Zweites RPC-Beispiel (Stub / Skeleton)

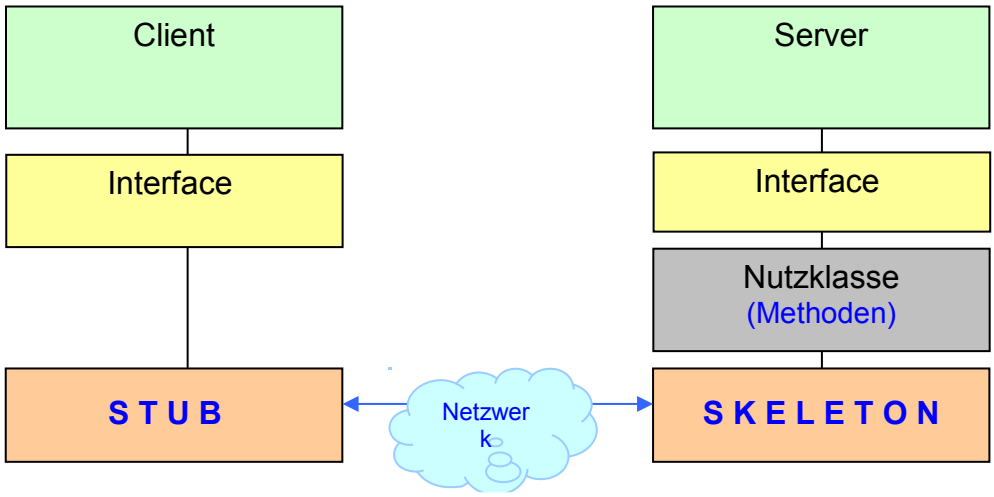
Im folgenden Beispiel wird gezeigt, wie die Bestandteile einer RPC-Anwendung modularisiert werden können, um eine grösstmögliche Abstraktion und Allgemeingültigkeit zu erreichen. Das hier vorgestellte Arbeitsprinzip mit Stubs und Skeletons erlaubt es, umfassende RPC-Programme überschaubar und kontrollierbar zu entwickeln. Der Aufruf entfernter Methoden mit Stub-Skeleton-Klassen erlaubt sogar die Nutzung von entfernten Objekten (Remote-Objekten). Deren Nutzbarkeit ist aber eingeschränkt, weil der Client keine Möglichkeit, sich das individuelle Remote-Objekt auszusuchen, für das die Methoden ausgeführt werden. Es gibt keinen Objektvermittlungsdienst (Object Broker).

Folgende Massnahmen sind erforderlich:

1. Die client- und serverseitigen Sockets werden in eigenen Klassen (Stub / Skeleton) gekapselt, die alles enthalten, das zum Aufbau einer Verbindung nötig ist. Hier werden die benötigten *ObjectStreams* erzeugt, die zum Versenden der Methodenaufrufe und Empfang eventueller Rückgabewerte gebraucht werden. Stub und Skeleton enthalten aber noch mehr. Der Client-Stub enthält zwar formal alle Methoden, die beim Server-Nutz-Objekt definiert sind, allerdings ist ihr Anweisungsrumpf ein ganz anderer. Eine Stub-Methode tut nichts anderes, als den Methodenaufruf (über seine Socketverbindung) an das Skeleton des Servers weiter zu leiten. Das Skeleton benutzt sein serverseitiges Objekt (der Nutzklasse) um den empfangenen Methodenaufruf auszuführen. Der Ergebniswert wird dann über die Socketverbindung an den Client-Stub zurückgesendet. Der Client kann den Rückgabewert als Resultat seines eigenen Stub-Aufrufs verarbeiten.

2. Die Nutzklasse legt alle Eigenschaften und Methoden des Nutz-Objekts fest. Sie ist nur auf dem Server vorhanden.

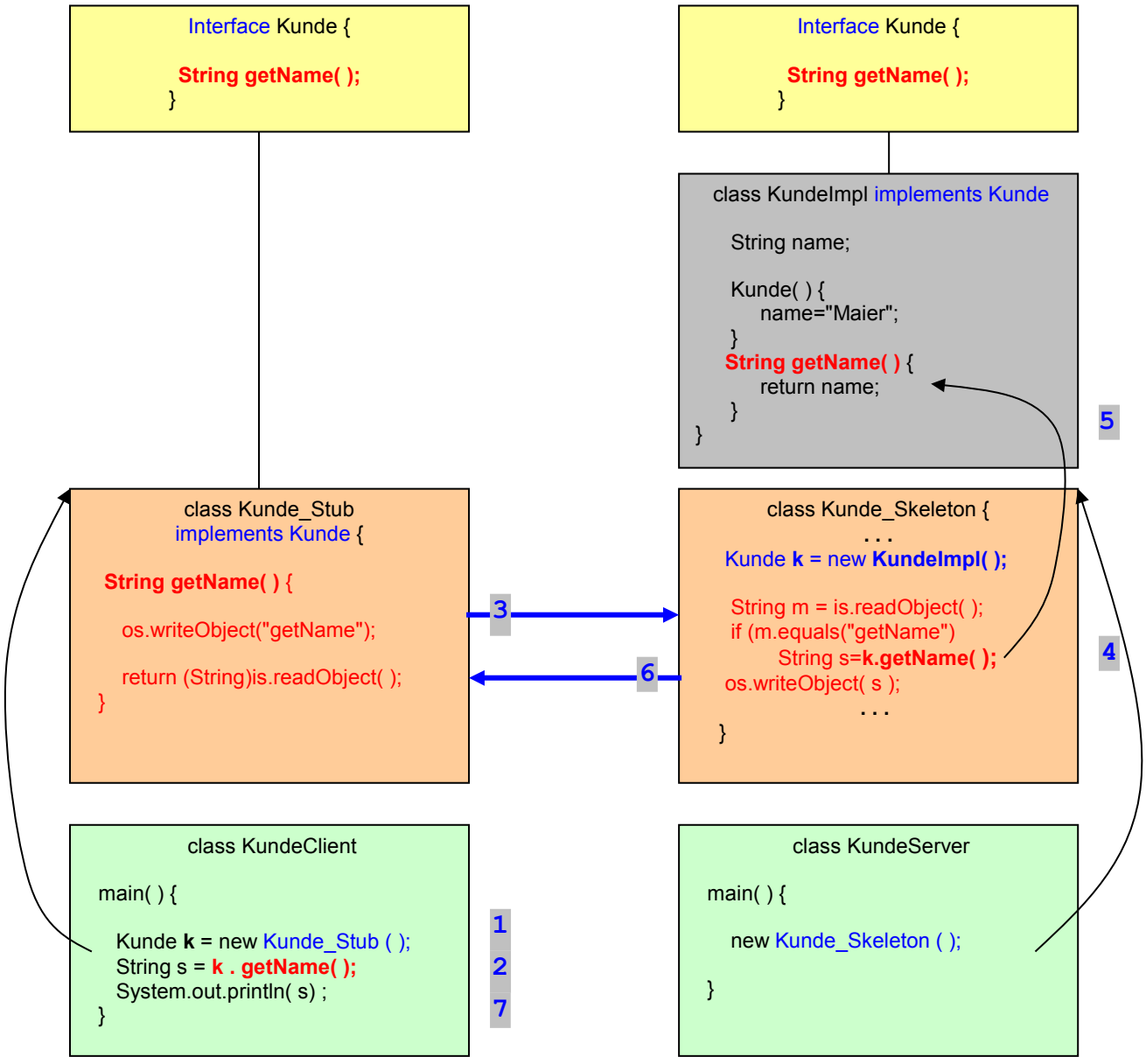
3. Auf beiden Seiten gibt es identische Interfaces, die alle Methoden der Server-Nutz-Objekts beschreiben. Stub- und Skeletonklasse müssen dieses Interface implementieren, damit die Identität der Methodensignaturen gesichert ist.



Schematischer Aufbau einer RPC-Anwendung
mit Stub, Skeleton und Interface

Ablauf eines RPC-Aufrufs

(1) Der Client erzeugt ein Ersatz-Objekt (stub-Objekt), das vom Typ seines Interfaces sein muss. (2) Er ruft eine Methode für dieses Stub-Objekt auf. (3) Darin wird der Methodenaufruf (per Socket-Streams) an das Skeleton des Servers übertragen. (4) Das Skeleton analysiert und prüft den empfangenen Aufruf und ruft dann (5) gegebenenfalls die entsprechende Methode seines Nutz-Objekts auf. (6) Den Rückgabewert sendet er an den Client zurück. (7) Dieser kann den Wert ausgeben.



RPC-Methodenaufruf mit Stub / Skeleton

Der Quellcode ist hier erhältlich (RPC_Example2.ZIP)

Client-Interface

```
////////////////////////////////////////////
// Kunde.java

interface Kunde {

     public String getName() throws Exception ;
}
```

Client-Stub

```
////////////////////////////////////////////
// Kunde_Stub.java

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

class Kunde_Stub implements Kunde{

   Socket server;

   Kunde_Stub () throws Exception {
     server = new Socket ( "", 9999 );
   }

   public String getName() throws Exception {

     ObjectOutputStream os = new ObjectOutputStream (
                                server.getOutputStream());
     ObjectInputStream is =  new ObjectInputStream (
                                server.getInputStream());
     os.writeObject("getName");
     os.flush();
     Object returnWert = is.readObject();
     return (String) returnWert;
  }
}
```

Client

```
////////////////////////////////////////////////////////
// KundeClient.java

import java.io.*;

public class KundeClient {

  public static void main (String args[])  {

   try {
     Kunde k = new Kunde_Stub();
     System.out.println("Kunde heisst "+k.getName());
     System.in.read();

   } catch( Exception e) {
     System.out.println( e);
   }
  }
}
```

Server-Interface

```
////////////////////////////////////////////
// Kunde.java
```

```
interface Kunde {

     public String getName() throws Exception ;
}
```

Server-Nutzklasse

```
//////////////////////////////////////////
// KundeImpl.java

class KundeImpl implements Kunde{

    String name;

    KundeImpl(String n) {

       name = n;
    }

    public String getName() {

       return this.name;
    }
}
```

Server-Skeleton

```
//////////////////////////////////////////////////////
// Kunde_Skeleton.java

import java.io.*;
import java.net.Socket;
import java.net.ServerSocket;

class Kunde_Skeleton {

   ServerSocket ss;
   Socket socket;
   ObjectOutputStream os;
   ObjectInputStream  is;
   Kunde k;

   Kunde_Skeleton ()  {

    try {
      k = new KundeImpl("Maier");  // Kundenobjekt erzeugen

      ss = new ServerSocket ( 9999 );
      System.out.println("Server erfolgreich gestartet!\n"+
                         "Warte auf Client... \n");
      socket = ss.accept();

      System.out.println( "Client angemeldet...");

      os = new ObjectOutputStream(socket.getOutputStream());
      is = new ObjectInputStream (socket.getInputStream());

      String methodenName = (String)is.readObject();

      String n = "Methode nicht gefunden";

      if ( methodenName.equals("getName") )
          n = k.getName();

      os.writeObject(n);
      os.flush();
```

```
            os.close();
            is.close();
          } catch (Exception e) {
             System.out.println(e);
          }
        }
    }
```

Server

```
    ////////////////////////////////////
    // KundeServer.java

    class KundeServer {

        public static void main (String args[]) {

            new Kunde_Skeleton ( );

        }
    }
```