



CORBA mit Java

Verteilter Kaffee



Java kommt **aus dem Netz** und kann über das Netz verteilt werden. **Der Einstieg** in die CORBA-Programmierung **unter Java** ist keine teure Angelegenheit. Der freie **JacORB** gibt Ihnen die Gelegenheit direkt **ohne weitere Kosten** einzusteigen.

OLIVER MÜLLER

Auch unter Java müssen Sie nicht auf CORBA verzichten. Ähnlich wie MICO für C++ liefern wir Ihnen auch für Java einen freien ORB. Er heißt JacORB und Sie können ihn einsetzen, ohne Lizenzgebühren zu entrichten.

■ JacORB installieren

Um JacORB verwenden zu können, benötigen Sie JDK 1.1 oder später. Auf der Heft-CD finden Sie das Java 2 SDK 1.2.2 für Windows und Linux, mit dem JacORB eingesetzt werden kann.

JacORB müssen Sie nicht selbst kompilieren. Die Archive für Windows und Linux auf der Heft-CD enthalten bereits vorkompilierte Packages. Sollten Sie es dennoch in Erwägung ziehen den

ORB neu zu kompilieren, müssen Sie Ant installieren. Dieses Tools hält die CD-ROM ebenfalls für Sie bereit. Nähere Information zur Kompilierung von JacORB finden Sie in der Dokumentation.

Zunächst entpacken Sie das Archiv *JacORB1_2_2-full.tar.gz* (Linux) bzw. *JacORB1_2_3-full.zip* (Windows).

Unter Linux legen Sie die CD-ROM in das Laufwerk ein und führen folgende Kommandos auf der Shell als "root" aus:

```
mount /mnt/cdrom
cd /usr/local
tar xzf /mnt/cdrom/JacORB/
JacORB1_2_2-full.tar.gz
umount /mnt/cdrom
```

TIPP Wenn Sie mit SuSE Linux arbeiten, setzen Sie bei "mount" und "umount" statt */mnt/cdrom* die Angabe */cdrom* ein.

Unter Windows entpacken Sie das Archiv *JacORB1_2_3-full.zip* aus dem Verzeichnis *JacORB* der CD-ROM mit einem ZIP-Programm, wie PKZIP oder WinZip. Der folgende Text geht davon aus, dass Sie das Archiv in den Pfad *C:* entpacken. (Es entsteht das Verzeichnis *C:\JacORB-1_2_3*.)

Jetzt ist JacORB auf Ihrer Festplatte und Sie können sich der Konfiguration des Systems zuwenden. Als ersten Schritt bringen Sie Ihrem System bei, wo die JacORB-Tools schlummern. Unter Linux bearbeiten Sie in die Datei

/etc/profile und fügen folgenden Befehl an:

```
export PATH=$PATH:/usr/local/
JacORB-1_2_2/bin
```

Unter Windows bearbeiten Sie die Datei *C:\AUTOEXEC.BAT*. Fügen Sie das Kommando

```
set PATH=%PATH%;C:\JacORB-1_2_3\bin
```

an.

Voraussetzung ist, dass Sie das ZIP-Archiv auf *C:* entpackt haben, andernfalls passen Sie den Pfad entsprechend an.

TIPP Unter Windows NT/2000 verwenden Sie zum Anpassen der PATH-Variablen die Systemsteuerung. Über das Symbol *System* erhalten Sie den Dialog *Systemeigenschaften*. Klicken Sie darin auf den Reiter *Umgebung* und passen Sie die PATH-Variable an.

Jetzt müssen Sie die Datei *orb.properties* aus dem JacORB-Verzeichnis in das lib-Verzeichnis Ihrer Java/JDK-Installation kopieren. In dieser kopierten Datei sind noch ein paar Einstellungen auf Ihren Webserver abzustimmen. Die Zeilen aus dem Textkasten "Die Webserver-Einträge" geben die Einstellungen aus *orb.properties* wieder. Passen Sie diese auf die URLs bzw. lokalen Pfade Ihres Webserver an.

Die nächste Aufgabe ist das Bearbei-

QUICK INDEX

- ▶ **JacORB installieren**
Die Installation von JacORB bedarf einiger Konfiguration.
- ▶ **Minibank**
Als CORBA-Beispiel dient eine Miniaturbank.
- ▶ **Servieren**
Der POA-Server in Java verhält sich wie die C++-Variante.
- ▶ **Verwenden**
Auch der Client in Java funktioniert sein C++-Pendant.

DIE WEBSERVER-EINTRÄGE

```
jacorb.NameServerURL=http://ww
w.inf.fu-berlin.de/~brose/NS_Ref
jacorb.TradingServiceURL=http://w
ww.inf.fu-berlin.de/~brose/TS_Ref
jacorb.ImplementationRepository
URL=http://www.inf.fu-berlin.de/
~brose/ImR_Refjacorb.ProxyServer
URL=http://www.inf.fu-berlin.de/
~brose/Applicator_Ref
```



ten von zwei Scripts bzw. Batch-Dateien im bin-Verzeichnis Ihrer JacORB-Installation. Die Dateien `idl` und `jaco` bzw. `IDL.BAT` und `JACO.BAT` enthalten Verweise auf den JacORB-Pfad. Sie müssen in diesen beiden Dateien die Angabe `/export/brose` im Installationspfad von JacORB ändern: `/usr/local/JacORB-1_2_2` (Linux) bzw. `C:\JacORB-1_2_3` (Windows).

Die beiden Programme `idl` und `jaco` sind die wichtigsten Tools von JacORB. `idl` ist der IDL-Compiler, der im nächsten Abschnitt besprochen wird. `jaco` ist ein Wrapper-Script, das zum Ausführen von JacORB-Java-Programmen dient.

Normalerweise starten Sie ein Java-Programm durch den Java-Interpreter `java`. Angenommen die Hauptklasse heißt `ServerMain`, dann starten Sie das Programm durch das Kommando

```
java ServerMain
```

Wird in `ServerMain` jedoch JacORB verwendet, müssten Sie entweder den `CLASSPATH` anpassen, oder über `-cp` diesen beim Aufruf des Interpreters erweitern. Diese Aufgaben übernimmt das `jaco`-Script.

Wenn `ServerMain` JacORB verwendet, starten Sie das Programm einfach durch

```
jaco ServerMain
```

■ Minibank

Als Einführung in die Welt der CORBA-Programmierung in Java, haben wir ein kleines Beispiel auf der CD-ROM zurechtgelegt. Im Verzeichnis `Beispiele/CORBA-Java` finden Sie je ein Archiv für Linux (`tar/gzip`) und Windows (ZIP). Darin ist das kleine Bankbeispiel aus dem C++/CORBA-Beitrag als Java-Programm enthalten.

Die Implementierung setzt komplett auf den POA. Eine BOA-Version existiert nicht.

In Listing 1 sehen Sie die IDL-Definition der Klasse `Konto`. Dieses IDL ist identisch mit dem aus dem C++/CORBA-Beitrag. Dies ist nicht zufällig, da wir Ihnen die Chance geben wollen, dass Sie die Ergebnisse beider Beiträge kombinieren.

Sie können das POA-Beispiel von C++ durchaus mit dem Java-Beispiel kombinieren. Starten Sie später einmal einen C++-Server und bauen Sie eine Verbindung mit dem Java-Client auf und vice versa.

TIPP Welche Operationen die Klasse `Konto` bzw. deren Objekte bieten, entnehmen Sie dem C++/CORBA-Beitrag.

Das IDL-File wird in JacORB durch den Befehl

```
idl Konto.idl
```

übersetzt. Sie müssen nicht wie bei MICO explizit angeben, dass POA verwendet werden soll. JacORB arbeitet per Default mit POA.

Der IDL-Compiler generiert eine ganze Reihe von Klassen und außerdem Quellcodes:

- `Konto.java`
- `KontoHelper.java`
- `KontoHolder.java`
- `KontoOperations.java`
- `KontoPOA.java`
- `KontoPOATie.java`
- `_KontoStub.java`

JACORB IM INTERNET

JacORB selbst und jede Menge Informationen rund um den freien ORB finden Sie im Internet unter <http://jacorb.inf.fu-berlin.de>.

Wichtig sind hier vor allem die Klassen/Interfaces `Konto` (Basisklasse), `KontoOperations` (Interface der IDL-Methoden), `KontoPOA` (Skeleton) und `_KontoStub` (Stub).

Die Implementierung der CORBA-Klasse findet sich in der Klasse `KontoImpl`. Sie ist vom Skeleton abgeleitet. Indirekt über das Skeleton implementiert sie das Interface `KontoOperations`.

TIPP In `KontoOperations.java` finden Sie die vom IDL-Compiler in Java-Methoden-Interfaces umgesetzte IDL-Definition. Kopieren Sie sich diese Methoden aus `XxxxOperations.java` bei eigenen Projekten in die Klassendefinition von `XxxxImpl.java`, um sie zu implementieren.

Die Implementierung der Methoden ist pure in Java-Code umgesetzte Arithmetik.

■ Servieren

Die Klasse `KontoServer` implementiert den CORBA-Server. Hierzu ist nicht mehr als eine `main()`-Methode notwendig. Die ersten beiden Anweisungen, setzen die Eigenschaft `jacorb.verbosity`

auf null. Ansonsten würde JacORB seine Protokolldateien mit Meldungen zuschütten.

TIPP Über die Werte 1 und 2 können Sie die Mitteilungsfrequenz von JacORB wieder erhöhen.

Als erstes wird der ORB initialisiert. Hierzu wird die Methode `init()` der Klasse `ORB` aufgerufen. Dem ORB werden wie im C++-Beispiel die Kommandozeilenargumente (`args`) mit auf den Weg gegeben. Auf diese Weise kann der ORB etwaige übergebene Einstellungen auswerten. Außerdem werden die zuvor geänderten Properties übergeben.

Jetzt wird der POA initialisiert. Der POA ist selbst schon ein CORBA-Objekt und muss als solches über einen Stub instanziiert werden. Die Helper-Klasse dient in Java dazu über einen Stub ein CORBA-Objekt per `narrow()` zu holen. Anschließend wird der POA-Manager geholt und aktiviert.

Sie sehen, dass die Initialisierung des POA in Java im Wesentlichen nicht anders funktioniert als in C++. Durch die etwas andere Arbeitsweise von Java sind noch Helper-Klassen und Properties notwendig, aber das Prinzip ist das gleiche. Zuerst wird der ORB initialisiert, dann eine Objektreferenz für den POA aufgelöst, um dann den POA-Manager zu erhalten.

Die Konto-Objekte `konto1` bis `konto3` werden über die Klasse `KontoImpl` instanziiert. Das ist nicht überraschend. Die Konto-Instanzen im Server müssen voll funktionsfähig sein, die Methoden implementiert sein. Diese Voraussetzung erfüllt nur `KontoImpl`.

Nach der Instanzierung des jeweiligen Konto-Objekts, wird über `poa.servant_to_reference()` eine Objektreferenz generiert, die das Konto-Objekt innerhalb des ORB eindeutig identifiziert. Diese Objektreferenz wird über `orb.object_to_string()` in einen String gewandelt, der für jedes Konto wieder in eine separate Datei geschrieben wird.

LISTING 1

```
interface Konto {
    void gutschreiben(in unsigned long betrag);
    boolean abbuchen(in unsigned long betrag);
    long getKontostand();

    unsigned long getDispokredit();
    void setDispokredit(in unsigned long betrag);

    unsigned long getKontoNr();
};
```



Der Java-KontoServer verhält sich damit absolut konform zum C++-POP-Server. Da Sie zum Ansprechen eines CORBA-Objekts nur diesen Referenz-String benötigen, können Sie leicht aus dem C++-Client die CORBA-Objekte im Java-Server ansprechen.

Zuletzt muss noch der ORB gestartet werden. Auch das funktioniert wieder in ähnlicher Weise wie bei C++. Ein Aufruf von `orb.run()` genügt, um den Server verbindungsbereit zu machen.

■ Verwenden

Die erste Aufgabe im Client ist wieder die ORB-Initialisierung. Hier werden keine Eigenschaften übergeben und somit die Defaults von JacORB verwendet.

Ein einfacher Null-Pointer als Argument genügt hierfür. Um die CORBA-Objekte anzusprechen, werden über `orb.string_to_object()` die Referenz-Strings der drei Konto-Instanzen in Objektreferenzen gewandelt. Die Referenz-Strings werden dabei

wie im C++-Beispiel über eine lokale file:-URL angesprochen. Nachdem die Objektreferenzen existieren, können diese, wie zuvor bei der Instanziierung von POA, über die Helper-Klasse in Objekte gekehrt werden. Bei den Konto-Objekten verwenden Sie zu diesem Zweck dann die Methode `narrow()` der Klasse *KontoHelper*.



Wenn Sie Server und Client nicht auf dem gleichen Host im Netzwerk laufen lassen, müssen Sie vor dem Start des Clients die Referenzdateien kopieren.

LISTING 2:

```
import java.io.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;

public class KontoServer {
    static String REFFILE = "/tmp/Konto";

    public static void main(String[] args) {
        try {
            // Alle Ausgaben von JacORB
            // ausschalten
            java.util.Properties props = new java.util.Properties();
            props.put("jacorb.verbosity", "0");

            // ORB initialisieren
            ORB orb = ORB.init(args, props);
            POA poa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
            poa.the_POAManager().activate();

            // Server-Objekte erzeugen und
            // Referenz-Strings in Dateien schreiben
            org.omg.CORBA.Object obj;
            BufferedWriter reffile;

            KontoImpl konto1 = new KontoImpl();
            obj = poa.servant_to_reference(konto1);
            reffile = new BufferedWriter(new FileWriter(REFFILE + "1.ref"));
            reffile.write(orb.object_to_string(obj));
            reffile.newLine();
            reffile.close();

            KontoImpl konto2 = new KontoImpl();
            obj = poa.servant_to_reference(konto2);
            reffile = new BufferedWriter(new FileWriter(REFFILE + "2.ref"));
            reffile.write(orb.object_to_string(obj));
            reffile.newLine();
            reffile.close();

            KontoImpl konto3 = new KontoImpl();
            obj = poa.servant_to_reference(konto3);
            reffile = new BufferedWriter(new FileWriter(REFFILE + "3.ref"));
            reffile.write(orb.object_to_string(obj));
            reffile.newLine();
            reffile.close();

            // ORB starten
            orb.run();
        }
        catch(IOException e3) {
            System.err.println("Konnte Referenzdateien nicht anlegen.");
        }
        catch(SystemException e1) {
            e1.printStackTrace();
        }
        catch(java.lang.Exception e2) {
            e2.printStackTrace();
        }
    }
}
```



AUF DER CD...

...finden Sie Java, alles was Sie zu Corba benötigen und viele Entwicklungstools.

Außerdem stehen viele Informationen im Netz: www.PC-Magazin.de/download/special_21/

Die Objekte, die *KontoHelper.narrow()* aber instanziiert, sind vom Typ *Konto*. *Konto* ist wieder die (abstrakte) Basisklasse, auf der Stub und Skeleton gleichermaßen aufbauen.

Diese Basisklasse ist in CORBA der ideale Stellvertreter für entfernte Objekte, denn Sie deklarieren die Metho-

den zwar, implementieren sie nicht. Die Implementierung liegt auf dem Server. Im Client wird nur eine Schnittstelle benötigt, welche die Basisklasse liefert.

Anschließend können Sie mit den Objekten arbeiten, als ob sie lokal existierten.

UR

KONTOCLIENT.JAVA

```
import java.io.*;
import org.omg.CORBA.*;

public class KontoClient {
    static String REFFILE = "/tmp/Konto";

    public static void main(String args[]) {
        try {
            // ORB initialisieren
            ORB orb = ORB.init(args, null);

            // Objekte aus Referenzdateien initialisieren
            org.omg.CORBA.Object obj;
            Konto[] konto = new Konto[3];

            obj = orb.string_to_object("file://" + REFFILE + "1.ref");
            konto[0] = KontoHelper.narrow(obj);

            obj = orb.string_to_object("file://" + REFFILE + "2.ref");
            konto[1] = KontoHelper.narrow(obj);

            obj = orb.string_to_object("file://" + REFFILE + "3.ref");
            konto[2] = KontoHelper.narrow(obj);

            konto[0].setDispokredit(6000);
            konto[1].setDispokredit(3000);

            for(int n = 0; n < 3; n++) {
                printKontostand(konto[n]);
            }

            gutschreibenAufKonto(konto[0], 10000);
            gutschreibenAufKonto(konto[2], 540);

            abbuchenVonKonto(konto[0], 5000);
            abbuchenVonKonto(konto[1], 600);
            abbuchenVonKonto(konto[2], 700);
            abbuchenVonKonto(konto[2], 300);
        }
        catch(SystemException e) {
            System.err.println(e);
        }
    }

    static void printKontostand(Konto kto) {
        System.out.println("Konto Nr. " + kto.getKontoNr());
        System.out.println("  Kontostand : " + kto.getKontostand());
        System.out.println("  Dispokredit: " + kto.getDispokredit());
        System.out.println("");
    }

    static void abbuchenVonKonto(Konto kto, int betrag) {
        if(kto.abbuchen(betrag))
            System.out.println("Abbuchung erfolgreich (Konto Nr. " + kto.getKontoNr() + "): -" + (new Integer(betrag)).toString());
        else
            System.out.println("Kreditrahmen ueberschritten (Konto Nr. " + kto.getKontoNr() + "): -" + (new Integer(betrag)).toString() + " nicht moeglich");
        printKontostand(kto);
    }

    static void gutschreibenAufKonto(Konto kto, int betrag) {
        kto.gutschreiben(betrag);
        System.out.println("Gutschrift erfolgreich (Konto Nr. " + kto.getKontoNr() + "): +" + (new Integer(betrag)).toString());
        printKontostand(kto);
    }
}
```