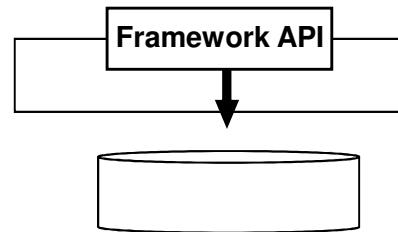


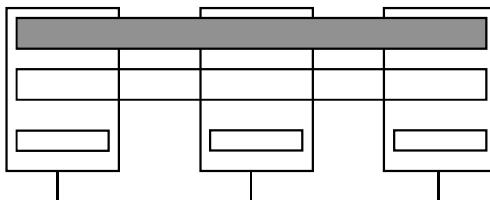
Software Engineering III – Wo sind wir?

Softwarearchitekturen

Softwarearchitekturen:
Begriffe & Fallbeispiel
Persistenz-Framework

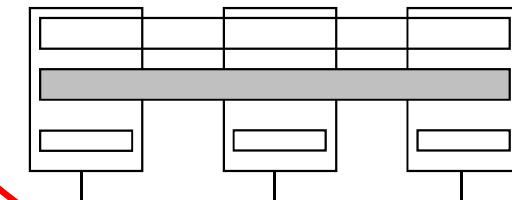


Verteilte Softwarearchitekturen



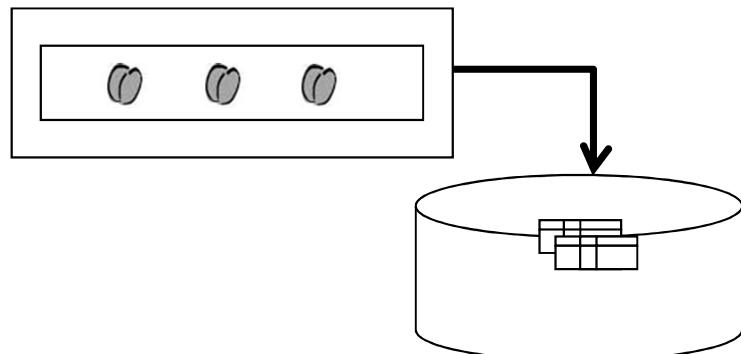
Middleware/Verteilung

Sockets, RMI, MoM/JMS
Web Services



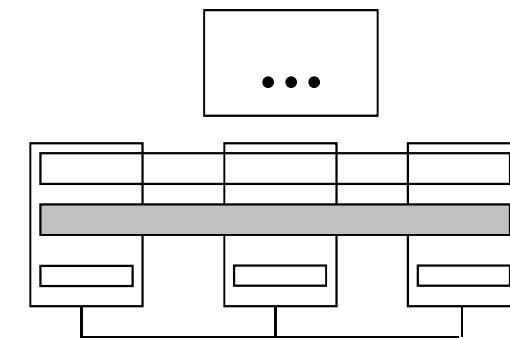
Enterprise Java

Java EE (EJB, ...)



Verteilung – weitere Konzepte

Weitere Middleware



Gliederung

1. Architektur der Persistenzschicht / Framework

2. Persistenz Framework

3. Verteilte Software-Architekturen

4. Sockets und RMI

 4.1 Sockets

 4.1.1 Interprozess-Kommunikation und Sockets

 4.1.2 Code-Beispiel Sockets

4.2 RMI (Remote Method Invocation)

4.2.1 RMI – Programmiermodell

4.2.2 Deployment

4.2.3 Parameterübergabe

4.2.4 Beispiel: Chat per RMI

4.2.5 Corba

4.3. Zusammenfassung

4.2.1 RMI – Programmiermodell

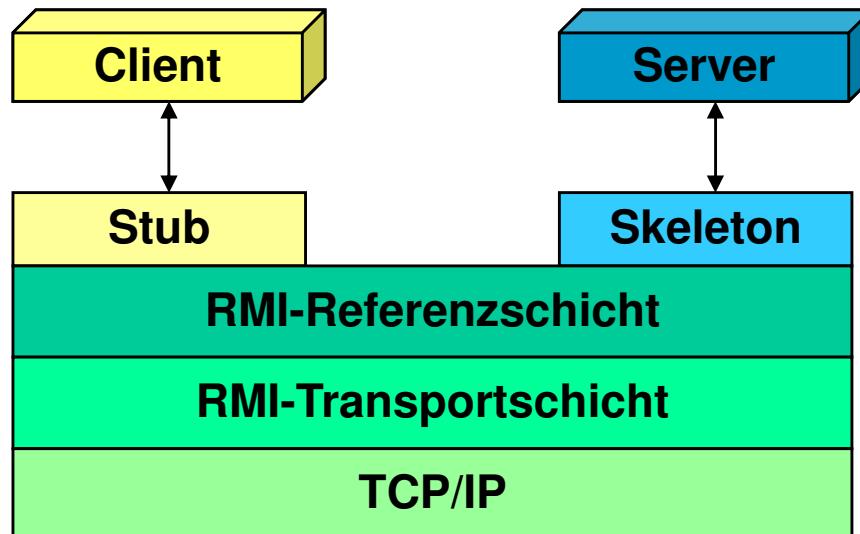
Remote Method Invocation (RMI) - (= Remote Procedure Call in Java)

- **Kommunikation mit entfernten Objekten**
 - Aufruf entfernter Methoden
 - Details werden vor dem Anwendungsentwickler verborgen
- **Referenzieren entfernter Objekte**
 - über Namensdienst (RMI-Registry)
 - über Austausch entfernter Referenzen (Parameter/Rückgabewerte entfernter Methoden)
- **Laden von verteilten Objekten**
 - entfernte Objekte können als Methoden-Parameter übergeben werden
 - RMI beinhaltet einen Mechanismus zum Laden von Objektcode
- **RMI und Java EE**
 - RMI oder RMI-IIOP ist meist auch (interne) Kommunikationsbasis für Java EE Application Server.

4.2.1 RMI – Programmiermodell

RMI-Compiler erzeugt

- **Stub:** Stellvertreterobjekt, das Clientaufruf an Server weiterreicht
- **Skeleton:** nimmt Aufrufe des Stubs entgegen und leitet sie an Serverobjekt weiter
- RMI-Referenzschicht stellt Namensdienst (=Registry) zur Verfügung
- RMI-Transportschicht verwaltet Verbindungen und wickelt Kommunikation ab



4.2.1 RMI – Programmiermodell

Stub-Klasse: (nur Beispiel!)

- baut Socket-Verbindung zu Server auf
- schickt Namen der Methode + Parameter und holt Ergebnis ab

```
public class HelloImpl_Stub implements Hello {  
    Socket socket = new Socket("nbak",4711);  
    ObjectOutputStream outStream =  
        new ObjectOutputStream(socket.getOutputStream());  
  
    // für remote Methode sayHello():  
    public String sayHello() throws RemoteException{  
        outStream.writeObject("sayHello");  
        ObjectInputStream in =  
            new ObjectInputStream(socket.getInputStream());  
  
        return (String)in.readObject();  
    }  
}
```



4.2.1 RMI – Programmiermodell

Skeleton-Klasse (nur Beispiel!)

- erzeugt Socket auf dem selben Port wie Stub
- wartet auf Methoden-Aufruf von Client und delegiert diesen an Objekt
- liefert Rückgabewert über Port

```
public class HelloImpl_skeleton extends Thread{  
    HelloImplementierung myHello;  
    ObjectInputStream inStream =  
        new ObjectInputStream(socket.getInputStream());  
  
    public void run(){  
        ServerSocket serverSocket = new ServerSocket(4711);  
        Socket socket = serverSocket.accept();  
  
        String method = (String)inStream.readObject();  
        if(method.equals("sayHello")){  
            String returnString = myHello.sayHello();  
            ObjectOutputStream outStream =  
                new ObjectOutputStream(socket.getOutputStream());  
            outStream.writeString(returnString);  
        }  
    }  
}
```



4.2.1 RMI – Programmiermodell

RMI-Beispiel – Serverseite (1/3)

1. Interface für Remote-Methoden

- wird von **beiden Rollen** (Client und Server) benötigt
- muss **von Remote** aus Paket java.rmi **abgeleitet** sein (Marker-Interface)
- alle Methoden müssen eine **RemoteException** werfen
- **Namenskonvention:** Remote-Interfaces haben fachlichen Namen ohne Suffix

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

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```



4.2.1 RMI – Programmiermodell

RMI-Beispiel – Serverseite (2/3)

2. Klasse zur Implementierung des Remote-Interface

- **implementiert** das Interface (Hello)
- jedes entfernte Objekt ist **Unterklasse** von **UnicastRemoteObject**
- benötigt **Konstruktor**, der **RemoteException** wirft
- **Namenskonvention:** <Interface-Name>Impl.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl
    extends UnicastRemoteObject implements Hello {

    public HelloImpl() throws RemoteException { super(); }

    public String sayHello() throws RemoteException{
        return "Hello World!";
    }
}
```

4.2.1 RMI – Programmiermodell

RMI-Beispiel – Serverseite (3/3)

3. Remote-Objekte erzeugen und in *RMI-Registry* anmelden

- auf Server-Rechner wird ein Objekt *obj* der Implementierungs-Klasse **erzeugt**
- es wird unter dem Namen *remoteHello* beim Rechner „nbak“ **registriert**
 - unter diesem Namen kann ein Client auf das Objekt zugreifen
- **Standard-Port-Nummer** für den rmiRegistry-Prozess auf Server ist **1099**

```
import java.rmi.Naming;

public class HelloServer {
    public static void main(String args[]) {
        try {
            HelloImpl obj = new HelloImpl();
            Naming.rebind("rmi://nbak/remoteHello", obj);
        } catch (Exception e) { ... }
    }
}
```



4.2.1 Hello World – Client-Seite

RMI-Beispiel – Clientseite

1. Remote-Interface

```
public interface Hello extends Remote {analog zum Server}
```

2. Client ruft Methode von remote-Object auf

- `lookup(..)` hole remote-Objekt namens `remoteHello` aus Registry auf nbak
 - wichtig: es wird das Remote-Interface verwendet !
 - registrieren des `RMISecurityManager`-Objekts ist nur erforderlich, wenn der Client den Stub-Code dynamisch vom Server laden will (s. u.)

```
import java.rmi.*;
public class RmiClient {
    public static void main(String[] args) {
        try {
            //System.setSecurityManager(new RMISecurityManager());
            Hello obj =(Hello)Naming.lookup("rmi://nbak/remoteHello");
            String message = obj.sayHello();
            System.out.println(message);
        } catch (Exception e) {...}
    }
}
```

4.2.2 Deployment – Starten des Servers

1. Start des RMIRegistry-Prozesses im Betriebssystem

```
rmiregistry -J-Djava.rmi.server.useCodebaseOnly=false &  
           // starte Prozess mit Default-Port 1099
```

- **Server** registriert Remote-Objects in RMIRegistry mit Naming.rebind(..)
- Client kann dieses Objekt in RMIRegistry mit Naming.lookup(..) finden

2. Compilieren der Server-Klassen

Hello.java, HelloImpl.java, HelloServer.java

```
javac *.java
```

3. Erzeugen der Stub-Klassen mit rmi-Compiler (seit Java 5 optional)

```
rmic -v1.2 -nowarn HelloImpl
```

- erstellt: **HelloImpl_Stub.class**
- Erzeugen der Skeleton-Klasse seit RMI v1.2 optional
- Hinweis: Option „-keep“ erzeugt auch **HelloImpl_Stub.java**

4. Code ausführen, der Server-Objekt in Registry registriert

```
java HelloServer &
```



4.2.2 Deployment – Starten des Servers

Exkurs: Server-Konfiguration zum dynamischen Laden der Stubs

Problem: auf den Clients befinden sich nicht die erforderlichen Stub-Klassen

Lösung: der Client lädt sich die Stubs über einen Webserver (oder ein Netzwerkverzeichnis)

4. Starten eines Webservers: Client holt Stub-Klasse per http

```
start java ClassFileServer 2001 c:\home\javaStuff\rmi\Server
```

1. Parameter: Port an dem gelauscht wird
2. Parameter: Pfad in dem Dateien (Stub-Files) liegen
 - ClassFileServer von Sun: Programm, mit dem man per HTTP Klassen laden kann

5. HelloServer ausführen erzeugen und registrieren

```
start java -Djava.rmi.server.codebase=http://nbak:2001/ HelloServer
```

- Stub-Klassen werden per Webserver übertragen: codebase=http://nbak:2001/
- Stub-Klassen von Fileserver geladen übertragen: **codebase=file:/c\:/home\rmis/**

- Tipp: wenn Stub-Klassen nicht gefunden werden, codebase-Parameter explizit setzen !



4.2.2 Deployment - Starten des Client

1. Compilieren der Client-Klasse

```
javac RmiClient.java
```

2. Eventuell: Erstellen einer Policy-Datei

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

- Policy-Datei (hier `.policy`) erlaubt Client den Connect zur Server-Registry
- erforderlich, wenn Stub-Klassen dynamisch vom Server geladen werden sollen → Anhang

3. Ausführen des Clients

```
java RmiClient
```

(ohne Security-Policy)

- `HelloImpl_Stub.class` ins Client-Verzeichnis kopieren

```
java -Djava.security.policy=policy RmiClient
```

- bei der Ausführung wird die im `policy` -File spezifizierte Sicherheit zugrunde gelegt

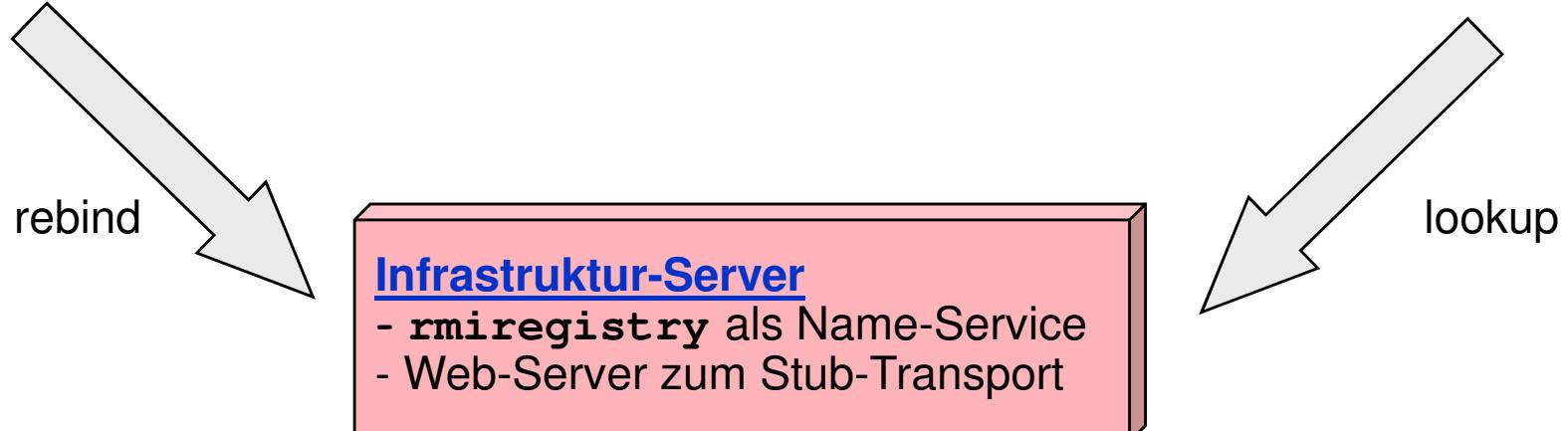
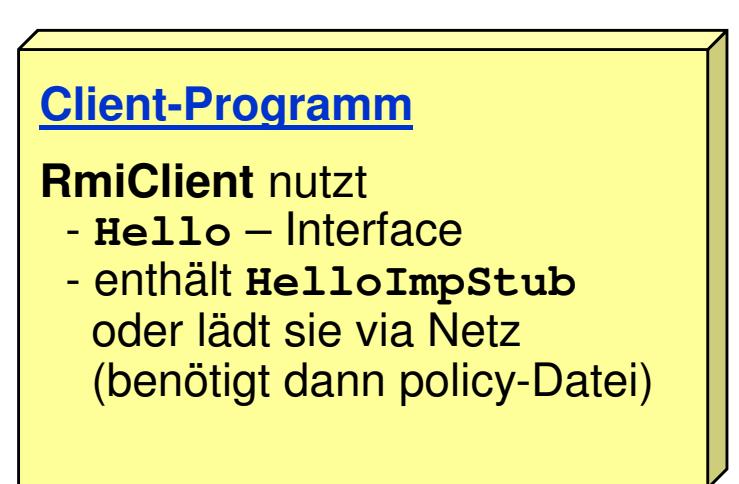
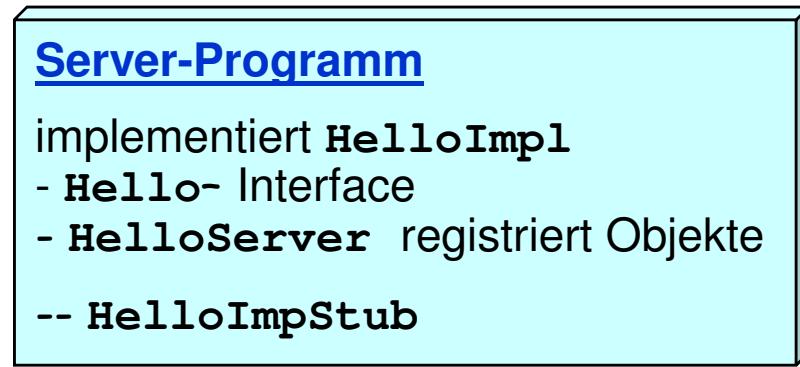
- Hinweis: dynamisches Laden von fehlenden Stub-Klassen → vorherige Folie



4.2.2 Deployment

Verteilung der Sourcen, Dienste und Prozesse

- **wichtig:** bei bidirektionaler Kommunikation ist eine Anwendung **sowohl Client als auch Server**
- d.h. es werden beide Rollen eingenommen (symmetrisches Client-Server-Modell)



Demo

RMI „Hello World“

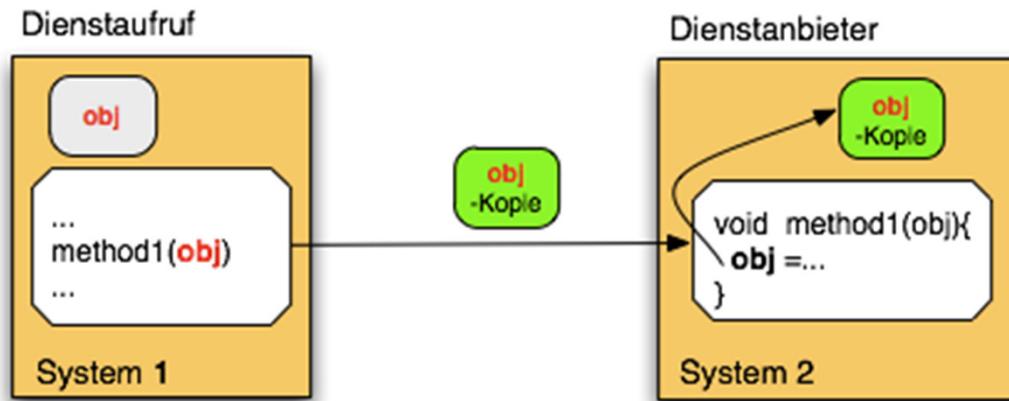
Weiteres Code-Beispiel „Chat mit RMI“ → Anhang



4.2.3 Parameterübergabe

Wert-Semantik und Wertübergabe in RMI

- es werden Kopien der Objekte zwischen Client und Server verschickt
- **Vorteil:** die Anzahl der remote Zugriffe ist reduziert
- **Nachteil:** Änderungen von entfernten Objekten sind so nicht möglich



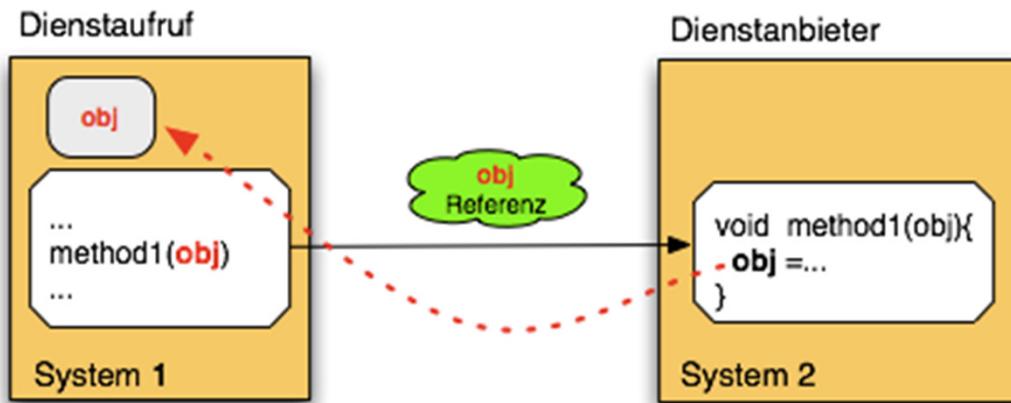
- die Parameter-Klassen implementieren **Serializable**
- `Account createAccount (kunde)` erhält bspw. nur die *Daten* des `kunde`-Objekts

```
class Kunde implements Serializable {...}  
class Account implements Serializable {...}
```

4.2.3 Parameterübergabe

Referenz-Semantik und Referenzübergabe in RMI

- Referenz auf das Parameter-Objekt wird an den Server übertragen
 - **Vorteil:** Server kann übergebene Client-Objekte auf Client ändern
 - **Nachteil:** Remote-Zugriffe sind teuer



- die Parameter-Klassen sind Unterklassen von `Remote`, d.h. sie haben ein Remote-Interface und sind von `UnicastRemoteObject` abgeleitet
 - die Server-Methode `createAccount(kunde)` ändert das `kunde`-Objekt auf Client

```
interface Kunde extends Remote {...}  
class KundeImpl extends UnicastRemoteObject implements Kunde {...}  
  
interface Account extends Remote {...}  
class AccountImpl extends UnicastRemoteObject implements Account {...}
```

4.2.4 CORBA (Common Object Request Broker Architecture)

CORBA im Vergleich zu RMI

- von der OMG (Object Management Group) 1989 spezifizierte Middleware
- verschiedene Hersteller/Implementierungen:
 - Microfocus (Ex Iona): Orbix, Orbacus & Visibroker (Ex borland), Java JDK ORB, TAO, . . .
- Heute zwar noch vielfach in Bestandssystemen im Einsatz aber nur noch wenig Neuentwicklung mit CORBA.

Gemeinsamkeiten zu RMI

- Softwarearchitektur für verteilte Objekte
 - entfernte Methodenaufrufe und Referenzen
 - Remote-Refrenzen
 - Registrierungsdienst (=Object Request Broker)

Unterschiede zu RMI

- verteilte Objekte in heterogenen Systemen
 - unterschiedliche Programmiersprachen (Java, C/C++, Smalltalk, Ada, ...)
- vielfach weitergehende Dienste (Quality of Service)
 - Sicherheit, Transaktionen,.Events, ..

4.2.4 CORBA

CORBA IDL (Interface Definition Language)

- **standardisierte Beschreibungssprache für Schnittstellen**
 - an C++ Klassendeklarationen orientiert, aber programmiersprachunabhängig
- **Sprach-Mapping:** Abbildung von IDL-Spezifikation auf Implementierung in Zielsprache
 - in/out/inout-Parameter
 - aus IDL-Beschreibung werden mit IDL-compiler die Stub-Klassen generiert

```
module Chat {  
    exception Reject {string reason;};  
    interface ChatServer {  
        void login (in Name name,in ChatClient chatter) raises (Reject);  
        void logout (in Name name);  
        void send (in Message message);  
    };  
};
```

Bem.: IDL wird auch unabhängig von CORBA genutzt, Bsp.: XML DOM

Sockets, Java RMI - Fragenblock

Diskutieren Sie 3 Minuten die folgende Frage mit
Ihrem Nachbarn bzw. Ihrer Nachbarin:

Vergleichen Sie die Eigenschaften von
Sockets & Java RMI anhand einiger
Kriterien, wie z.B. Performanz



4.3 Zusammenfassung

- **Sockets**
 - Interprozess-Kommunikation und Sockets
 - Beispiel: Chat-System mit Sockets
- **Remote Method Invocation**
 - Programmiermodell
 - Deployment
 - Parameterübergabe
 - Beispiel: „Hello World“ per RMI
 - Vergleich mit CORBA
- **Weitere RPC-Beispiele**
 - Direkte SOAP-Aufrufe → WS-*
 - gRPC (Google RPC) → eigene Recherche
 - ...

4.3 Zusammenfassung

Referenzen

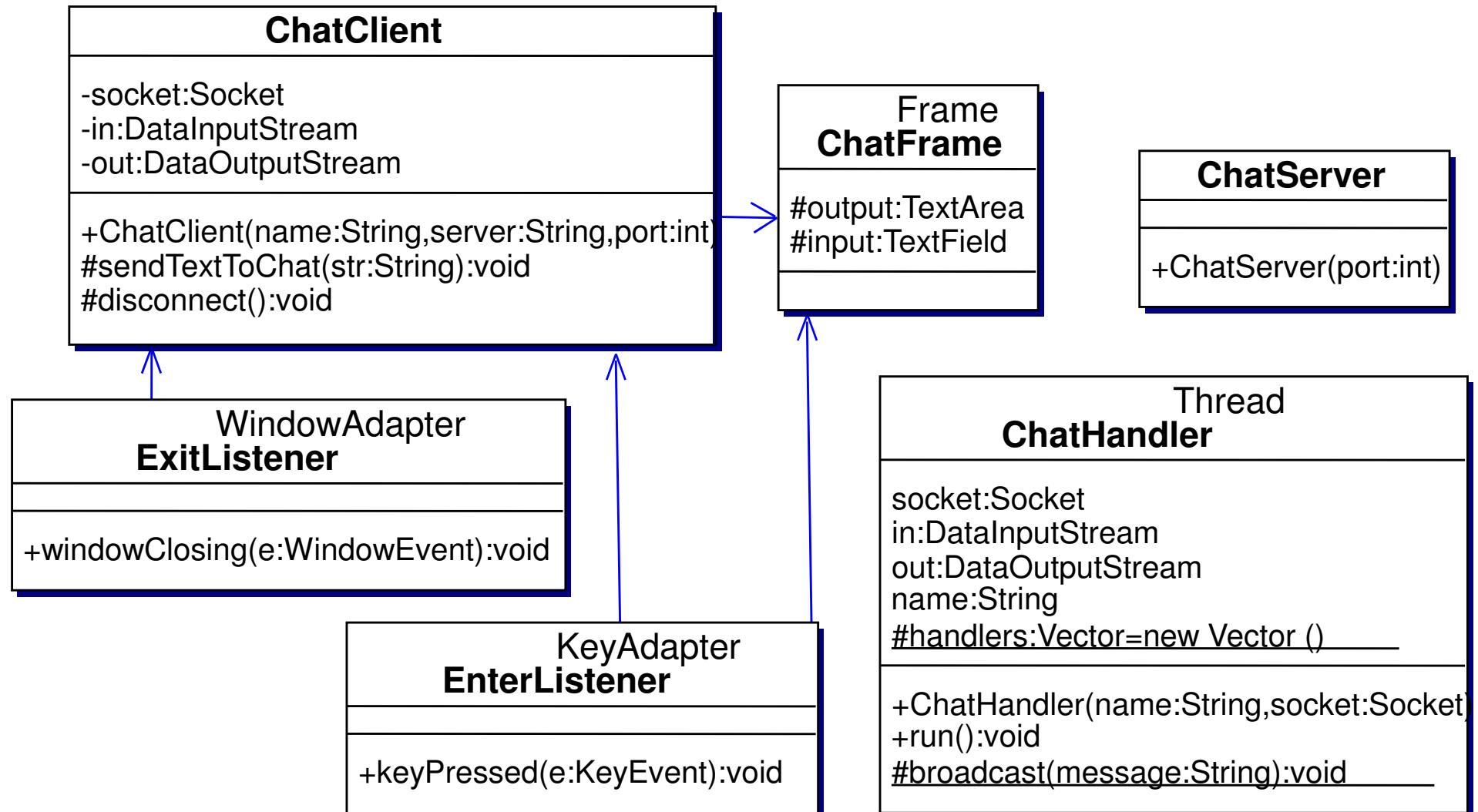
- M. Boger: Java in verteilten Systemen, dpunkt.verlag, 1999
- G. Bengel: Grundkurs Verteilte Systeme, vieweg, 3. Auflage, 2004
- D. Abts: Aufbaukurs Java: Client/Server-Programmierung mit JDBC, Sockets, XML-RPC und RMI, vieweg, 2003
- Rainer Oechsle: Parallele und verteilte Anwendungen in Java, Hanser, 2. Aufl., 2007.
- Jürgen Dunkel, Andreas Eberhart, Stefan Fischer, Carsten Kleiner, Arne Koschel: Systemarchitekturen für Verteilte Anwendungen - Client-Server, Multi-Tier, SOA, EDA, Grid, P2P, ..., Hanser, 2008.
- C.S. Horstmann, G. Corell: Core Java 2 – Volume 2 Advanced Features, 7. Edt, Prentice Hall 2005
- D. Lea: Concurrent Programming in Java, Design Principles and Patterns, Addison Wesley, 1997

Oracle/Sun-Dokumentation



ANHANG A – Ergänzung Beispiele (Chat mit Sockets & RMI)

4.1.2 Beispiel: Chat-System – Klassendiagramm



4.1.2 Beispiel: Chat-System

- **Chat-Server:**
 - erzeugt Server Socket
 - wartet auf Client-Anmeldung
 - erzeugt ClientSocket
 - delegiert Kommunikation mit Client an Handler-Objekt
(Server/Handler-Muster: eigener Thread für jeden neuen Client)
- **Chat-Handler:**
 - verwaltet statischen Vector mit allen Client-Handlern (für Broadcast an alle Clients)
 - erzeugt BufferedReader und PrintWriter für ClientSocket
 - liest Nachricht und führt Broadcast durch
- **Chat-Client(s):**
 - baut Verbindung zum Server auf
 - schickt und empfängt Nachrichten an Server
 - besitzt GUI: Eingabe neuer Nachrichten u. empfangene Nachrichten anzeigen



4.1.2 Beispiel: Chat-System – Server

```
import java.io.BufferedReader;  
  
public class ChatServer {  
    public ChatServer(int port) {  
        try {  
            ServerSocket server = new ServerSocket(port);  
  
            while (true) {  
                // akzeptiere Requests - erzeuge Clientsocket  
                Socket clientSocket = server.accept();  
  
                // lese Namen des Client  
                BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));  
                String cName = in.readLine();  
                System.out.println("new client " + cName + " from " + clientSocket.getInetAddress());  
  
                //erzeuge Handler-Objekt fuer jeden neuen Client und starte  
                ChatHandler c = new ChatHandler(cName, clientSocket);  
                c.start();  
            }  
        } catch (IOException e) {e.printStackTrace();}  
    }  
  
    public static void main(String args[]) throws IOException {  
        int port = 1234; new ChatServer(port);  
    }  
}
```

4.1.2 Beispiel: Chat-System – Handler (1)

```
+import java.io.BufferedReader;[]

public class ChatHandler extends Thread {
    Socket socket;
    String name;
    BufferedReader in;
    PrintWriter out;
    protected static Vector handlers = new Vector(); // statisch: alle Handler fuer broadcast

    public ChatHandler(String name, Socket clientSocket) throws IOException {
        this.name = name;      this.socket = clientSocket;
        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        out = new PrintWriter(clientSocket.getOutputStream(), true);
    }

    public void run() {}

    protected static void broadcast(String message) {}
```

4.1.2 Beispiel: Chat-System – Handler (2)

```
public void run() {  
    try {  
        broadcast(name + " entered"); // fuehrt broadcast fuer Anmeldung durch  
        handlers.addElement(this); // fuegt Handler hinzu  
        while (true) {  
            String message = in.readLine(); // liest Eingabe von seinem Client (=name)  
            if (message == null) {  
                break;  
            }  
            broadcast(name + ": " + message); // broadcast der Eingabe  
            System.out.println(name + ": " + message);  
        }  
    } catch (IOException ex) { ex.printStackTrace();}  
    finally { // Thread beendet  
        handlers.removeElement(this);  
        broadcast(name + " left");  
        try {  
            socket.close(); } catch (IOException ex) { ex.printStackTrace(); }  
    }  
}
```

4.1.2 Beispiel: Chat-System – Handler (3)

```
protected static void broadcast(String message) {  
    synchronized (handlers) {  
        Enumeration e = handlers.elements();  
        while (e.hasMoreElements()) {  
            ChatHandler handler = (ChatHandler) e.nextElement();  
            //TODO remove try catch from slide  
            handler.out.println(message);  
            handler.out.flush();  
        }  
    }  
}
```



4.1.2 Beispiel: Chat-System – Client (1)

```
public class ChatClient {  
  
    public ChatFrame gui;      private Socket socket2Server;  
    private BufferedReader in; private PrintWriter out;  
  
    public ChatClient(String name, String server, int port) {  
        // GUI erzeugen  
        gui = new ChatFrame(name + " - Chat mit Sockets");  
        gui.input.addKeyListener(new EnterListener(this, gui));  
        gui.addWindowListener(new ExitListener(this));  
  
        // Socket erzeugen, am Server anmelden und lauschen  
        try {  
            socket2Server = new Socket(server, port);  
            in = new BufferedReader(new InputStreamReader(socket2Server.getInputStream()));  
            out = new PrintWriter(socket2Server.getOutputStream(), true);  
            // anmelden des neuen Client bei Server  
            out.println(name);  
            // wartet auf Eingaben anderer Chat-Clients  
            while (true) {  
                gui.output.append("\n" + in.readLine());  
            }  
        } catch (Exception e) { e.printStackTrace(); }  
    }  
}
```

4.1.2 Beispiel: Chat-System – Client (2)

```
// wird von KeyListener aufgerufen
protected void sendTextToChat(String str) {
    out.println(str);
}

// wird von WindowListener aufgerufen
protected void disconnect() {
    try {
        socket2Server.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String args[]) throws IOException {
    // Aufruf: java ChatClient daisy localhost 4711
    String name = args[0];  String host = args[1];
    int port = Integer.parseInt(args[2]);
    ChatClient c = new ChatClient(name, host, port);
}
```

4.1.2 Beispiel: Chat-System – GUI

```
import java.awt.BorderLayout;  
  
public class ChatFrame extends Frame {  
    protected TextArea output;  
  
    protected TextField input;  
  
    public ChatFrame(String title) {  
        super(title);  
        setLayout(new BorderLayout());  
        add("Center", output = new TextArea());  
        output.setEditable(false);  
        add("South", input = new TextField());  
        pack();  
        show();  
        input.requestFocus();  
    }  
}
```



4.1.2 Beispiel: Chat-System – GUI

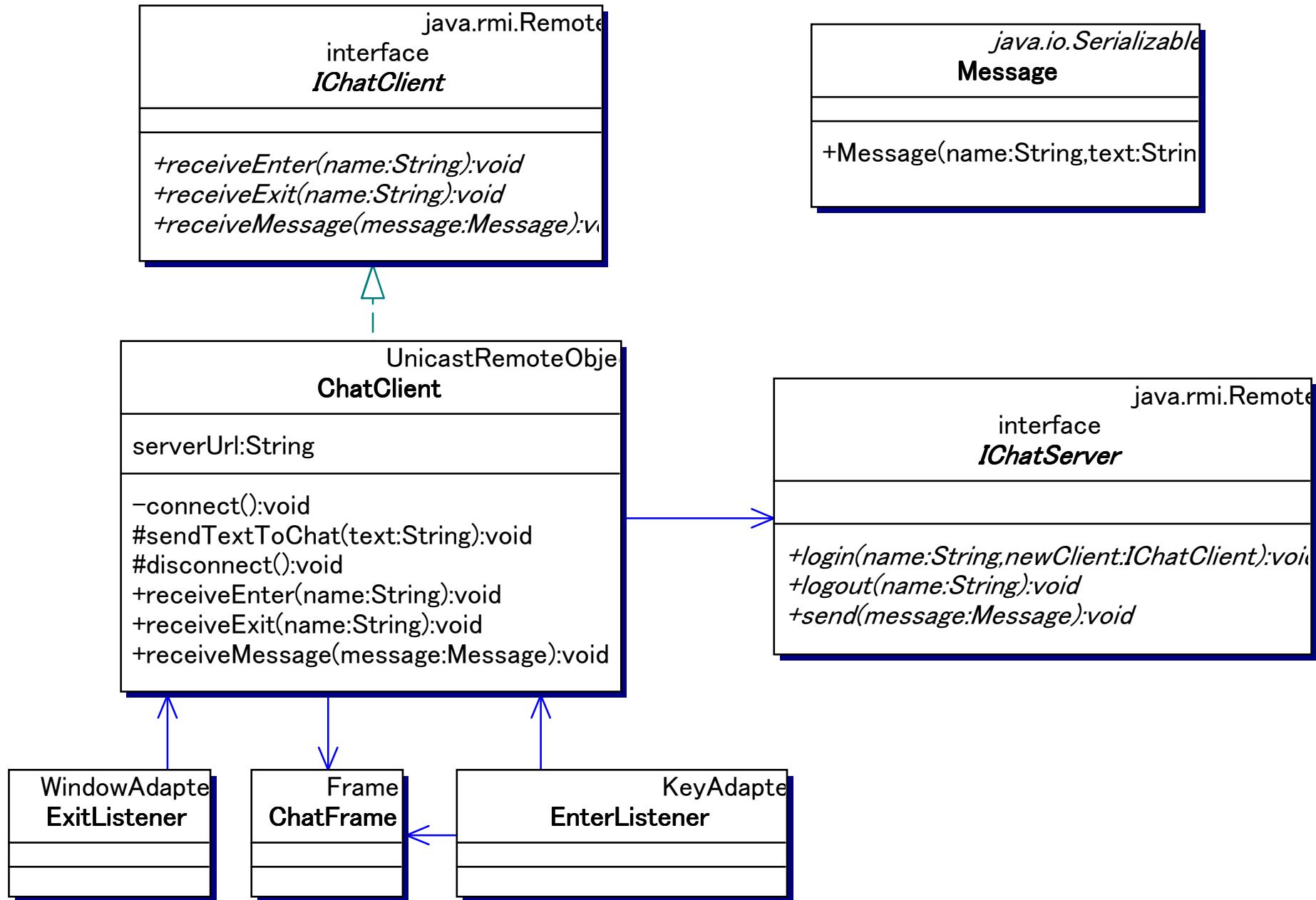
```
import java.awt.event.KeyAdapter;  
  
public class EnterListener extends KeyAdapter {  
    ChatClient client;  
    ChatFrame gui;  
  
    public EnterListener(ChatClient client, ChatFrame gui) {  
        this.client = client; this.gui = gui;  
    }  
  
    public void keyPressed(KeyEvent e) {  
        if (e.getKeyCode() == KeyEvent.VK_ENTER) { // sende eingegebenen Text an Client  
            client.sendTextToChat(gui.input.getText());  
            gui.input.setText(""); // leere Eingabefeld in gui  
        }  
    }  
}
```

4.1.2 Beispiel: Chat-System – GUI

```
import java.awt.event.WindowAdapter;  
  
public class ExitListener extends WindowAdapter {  
    ChatClient client;  
  
    public ExitListener(ChatClient client) {  
        this.client = client;  
    }  
  
    public void windowClosing(WindowEvent e) {  
        // schliesse all Sockets  
        client.disconnect();  
        System.exit(0);  
    }  
}
```

Anhang B – Chat mit RMI

4.2.4 Beispiel: Chat per RMI



4.2.4 Beispiel: Chat per RMI

- **Chat-Server**

- Hash-Tabelle mit Remote-Referenzen auf alle ChatClients
- Schnittstelle zum Anmelden und Senden von Nachrichten
- meldet sich in Registry an

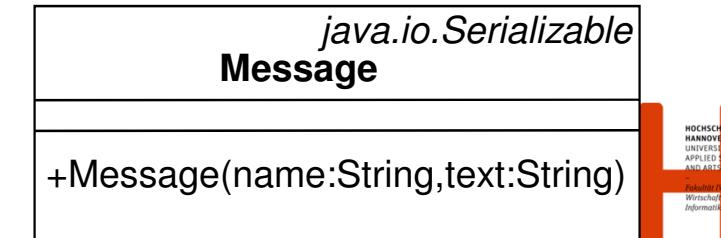
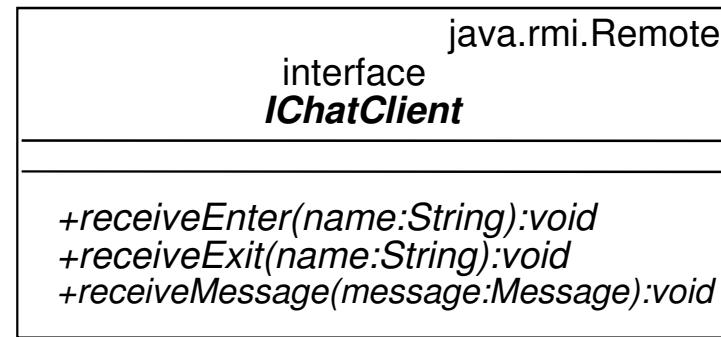
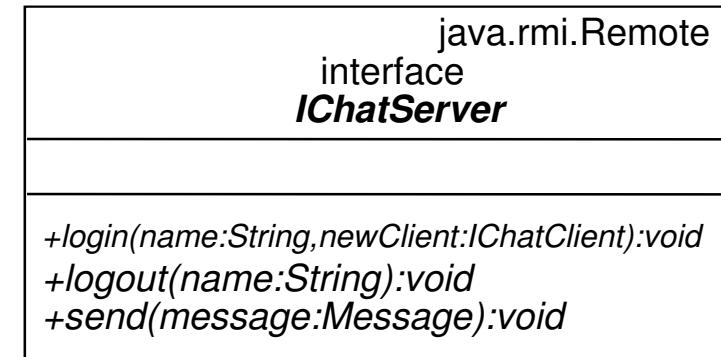
- **Chat-Client**

- baut Referenz zum Server auf und nutzt dessen Dienste
- übergibt beim *Login* eine Referenz auf sich
- sendTextToChat(Message) schickt Nachricht an Server, der die Nachricht weiterleitet
- receive(Message) empfängt Nachricht

- **Message**

- enthalten Absender + Inhalt
- ist Datencontainer, d.h. serialisierbar

Hinweis: gesamte Kommunikation erfolgt über remote-Objekte



4.2.4 Beispiel: Chat per RMI

```
public class ChatServer extends UnicastRemoteObject implements IChatServer {  
  
    Hashtable chatters = new Hashtable();  
    public ChatServer() throws RemoteException { }  
  
    public synchronized void login(String name, IChatClient newClient) throws RemoteException {  
        Enumeration chatClients = chatters.elements();  
        chatters.put(name, newClient);  
        // login allen Clients bekannt geben  
        while (chatClients.hasMoreElements()) {  
            ((IChatClient) chatClients.nextElement()).receiveEnter(name);  
        }  
        System.out.println("new client " + name + " is logged in");  
    }  
  
    public synchronized void logout(String name) throws RemoteException {  
        chatters.remove(name);  
        Enumeration chatClients = chatters.elements();  
        // logout allen Clients bekannt geben  
        while (chatClients.hasMoreElements()) {  
            ((IChatClient) chatClients.nextElement()).receiveExit(name);  
        }  
    }  
}
```

4.2.4 Beispiel: Chat per RMI

```
public synchronized void send(Message message) throws RemoteException {  
    Enumeration chatClients = chatters.elements();  
    while (chatClients.hasMoreElements()) {  
        ((IChatClient) chatClients.nextElement()).receiveMessage(message);  
    }  
}  
  
public static void main(String[] args) {  
    try {  
        ChatServer server = new ChatServer();  
        Naming.rebind("ChatServer", server);  
    } catch (Exception ex) { ex.printStackTrace(); }  
}
```

4.2.4 Beispiel: Chat per RMI

```
public class ChatClient extends UnicastRemoteObject implements IChatClient{  
    ChatFrame gui;  String name;  IChatServer server;  String serverUrl;  
  
    public ChatClient(String name, String url) throws RemoteException {  
        this.name = name;  serverUrl = url;  
        gui = new ChatFrame("Chat mit RMI");  
        gui.input.addKeyListener (new EnterListener(this,gui));  
        gui.addWindowListener(new ExitListener(this));  
        connect();  
    }  
  
    private void connect() {  
        try {  
            server = (IChatServer) java.rmi.Naming.lookup("rmi://" + serverUrl + "/ChatServer");  
            server.login(name, this);  
        } catch (Exception e) { e.printStackTrace(); }  
    }  
  
    protected void sendTextToChat(String text) {  
        try {  
            server.send(new Message(name, text));  
        } catch (RemoteException e) { e.printStackTrace(); }  
    }  
}
```

4.2.4 Beispiel: Chat per RMI

```
protected void disconnect() {  
    try {  
        server.logout(name);  
    } catch (Exception e) { e.printStackTrace(); }  
}  
  
public void receiveEnter(String name) {  
    gui.output.append(name+" entered \n");  
}  
  
public void receiveExit(String name) {  
    gui.output.append(name+" left \n");  
}  
  
public void receiveMessage(Message message) {  
    gui.output.append(message.name+": "+message.text+"\n");  
}  
  
public static void main(String[] args) throws Exception {  
    new ChatClient(args[0],args[1]);  
}
```



4.2.4 Beispiel: Chat per RMI

```
public class Message implements java.io.Serializable {  
    public String name;  
    public String text;  
  
    public Message(String name, String text) {  
        this.name = name; this.text = text;  
    }  
}
```

Verteilung der Sourcen

