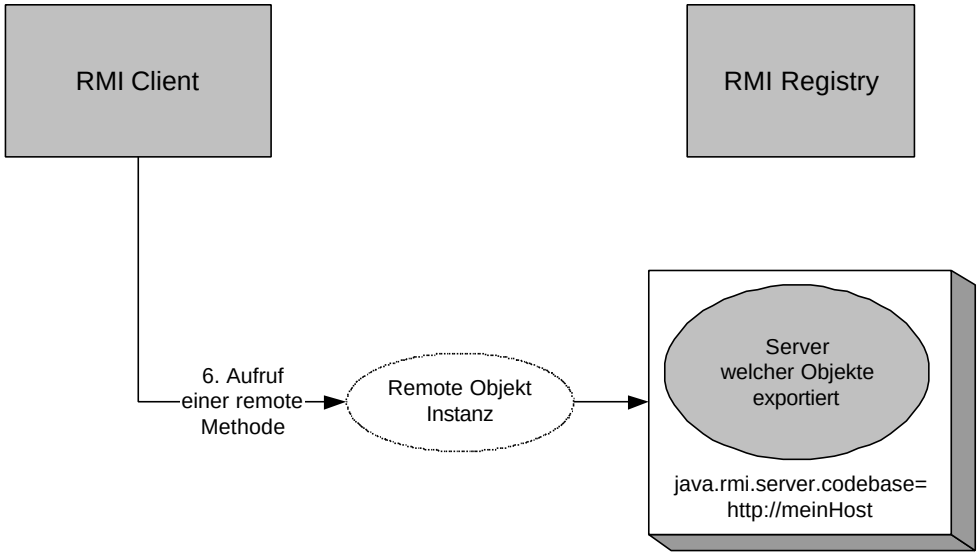


Abbildung 3 RMI Client ruft eine remote Methode auf

2.4.5.5. Einsatz der Codebase für mehr als das Herunterladen von Stubs

Neben Stubs können auch andere Klassen, Class Dateien aus Codebases heruntergeladen werden.

Bei remote Methodenaufrufen können unterschiedliche Fälle auftreten:
1. die remote Methode verwendet keine Argumente
2. die remote Methode verwendet ein oder mehrere Argumente

Dabei kann man wieder weitere Fälle unterscheiden, je nach Datentyp der Argumente:
1. alle Parameter und Rückgabewerte sind einfache / primitive Datentypen.
   In diesem Fall weiss das remote Objekt, wie die Daten interpretiert werden können. Daher wird der CLASSPATH oder eine allfällige Codebase nicht abgefragt.
2. mindestens ein remote Parameter oder Rückgabewert ist ein Objekt, dessen Klassendefinition lokal im CLASSPATH gefunden werden kann.
3. die remote Methode verwendet oder liefert ein Objekt, dessen Klassendefinition nicht lokal im CLASSPATH gefunden werden kann. In diesem Fall muss die Klassendefinition vom remote Server geladen oder zur Verfügung gestellt werden.
   Die Client Codebase wird verwendet um die benötigten Klassendefinitionen des Aufrufs zur Verfügung zu stellen.

2.4.5.6. Beispiele

Sie haben bereits einige Beispiele in den .bat Dateien gesehen. Windows Dateinamen sind etwas kritisch. Zur Illustration dieses Problems steht Ihnen der Class Loader von Sun zur Verfügung. Wenn Sie die Version auf dem Server verwende, erhalten Sie detaillierte Informationen darüber, auf welche Pfade zugegriffen wird:

Falls Sie file://d:\... verwenden kann ein Stack Dump resultieren, welcher bei der Verwendung von file:/d:\... nicht mehr vorhanden ist.

Im Falle eines Applets ist die Angabe der Codebase sehr einfach. Die Angabe steht im Applet Tag der HTML Seite.

Im Falle eines RMI Systems sieht die Situation etwas anders aus:
der Client befragt die RMI Registry für eine Referenz auf ein remote Objekt. Die URL weist auf die Stub Klasse und alle andern Klassen, welche von der Stub Klasse benötigt werden.

Die Codebase kann auf unterschiedliche Lokationen verweisen:

- die URL eines Verzeichnisses, in dem die Klassen korrekt inklusive Packagenamen abgelegt sind.
- die URL eines JAR Archivs, welches die Klassen korrekt inklusive Packagenamen enthält.
- mehrere Jar Dateien und oder Verzeichnisse, jeweils durch ein Leerzeichen getrennt, voll qualifiziert.

Achtung: *Alle Verzeichnisangeben müssen mit einem "/" abgeschlossen werden.*

Beispiele

```
Rem
java -classpath .;.. -Djava.rmi.security=java.policy -
Djava.rmi.server.codebase=file:/d:\UnterrichtsUnterlagen\ParalleleUndVertei
lteSysteme\Kapitel16_RMInvocation\Programme\Zeitserver\Server\
Server.ZeitServer
```

oder

```
Rem
java -classpath .;.. -Djava.rmi.security=java.policy -
Djava.rmi.server.codebase=file:/d:\UnterrichtsUnterlagen\ParalleleUndVertei
lteSysteme\Kapitel16_RMInvocation\Programme\Zeitserver\Server\/
Server.ZeitServer
```

oder

```
java -classpath . -Djava.rmi.security=java.policy
-Djava.rmi.server.codebase=file:///d:/UnterrichtsUnterlagen\ParalleleUnd
VerteilteSysteme\Kapitel16_RMInvocation\Programme\Zeitserver\Server\/
Server.ZeitServer
```

Der Dateiname in dieser Darstellung der Codebase muss unter WinNT wie oben in der Form **file:/<Laufwerk...> oder** file:///<Laufwerk **...>**angegeben werden, *nicht* file://<Laufwerk...>

```
Rem
echo ACHTUNG: der HTTP Server muss passend gesetzt sein (Root)
java -classpath .;.. -Djava.rmi.security=java.policy -
Djava.rmi.server.codebase=http://localhost/ Server.ZeitServer
```

Im Falle eines lokalen HTTP Servers, hier dem EMWAC HTTP Server, wird einfach die Internet Adresse angegeben.

Falls Sie mehrere Lokationen haben, darunter ein JAR Archiv, könnte die Codebase Angabe wiefolgt aussehen:

```
-Djava.rmi.server.codebase="http://www.joller-voss.ch/FHZ/Archive/Datum.jar
http://www.joller.au/Auckland/Server2.jar"
```

Die Pfade, URLs werden in diesem Fall in Anführungszeichen, jeweils durch ein Leezeichen getrennt, aufgeführt.

Jetzt dürfte Ihnen auch klar sein, warum ein Verzeichnisname wie "d:\MeineUnterlagen\Kapitel16 RMI" ungeschickt ist: diese Angabe in einer Codebase würde als zwei Verzeichnisse interpretiert!

2.4.5.7. Verschiedene Tips - Troubleshooting

Jede serialisierbare Klasse, inklusive RMI Stubs, kann herunter geladen werden, sofern Ihr RMI Programm korrekt konfiguriert ist.

Unter folgenden Bedingungen funktioniert das dynamische Herunterladen:

A. Stub und andere benötigte Klassen, die der Stub benötigt, sind von einer URL, die vom Client erreicht werden kann, herunterladbar.
B. die `java.rmi.server.codebase` Property, welche vom Server programm gesetzt wurde, oder im Falle eine aktivierbaren Klasse von deren Setup Programm, entspricht:
   - einer URL wie im Falle A.
   - entspricht einer URL, welche ein Verzeichnis darstellt und mit "/" endet.

C. die RMI Registry kann die Stub Klasse im lokalen Pfad CLASSPATH nicht finden. Dadurch wird der Eintrag in der Registry beim registrieren / binden durch die URL der Codebase ergänzt. Falls die Klasse lokal gefunden wird, fehlt diese Ergänzung in der Registry und der Client wird die Klasse nicht finden.

D. der Client muss einen SecurityManager laden, der es diesem erlaubt, den Stub herunterzuladen. Ab Java 2 bedeutet dies, dass auch eine Policy Datei vorhanden sein muss.

2.4.5.8. Typische Codebase Probleme mit dem RMI Server

Das erste Problem, mit dem Sie konfrontiert werden könnten, ist eine `ClassNotFoundException` beim Binden oder Rebind eines remote Objekts an einen Namen in der RMI Registry. Typischerweise tritt dieser Fehler auf, weil Sie die Codebase falsch oder nicht spezifiziert haben. Dadurch kann der Stub oder die Klassen des remote Objekts nicht gefunden werden.

Der Stub implementiert alle Methoden, welche das remote Objekt anbietet: es ist ja ein Proxy Objekt. Damit müssen auch alle Klassen verfügbar sein,welche als Parameter oder Rückgabewerte auftreten können.

Oft wird die Exception geworfen, weil eine Dateiangabe falsch ist, also entweder ein "/" zuviel oder ein "\" zu wenig verwendet wurde.

Ein anderer Grund kann sein, dass einige der benötigten Klassen an der angegebenen Stelle nicht gefunden werden. Eine übliche Praxis ist daher die Generierung eines JAR Archivs mit allen benötigten Klassen.

In diesem Fall würde folgende Exception geworfen werden:

```
java.rmi.ServerException: RemoteException occurred in server thread; nested
exception is:
      java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
      java.lang.ClassNotFoundException:
examples.callback.MessageReceiverImpl_Stub
java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
      java.lang.ClassNotFoundException:
examples.callback.MessageReceiverImpl_Stub
java.lang.ClassNotFoundException:
examples.callback.MessageReceiverImpl_Stub
      at
sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(Compiled
Code)
      at sun.rmi.transport.StreamRemoteCall.executeCall(Compiled Code)
      at sun.rmi.server.UnicastRef.invoke(Compiled Code)
      at sun.rmi.registry.RegistryImpl_Stub.rebind(Compiled Code)
      at java.rmi.Naming.rebind(Compiled Code)
      at examples.callback.MessageReceiverImpl.main(Compiled Code)
RemoteException occurred in server thread; nested exception is:
      java.rmi.UnmarshalException: error unmarshalling arguments; nested
exception is:
      java.lang.ClassNotFoundException:
examples.callback.MessageReceiverImpl_Stub
```

2.4.5.9. Probleme beim Starten des RMI Clients

Falls der Server funktioniert geht es darum, den Client starten zu können. Auch hier kann eine `ClassNotFoundException` geworfen werden. Dies kann beim Versuch ein remote Objekt in der Registry nachzuschauen passieren.

In diesem Fall wurde vermutlich die Klasse im CLASSPATH der RMI Registry gefunden. Die Fehlermeldung sieht typischerweise folgendermassen aus:

```
java.rmi.UnmarshalException: Return value class not found; nested exception
is:
      java.lang.ClassNotFoundException: MeinImpl_Stub
      at
sun.rmi.registry.RegistryImpl_Stub.lookup(RegistryImpl_Stub.java:109
      at java.rmi.Naming.lookup(Naming.java:60)
      at RmiClient.main(MeinClient.java:28)
```

2.4.6. Anwendung des Factory Entwurfmuster in RMI

Was ist eine Factory und warum sollte man eine Factory überhaupt benutzen?
Eine Factory, in diesem Kontext, ist ein Stück Software, welches ein Entwurfsmuster aus dem Buch der GoF implementiert, Design Patterns, Elements of Reusable Object-Oriented Software. Im allgemeinen ist eine Factory nützlich, falls man ein Objekt benötigt, welches das Kreieren oder den Zugriff auf andere Objekte regelt. Miteiner Factory kann die Anzahl Objekte die das RMI Programm (und das RMI System) benötigen reduziert werden.

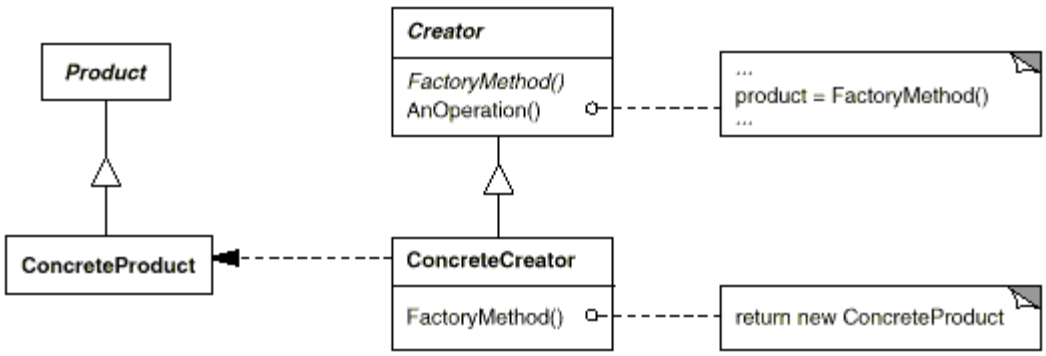
Hier ein Auszug aus dem obigen Standardwerk:

Aufgabe des Entwurfsmusters
Definition eines Interfaces für das Kreieren eines Objekts, wobei aber eine Unterklasse entscheidet, welche Klasse konkret instanziert wird. Die Factory Methode gestattet es die Instanzierung an eine Unterklasse zu delegieren.

Auch bekannt als

Virtual Constructor

Motivation
Frameworks benutzen abstrakte Klassen um Beziehungen zwischen Objekten zu definieren und zu unterhalten. Oft sind Frameworks auch für das Kreieren dieser Objekte zuständig.



2.4.6.1.1. Factory Beispiele aus der realen Welt

Die Bank
Nehmen wir an, Sie wollen ein Konto eröffnen. Sie gehen zur Bank und die Bank wird alles oder fast alles für Sie erledigen!

Nicht Sie, sondern die Bank wird das Geld in den Tresor oder sonstwo hin verschieben und investieren.

Ihr Bankkonto-Manager wird, nachdem Sie einige Papiere ausgefüllt haben, das Konto für Sie am Rechner eröffnen und den gesamten adminsitrativen Kleinkram erledigen, so dass Sie in Zukunft Geld einbezahlen oder abheben können.

Der Bankkonto-Manager ist eine Factory; der Bankrechner kreiert und manipuliert das Konto.

Die Bibliothek
Wie gelangt ein Buch, eine CD , ein Video oder ein DVD von der Bibliothek nach Ihnen zuhause?
Bevor Sie den Gegenstand ausleihen können, müssen Sie zuerst zur Bibliothek gehen, eventuell virtuell über das Internet. Dann muss die Bibliothek den gewünschten Gegenstand auf Ihren Namen erfassen. Erst dann können Sie den Gegenstand ausleihen.

Die Bibliothek ist die Factory für <Ausleiher, Gegenstände> Datensätze.

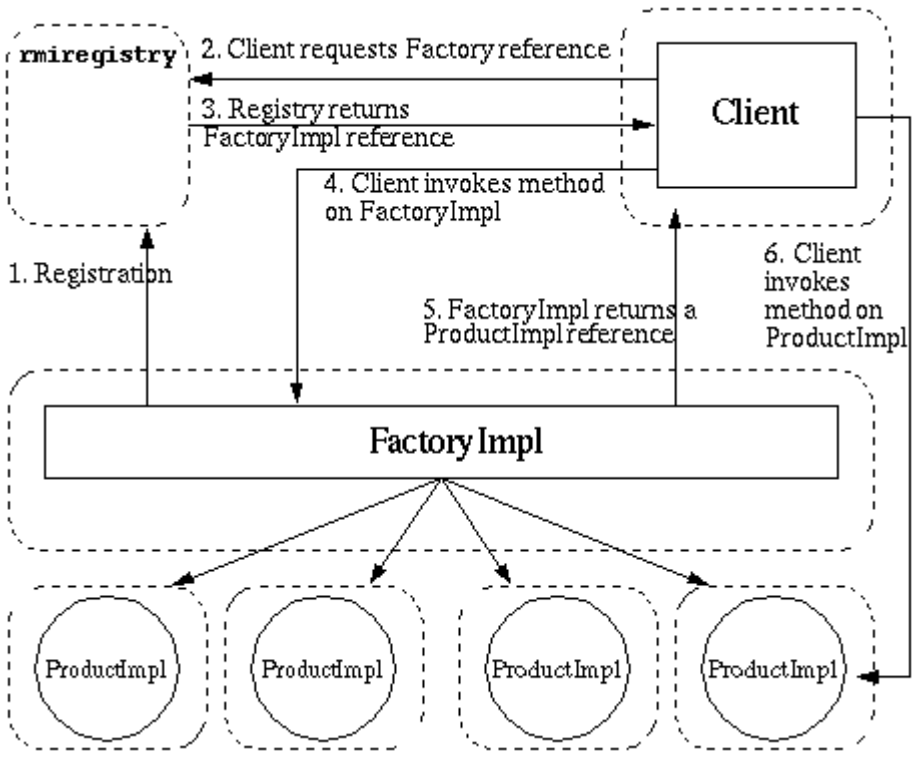
2.4.6.1.2. Einsatz von Factories in RMI

Wir haben bereits in einigen Beispielen Factories eingesetzt, ohne dass Ihnen dies explizit aufgefallen ist.

In einem RMI System spielen verschiedene Komponenten eine Rolle:

- der Server, welcher ein oder mehrere remote Objekte kreiert.
  Jedes dieser remote Objekte implementiert ein remote Interface.
- ein oder mehere Client(s), welche auf die RMI Registry, den Namensserver zugreift, um eine Referenz auf eines der remote Objekte zu erhalten.
- die RMI Registry, welche den Erstkontakt mit dem Server ermöglicht.
  Nachdem der Erstkontakt hergestellt wurde, kann der Client auch ohne Registry auskommen! Sie können dies leicht mit einem der Demo-Programme testen.

Das Gesamtbild sehen im untenstehenden Diagramm:



Damit lässt sich ein sehr flexibles und komplexes RMI System implementieren.

Nehmen wir der Einfachheit halber an, dass :

- es zwei remote Interfaces gibt, welche der Client versteht: Factory und Product
- FactoryImpl implementiert das Factory Interface und ProductImpl implementiert das Product Interface.

1. FactoryImpl registriert sich oder wird bei der RMI Registry registriert
2. Der Client verlangt eine Referenz auf die Factory
3. Die RMI Registry liefert eine remote Referenz auf ein FactoryImpl
4. Der Client ruft die remote Methode der FactoryImpl auf und erhält eine remote Referenz auf ein ProductImpl.
5. Die FactoryImpl liefert eine remote Referenz auf eine existierende ProductImpl oder auf eine gerade kreierte, basierend auf einer Client Anfrage.
6. Der Client ruft eine remote Methode der ProductImpl auf

2.4.6.2. Wie könnte eine Bank oder eine Bibliothek mit einer Factory implementiert werden?

Die Bank
Der BankkontoManager wäre ein remote Interface mit oder ohne remote Methoden. Diese Methoden liefern Objekte, welche das Konto Interface implementieren. Konto wäre ein Interface, welches alle Methoden/Operationen deklarieren würde, die eine Person bei einer Bank ausüben könnte: einzahlen, abheben, einen Kontoauszug verlangen oder die lestzten Transaktionen auflisten.

In RMI würde nur die Instanz des BankkontoManagers bei der RMI Registry eingetragen. Die Bankkonto Manager Implementation wäre eine Factory, welche remote Referenzen (oder eine serialisierte Instanz) der Konto Implementation liefert.

Die Bibliothek
Das Bibliothekenbeispiel einer Factory liesse sich analog implementieren. Wir könnten eine Factory definieren, welche Ausleihen für Bücher, CDs, ... kreieren könnte.

Ein Bankkonto Beispiel finden Sie auf dem Server. Versuchen Sie ein Bibliotheksystem zu definieren und zu implementieren.

2.4.7. RMI over IIOP (RMI-IIOP) - der Schritt in Richtung CORBA

Die OMG, Object Management Group, definierte bereits bevor es Java überhaupt gab, einen Standard CORBA, den Common Object Request Broker. Objekte, welche mit CORBA oder über CORBA kommunizieren möchten, müssen mit Hilfe einer Interface Definition Language IDL definiert werden. Die Kommunikation zwischen dem Client und dem Server in einem CORBA System geschieht mit Hilfe eines CORBA proprietären Protokolls. Damit CORBA Systeme untereinander kommunizieren können, wurde das Internet Inter ORB Protocol IIOP definiert. Java bietet mit RMI over IIOP die Möglichkeit, mit ORBs zu kommunizieren, oder aber einfach das RMI Wire Protokoll durch IIOP zu ersetzen.

Dieser Abschnitt ist also sozusagen der Übergang zu CORBA. Wir werden CORBA im nächten Kapitel untersuchen und Beispiele kennen lernen. Hier interessiert uns CORBA eher nicht.

2.4.7.1. Java RMI-IIOP Dokumentation

Bevor das IIOP für RMI nutzbar gemacht wurde musste man sich entscheiden, ob man RMI oder CORBA einsetzen möchte. RMI ist nur mit Java nutzbar; CORBA allerdings mit vielen unterschiedlichen Programmiersprachen.

RMI-IIOP kombiniert die Einfachheit von RMI und JavaIDL mit der Kommunikationsfähigkeit von CORBA und damit Software, welche in unterschiedlichen Programmiersprachen geschrieben wurde.

2.4.7.1.1. RMI-IIOP Tools Dokumentation

- IDL-to_Java Compiler (idlj)
- RMI Compiler (rmic)

2.4.7.1.2. RMI-IIOP Javadoc API Reference

- package org.omg.CORBA
- package org.omg.CosNaming

2.4.7.2. RMI-IIOP Programmierung

Themen:

- Einführung
- Hintergrundwissen
- Was ist RMI-IIOP
- Der neue rmic Compiler
- Der neue idlj Compiler
- Einsatz von IIOP in RMI Programmen
- Einschränkungen von RMI IIOP
- Konversion eines RMI Programms in ein RMI IIOP Programm
- Noch einige Ratschläge

2.4.7.2.1. Einführung

Im Abschnitt RMI over IIOP geht es darum, Ihnen zu zeigen, wie man Java RMI Programme schreibt, welche remote Objekte nutzen können und über das Internet Inter-ORB (IIOP) Protokoll kommuniziert. Ein RMI Programm kann durch kleine Änderungen und Einschränkungen zu einem IIOP RMI Programm umgebaut werden. Damit kann es auch auf CORBA Objekte zugreifen. RMI over IIOP kombiniert damit die Einfachheit von RMI mit der Universalität von CORBA.

Sie finden zusätzliche Informationen zum Thema unter folgenden Internet Adressen:

- der RMI-IIOP Home Page: diese enthält Links zu RMI-IIOP Dokumentationen, Beispielen, News und andern Web Sites.
- der RMI-IIOP FAQ Seite: dort finden Sie Hinweise und Antworten auf grundlegende Fragen zu RMI-IIOP.
- der Java RMI Home Page: diese enthält Links zur RMI Dokumentation, Beispiele, Spezifikationen und vieles mehr.
- der RMI API Dokumentation.
- der Java IDL Web Seite: dort können Sie grundlegendes über die Sun CORBA/IIOP Implementation nachlesen.
- dem Java Language to IDL Mapping Dokument: dies enthält eine detaillierte technische Spezifikation von RMI-IIOP.

2.4.7.2.2. Was ist RMI-IIOP?

RMI

Mit RMI kann man verteilte Programme in Java schreiben. RMI ist einfach zu benutzen, ohne dass man dazu die IDL, die Interface Definition Language lernen muss. Clients, remote Interfaces und Servers sind vollständig in Java geschrieben. RMI verwendet das Java Remote Method Protocol (JRMP) für die remote Java Objekt Komunikation. Sie haben RMI, falls Sie bis hier alles oder einiges gelesen haben, einen vertieften Einblick in RMI gewonnen.

Was RMI fehlt ist die Interoperabilität von RMI mit andern Sprachen. Da RMI über ein eigenes Wire Protokoll (JRMP) kommuniziert und dieses Protokoll proprietär ist, kann RMI nicht mit CORBA Objekten kommunizieren.

IIOP, CORBA und Java IDL

IIOP ist CORBA's Kommunicationsprotololl. Es definiert die Art und Weise wie Bits über eine Verbindung zwischen CORBA Clients und Servern. CORBA ist ein Standard für verteilte Objekt Architekturen, entwicklelt von der Object Management Group (OMG). Interfaces zu einem remote Objekt werden in einer platform-neutralen Interface Definition Language (IDL) beschrieben. Mappings von IDL auf specifische Programmiersprachen werden jeweils implementiert und gestatten eine Verbindung zu CORBA/IIOP.

Die JDK CORBA/IIOP Implementation ist bekannt als Java IDL. Zusammen mit dem `idltojava` Compiler kann Java IDL benutzt werden, um CORBA Objekte zu definieren, implementieren und auf CORBA Objekte aus Java zuzugreifen.

RMI-IIOP

Frühere Java Implementationen erlaubten die Kombination RMI mit CORBA nicht. Dies führte zu Entscheiden in einem frühen Stadium der Projekte:

- falls die Universalität von CORBA benötigt wurde, konnte die schnellere Entwicklung mit Java nicht ausgenutzt werden

2.4.7.2.3. Der neue rmic Compiler

Die RMI-IIOP Software wird mit einem neuen `rmic` Compiler ausgeliefert und kann IIOP Stubs und Ties generieren und IDL erzeugen.

Der Compiler besitzt auch einige neue Flags. Wir gehen hier nicht auf Details ein, weil wir später noch darauf zurück kommen werden.

Falls Sie kein Ausgabeverzeichnis angeben, mit -d, werden Stubs und Skeletons *nicht* in das aktuelle Arbeitsverzeichnis geschrieben sondern in das Unterverzeichnis, welches dem Package Namen entspricht.

2.4.7.2.3.1. Das -iiop Flag

Mit dem IIOP Flag werden Stubs und Skeletons (Tie Datei) gemäss IIOP generiert.
Stubs werden auch für abstrakte Interfaces generiert, welche Remore Exceptions werfen können.

2.4.7.2.3.2. Das -idl Flag

Mit der IDL Flag können Sie OMG konforme IDL Dateien, Beschreibungen der Schnittstelle angeben.

2.4.7.3. Wie setzt man IIOP in RMI Programmen ein?

Im Folgenden geben wir eine Checkliste für das Vorgehen, um RMI - IIOP Applikationen zu generieren.

1. **Falls Sie die RMI Registry benutzen**
   Dann müssen Sie auf JNDI, Java Nameing & Directory Interface, wechseln.

   a) in Client und Server: kreieren einen Context für JNDI
   neu:
   ```
   import javax.naming.*;
   ...
   Context initialNamingContext = new InitialContext();
   ...
   ```

   b) typischerweise im TMI Server: ersetzen Sie Lookup und Bind durch die entsprechenden Methoden in JNDI
   alt:
   ```
   ...
   import java.rmi.*;
   ...
   Naming.rebind("MeinObjekt", meinObj);
   ```
   neu:
   ```
   ...
   import javac.naming.*;
   ...
   initialNamingContext.rebind("meinObjekt", meinObj);
   ...
   ```

c) Falls der Client ein Applet ist, dann sieht der Import folgendermassen aus:

```
...
import java.util.*;
import javax.naming.*;
...
Hashtable env = new Hashtable();
env.put("java.naming.applet", this);
Context aic = new InitialContxext(env);
```

2. **Falls Sie die RMI Registry für Naming Services nicht einsetzen, dann haben Sie selbst irgend einen Bootstrapping Dienst eingerichtet.**
   Beispiel:
   Serverseitig

```
...
org.omg.CORBA.ORB meinORB = org.omg.CORBA.ORB.init(new String[0], null);
Wombat meinWombat = new WombatImpl();
javax.rmi.CORBA.Stub.meinStub =
   (javax.rmi.CORBA.Stub)PortableRemoteObject.toStub(meinWomba);
meinStub.connect(meinStub);
// nun ist meinWombat an meinORB gebunden.
FileOutputStream meineDatei = new FileOutputStream("test.tmp");
ObjectOutputStream meinStream = new ObjectOutputStream(meineDatei);
meinStream.writeObject(meinStub);
// damit ist das Objekt auf der Serverseite serialisiert
...
```

ClientSeitig

```
FileInputStream meineDatei = new FileInputStream("temp.tmp");
ObjectInputStream meinStream = new ObjectInputStream(meineDatei);
Wombat meinWombat = (Wombat)meinStream.readObject();
org.omg.CORBA.ORB meinORB = org.omg.CORBA.ORB.init(new String[0], null);
((javax.rmi.CORBA.Stub)meinWombat).connect(meinORB);
// nun ist MeinWombat verbunden mit meinORB
```

2.4.7.4. Einschränkungen der RMI Programme über IIOP

Damit RMI Programme über IIOP problemlos funktionieren, muss man folgende Einschränkungen beachten:

1. alle Konstanten in remote Interfaces müssen einfache Datentypen sein oder Zeichenketten, wleche zur Übersetzungszeit gesetzt werden.
2. Java Namen dürfen nicht mit IDL Namen in Konflikt sein. Welche Namen dies sind, steht in der Spezifikation von *Java Language to IDL Mapping* der OMG.
3. Methodennamen in remote Interfaces dürfen nur einmal definiert werden, Namenskonflikte müssen also vermieden werden. Auch Unterschiede nur in der Gross- / Kleinschreibung kann nicht ausreichen.
4. Die folgenden Konzepte sind nicht mehr anwendbar:
   - `UnicastRemoteObject`
   - `Unreferenced`
   - das DGC Interfaces
   - `RMISocketFactory`

2.4.7.5. Konvertiierung des RMI Hello World Programms in RMI-IIOP

Das folgende Beispiel haben wir bereits als normales RMI Programm realisiert und getestet. Das Hello World Applet haben wir bereits im Detail besprochen. Hier werden wir dieses Applet, Client und Server, in ein RMI - IIOP Applet und anschliessend in eine RMI - IIOP Anwendung umwandeln.

- `HelloImpl.java` ist der RMI server.
- `Hello.java` ist das remote Interface, welches von `HelloImpl` implementiert.
- `HelloApplet.java` ist der RMI Client.
- `Hello.html` lädt `HelloApplet`.

Die Umwandlung in RMI IIOP geschieht in folgenden Schritten:

Schritt 1:
Adaptieren der Implementationsklasse (Server) an RMI-IIOP

RMI klassisch:
`import javax.rmi.server.UnicastRemoteObject`:

RMI - IIOP:
`import javax.rmi.server.PortableRemoteObject`

Beispiel:
```
//klassisches RMI
//import java.rmi.server.UnicastRemoteObject;


//RMI over IIOP
import javax.rmi.PortableRemoteObject;
// JNDI naming package:
import javax.naming.*;

// HelloImpl extend PortableRemoteObject statt UnicastRemoteObject:
public class HelloImpl extends PortableRemoteObject
                              ...
//Einsatz der JNDI Registry, ans Stelle der RMI Registry
Context initialNamingContext = new InitialContext();
```

Schritt 2:
Ersetzen Sie die `rebind()` Methode von RMI durch jene von JNDI:

```
// klassiche RMI:

HalloImpl obj = new HalloImpl("HalloServer");
Naming.rebind("HalloServer", obj);


//RMI over IIOP

HalloImpl obj = new HalloImpl("HalloServer"); //unverändert
initialNamingContext.rebind("HalloServer",obj);
```

Schritt 3:
Anpassen des Applets

```
//RMI over IIOP
//Import der PortableRemoteObject Package:
import javax.rmi.PortableRemoteObject;

//kreieren eines JNDI initial Naming Contex
import java.util.*;
import javax.naming.*;
...
Hashtable env = new Hashtable();
env.put("java.naming.corba.applet", this);

//Die nächsten zwei Werte soll man in Zukunft auf als
//Applet Tags eingeben können

env.put("java.naming.factory.initial",
             "com.sun.jndi.cosnaming.CNCtxFactory");

// Ihr Host muss hier eingegeben werden
env.put("java.naming.provider.url", "iiop://<hostname>:900");

Context ic = new InitialContext(env);
```

Schritt 4:
Ersetzen Sie die RMI Version des Lookups durch die JNDIVersion

```
//klassisches RMI
import java.rmi.*;
...
Hallo obj = (Hallo)Naming.lookup("//" +
                     getCodeBase().getHost() + "/HalloServer");

//RMI oder IIOP
import javax.naming.*;
...
Hallo obj = (Hallo)PortableRemoteObject.narrow(
  initialNamingContext.lookup("HalloServer"),Hallo.class);
```

Schritt 5:
Anpassen der Applet Tag

Die zwei Zusätze sind, wie oben erwähnt, noch nicht funktionsfähig. Die Angabe der Parameter kann als Parameter angegeben werden.

```
<param name="java.naming.factory.initial"
       value="com.sun.jndi.cosnaming.CNCtxFactory"

<param name="java.naming.provider.url"
       value="iiop://<hostname>:900"
```

Schritt 6:
Übersetzen der Java Source Dateien

Dieser Schritt unterscheidet sich in keiner Art und Weise von der klassichen RMI Version.

Schritt 7:
Generieren der Stub und Tie Klassen

Der RMI Compiler muss mit dem IIOP Flag die Stubs und Tie (serverseitige Proxy) Klassen generieren.

```
rmic -iiop -d . HalloDuDa.HalloImpl
```

Schritt 8:
Starten the JNDI Name Server

```
tnameserv
```

Dies startet den JNDI Namenserver am Standard Port 900. Sie können auch einen andern Port verweden:

```
tnameserv -ORBInitialPort 1050
```

Schritt 9:
Start des Hallo Server

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
        -Djava.naming.provider.url=iiop://<hostname>:900
       HalloBeispiel.HalloImpl
```

Schritt 10:
Start des Hallo Client

Beispielsweise mit dem Appletviewer

```
appletviewer Hello.html
```

mit folgender HTML Seite

```
<html
   <title>Hallo World</title>
   <center <h1>Hallo World</h1></center
   ...

   Die Meldung des HalloServers lautet:
   ...

   <p>
   ...
   ...
   <applet code="HalloBeispiel.HalloApplet"
           width=500 height=120

   </applet>
   ...
   </BODY>

   </HTML>
```

2.4.7.6. Konversion des Client Applets zu einer Applikation

Die Umwandlung eines Applets in eine Applikation ist recht einfach, unabhängig davon, ob es sich um eine RMI oder eine Standard-Applikation handelt. Sie finden funktionsfähigen Programmcode auf dem Server oder der CD, inklusive Batch Dateien.

2.4.7.6.1. Anpassen einer Client Applikation an RMI-IIOP

Schritt 1:
Konversion des Applets in eine Applikation

- kopieren der Datei:
  HalloApplet.java in HalloApplikation.java
- ändern des Package Namens und des Namens der Klasse, beispielsweise HalloAppl
- entfernen von extends Applet
- ändern von ... `init()` auf `public static void main(String[] args])`
- verschieben der Anweisung `message = "";`
- eliminieren der `paint()` Methode.

Schritt 2:
Ersetzen der RMI Registry durch JNDI

```
import javax.naming.*;
...
Context initialNamingContext = new InitialContext();
```

Schritt 3:
Einsatz von JNDI lookup() *an Stelle der RMI Version und ersetzen des RMI Castings durch einen Aufruf von* `javax.rmi.PortableRemoteObject.narrow()`:

RMI Programmcode

```
import java.rmi.*;
...
Hallo obj = (Hallo)Naming.lookup("//" +
                    getCodeBase().getHost() + "/HalloServer");
```

RMI IIOP Programmcode

```
import javax.naming.*;
...
Hallo obj = (Hallo)PortableRemoteObject.narrow(
                        initialNamingContext.lookup("HalloServer"),
                        Hallo.class);
```

Host und Port werden wir beim Starten des Servers festlegen.

Schritt 4:
Übersetzen des Quellcodes der Hallo Applikation

```
javac -d . HalloApp.java
```

In unserem Beispiel gehen wir davon aus, dass Stub und Tie bereits generiert wurden, aus der Applet Applikation. Andernfalls müsste dieser Schritt jetzt noch eingeschoben werden:

Schritt 5:
Generieren der Stub und Tie Klassen

Der RMI Compiler muss mit dem IIOP Flag die Stubs und Tie (serverseitige Proxy) Klassen generieren.

```
rmic -iiop -d . HalloApplikation.HalloImpl
```

Schritt 6:
Starten des Name-Servers und des Hallo Servers:

Starten the JNDI Name Server

```
tnameserv
```

Dies startet den JNDI Namenserver am Standard Port 900. Sie können auch einen andern Port verweden:

```
tnameserv -ORBInitialPort 1050
```

Start des Hallo Server

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
        -Djava.naming.provider.url=iiop://<hostname>:900
       HalloApplikation.HalloImpl
```

Schritt 7:
Start the Hello Application Client

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
     -Djava.naming.provider.url=iiop://<localhost>:900
      HalloApplikation.HalloApp
```

2.4.7.6.2. Was Sie beachten sollten

2.4.7.6.3. Server müssen Thread Safe sein

Da remote Methoden auf einem Server von verschiedenen Clients gleichzeitig ausgeführt werden können, muss die Applikation Thread safe sein.

2.4.7.6.4. Hashtables mit identischen Vector Keys

Sie müssen aufpasssen, dass die Einträge in der Hashtabelle eindeutig sind. Dies ist bei alten Applikation, mit JDK1.1, eher ein Problem als in Java 2.

2.4.7.7. Interoperabilität mit andern ORBs

RMI-IIOP kann mit andern ORBs kommunizieren, sofern diese der CORBA 2.3 Spezifikation entsprechen. Ältere ORBs werden nicht unterstützt, da die IIOP Verschlüsselung nicht übereinstimmt. Die meisten kommerziellen ORBs sind leider alt und unterstützen CORBA 2.3 noch nicht. Aber das ist eine Frage der Zeit.

Gängige ORBs lernen wir im folgende Kapitel über CORBA kennen.

2.4.7.8. Bekannte Probleme

Hier eine Liste der noch vorhandenen Probleme:

- JNDI 1.1 unterstützt `java.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory` nicht als Applet Parameter. Damit dieser Parameter angegeben werden kann, muss er auf als Property an den `InitialContext` Konstruktor übergeben werden. JNDI 1.2. löst dieses Problem.
- Port 900 ist auf Solaris bereits reserviert. Auf Solaris muss daher ein höherer Port verwendet werden, höher als 1024.
- auf Solaris kann eine "out of file descriptors" Ausnahme auftreten, da die RMI-IIOP jede Menge Dateien verwendet. Dies lässt sich beheben, indem wir: `ulimit -n 90` eingeben.

Damit haben wir den RMI Praxisteil abgeschlossen. Sie sollten sich auch noch mit demRMI Übungsteil beschäftigen. Die Übungen leiten Sie durch typische Anwendungen im RMI Umfeld, von dynamischen Klassen bis zu verteilten Garbage Colletors.

2.4.8. Alternative Ansätze - RMI Light

Verschiedene Firmen und Institute arbeiten an "leichten" RMI Versionen, also Versionen, welche wesentlich weniger Speicherplatz benötigen, RMI für embedded Systems ....

Heute sind diese Ansätze aber in der Regel noch kaum einsetzbar. Sobald wir oder andere weitere, brauchbarere Ansätze vorliegend haben, wird dieser Abschnitt entsprechend ergänzt werden.