



CORBA mit C++

(Ver)teile und herrsche



Um mit CORBA arbeiten zu können, müssen keine astronomischen Summen ausgegeben werden. Es gibt eine ganze Reihe von freien ORBs. Mit MICO können Sie in die CORBA-Programmierung mit C++ einsteigen.

OLIVER MÜLLER

CORBA ist zunächst eine Spezifikation und ein Standard, der jedem zugänglich ist und nicht von einer einzigen Firma dominiert wird. Sie haben Freiheit bei der Auswahl des ORBs, da Sie nicht auf einen einzigen Hersteller festgelegt sind.

Gerade für den Einsteiger kommt noch ein unschlagbarer Vorteil hinzu. Eine ganze Reihe von ORBs sind frei verfügbar. Sie müssen für Ihre ersten

Gehversuche keine horrenden Summen investieren, nur um einmal CORBA-Luft zu schnuppern. Die freien CORBA-Systeme sind dabei nicht nur Spielzeuge, sondern teilweise kommerziellen Systemen durchaus ebenbürtig.

■ Frei mit MICO

MICO ist eine Open-Source-Implementierung der CORBA-Spezifikation 2.1. MICO selbst steht unter der GPL und seine Libraries unter der LGPL. MICO selbst (also der ORB und die Tools) können nicht in kommerzielle Produkte einfließen. CORBA-Anwendungen, die MICO verwenden, müssen allerdings nicht Open-Source sein. Sie können also Programme mit MICO entwickeln und diese unter eine proprietäre Lizenz stellen.

MICO ist ein ORB für C++. Sie können ihn unter Windows 9x/ME/NT/2000, Linux, Solaris, AIX und HP-UX einsetzen. Die entwickelten Anwendungen sind selbstverständlich über Plattformgrenzen hinweg lauffähig. Außerdem zeigt sich MICO sehr kontaktfreudig im Hinblick auf andere ORBs. Sie können MICO mit anderen CORBA-Systemen, wie den im nächsten Beitrag behandelten JacORB, zusammenarbeiten lassen. Einzige Voraussetzung: Sie dürfen nur in der CORBA-Spezifikation definierte Features verwenden. Sowie Sie spezielle MICO-Features einsetzen, wird der andere ORB Ihre Anwendung nicht mehr verstehen.

■ Installation unter Windows

Um MICO unter Windows einsetzen zu können, benötigen Sie entweder Visual C++ in Version 5.0 mit Service Pack 3 oder Version 6.0 mit Service Pack 2. Alternativ können Sie das freie GNU-Compilersystem Cygwin verwenden. Dieses finden Sie unter <http://www.cygwin.com>. In diesem Fall kompilieren Sie MICO ähnlich wie unter Linux.

Wenn Sie mit Windows 95 arbeiten, müssen Sie sich die WinSock2 installieren. Der TCP/IP-Protocol-Stack unter Windows 95 verursacht mit MICO erhebliche Probleme.

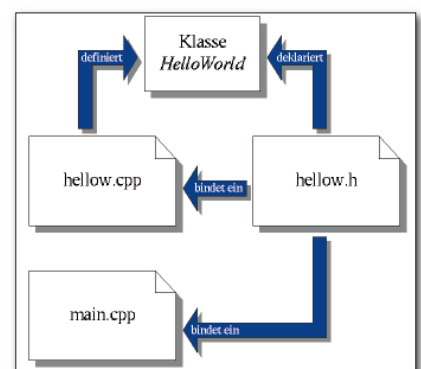
Die WinSock2-Library können Sie bei Microsoft unter http://www.microsoft.com/windows95/downloads/contents/wuadmintools/s_wunetwor kingtools/w95sockets2/default.asp? site=95 herunterladen.

MICO finden Sie auf der CD-ROM dieses Heftes im Verzeichnis \MICO. Entpacken Sie das ZIP-File *mico-2.3.2.zip* auf Ihre Festplatte. Wenn Sie es auf Laufwerk C: entpacken, entsteht dort ein Verzeichnis C:\MICO.

Bevor Sie nun sich frisch ans Werk

QUICK INDEX

- ▶ **Frei mit MICO**
MICO ist ein Open-Source-ORB für C++.
- ▶ **Installation unter Windows**
Mit Visual C++ können Sie MICO einsetzen.
- ▶ **Installation unter Linux**
Unter Linux kann MICO ebenfalls verwendet werden.
- ▶ **Miniaturbank**
Ein kleines Projekt zur Kontoführung mit CORBA realisieren.
- ▶ **Verteilt serviert**
Der Aufwand zum Programmieren eines CORBA-Servers ist mit BOA nicht groß.
- ▶ **Klientel**
Der Client ist mit BOA einfach zu realisieren.
- ▶ **Kontaktfreudiger**
Die Realisierung des Client-Server-Systems mit POA.



TRAGEN SIE DIE MICO-Pfade in die Visual C++ IDE ein.



machen, müssen Sie eine MS-DOS-Eingabeoberfläche öffnen. Danach setzen Sie die Umgebungsvariablen für Visual C++, sofern Sie nicht mit NT oder Windows 2000 arbeiten und bereits bei der Installation von Visual C++ angegeben haben, dass die Umgebung entsprechend angepasst werden soll. Sie finden hierzu im Installationsverzeichnis von Visual C++ eine Batch-Datei *VCVARS32.BAT*. In der Regel befindet sich diese Stapeldatei im Verzeichnis *C:\Programme\Microsoft Visual Studio\VC98\Bin*.



Ziehen Sie *VCVARS32.BAT* aus dem Explorer in das Fenster der Eingabeoberfläche. Auf diese Weise wird der Pfad automatisch eingefügt. Sie müssen dann nur noch die Eingabetaste drücken.

Jetzt wechseln Sie in das MICO-Verzeichnis. Wenn Sie MICO unter *C:\MICO* entpackt haben, geben Sie auf der Eingabeoberfläche folgenden Befehl ein:

```
cd \mico
```

Wenn Sie mit Windows NT, ME oder 2000 arbeiten, starten Sie den Build-Vorgang durch

```
nmake /f Makefile.win32
```

Verwenden Sie Windows 95 oder 98, müssen Sie folgenden Befehl eintippen:

```
nmake /f Makefile.win32 w95-all
```

Dieser Compilierungsprozess erzeugt die MICO-Tools und -Libraries im Unterverzeichnis *win32-bin*. Damit Sie diese auf der Kommandozeile einsetzen können, geben Sie dieses Verzeichnis in Ihrer PATH-Variablen an. Dies erreichen Sie durch das Kommando

```
PATH c:\mico\win32-bin;%PATH%
```

Dies funktioniert allerdings nur unter der Voraussetzung, dass MICO bei Ihnen im Verzeichnis *C:\MICO* liegt. Ist MICO bei Ihnen an anderer Stelle installiert, passen Sie den Pfad entsprechend an.



Nehmen Sie die Batch-Datei *VCVARS32.BAT* und die PATH-Anweisung in die Datei *C:\AUTOEXEC.BAT* auf. Auf diese Weise werden die Umgebungsvariablen bei jedem Boot automatisch gesetzt.

■ Installation unter Linux

Um MICO unter Linux zu installieren, loggen Sie sich zunächst als "root" ein. Anschließend öffnen Sie ein Shell-Fenster (=Terminal), wenn Sie unter X11

LISTING 1: DER BOA-SERVER

```
// ORB initialisieren
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "mico-local-orb");
CORBA::BOA_var boa = orb->BOA_init(argc, argv, "mico-local-boa");

// Server-Objekte erzeugen und
// Referenz-Strings in Datei schreiben
CORBA::String_var ref;
ofstream refFile(REFFILE);

if(!refFile) {
    cerr < "Konnte " < REFFILE < " nicht oeffnen!" < endl;
    return 1;
}

Konto_impl *Konto1 = new Konto_impl;
ref = orb->object_to_string(Konto1);
refFile < ref < endl;

// ... dasselbe für Konto2 und Konto3

refFile.close();

// ORB starten
boa->impl_is_ready(CORBA::ImplementationDef::_nil());
orb->run();
```

LISTING 2: DER BOA-CLIENT

```
// ORB initialisieren
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "mico-local-orb");
CORBA::BOA_var boa = orb->BOA_init(argc, argv, "mico-local-boa");

// Objekte initialisieren aus Referenzdatei
ifstream reffile(REFFILE);

if(!reffile) {
    cerr < "Konnte " < REFFILE < " nicht oeffnen!" < endl;
    return 1;
}
reffile.unsetf(ios::skipws);

char ch;
char ref[1000];
unsigned n = 0, i = 0;
Konto_var Konto[3];

reffile > ch;
while(!reffile.eof()) {
    if(ch == '\\n') {
        ref[i] = '\\0';

        // CORBA-Objekt initialisieren
        CORBA::Object_var obj = orb->string_to_object(ref);
        Konto[n++] = Konto::_narrow(obj);

        i = 0;
    } else if(ch != '\\r') {
        ref[i++] = ch;
    }
    reffile > ch;
}

reffile.close();

// Arbeiten mit den Objekten
Konto[0]->setDispokredit(6000);
Konto[1]->setDispokredit(3000);
//...
```



arbeiten. Als erstes legen Sie die CD-ROM ein und mounten diese durch

```
mount /mnt/cdrom
```

bzw. unter SuSE Linux durch

```
mount /cdrom
```

Dann wechseln Sie in das Home-Verzeichnis:

```
cd
```

und entpacken die MICO-Sourcen durch

```
tar xzf /mnt/cdrom/MICO/  
mico-2.3.2.tar.gz
```

bzw. unter SuSE durch

```
tar xzf /cdrom/MICO/mico-  
2.3.2.tar.gz
```

Danach unmounten Sie die CD-ROM wieder und entnehmen Sie aus dem Laufwerk:

```
umount /mnt/cdrom
```

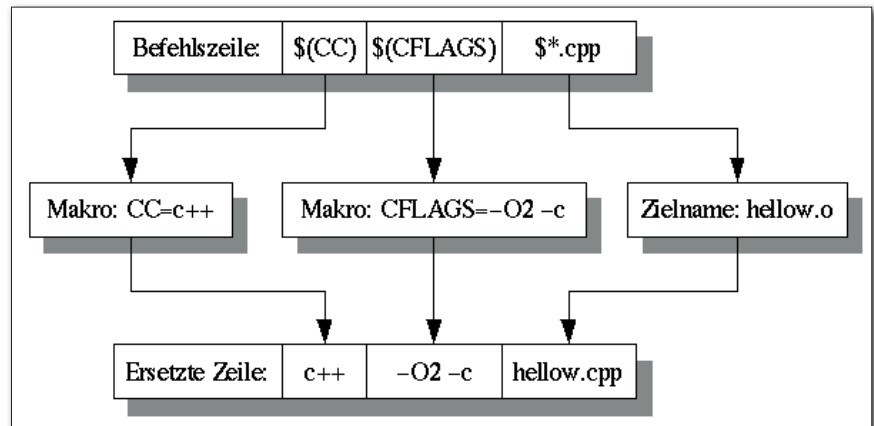
bzw.

```
umount /cdrom
```

Das Kompilieren von MICO starten Sie durch die folgenden drei Kommandos:

```
cd mico  
./configure  
make
```

Ist der Build von MICO erfolgreich, müssen Sie die neuen Tools, Include-



DIE MICO-BIBLIOTHEKEN müssen Sie bei Projekt angeben, damit's funktioniert.

Dateien und Libraries nur noch installieren:

```
make install
```

Damit die neuen Libraries gefunden werden, bearbeiten Sie zunächst die Datei /etc/ld.so.conf. Wenn dort die Zeile

```
/usr/local/lib
```

nicht zu finden ist, geben Sie diese noch ein und speichern die Datei ab.

Jetzt geben Sie noch

```
ldconfig
```

ein und die MICO-Bibliotheken sind verfügbar.

Die Werkzeuge von MICO sind ggf. noch nicht verfügbar.

Dies testen Sie durch das Kommando

```
which idl
```

Sollten Sie nicht die Ausgabe "/usr/local/bin/idl" erhalten, werden die Tools noch nicht von Linux gefunden. Um dieses Manko auszugleichen, tragen Sie in die Datei /etc/profile die Zeile

```
export PATH=$PATH:/usr/local/bin
```

ein. Nach dem nächsten Login, sollten die Tools dann verfügbar sein.

■ Miniaturbank

Als Demonstration, wie Sie in CORBA programmieren, haben wir für Sie auf der CD-ROM ein Projekt parat gelegt. Dieses Projekt implementiert eine Miniaturbank. Diese besteht aus einem CORBA-Server, der das Führen von Konten übernimmt. Sie können diesen Server als das Rechenzentrum unserer Bank auffassen. Der Client ist eine Bankfiliale, welche die Dienste des Rechenzentrums, die Kontoführung in Anspruch nimmt.



Mehrere Clients können parallel gestartet werden, um so mehrere Filialen zu simulieren.

Die Konten bieten die klassischen Girokontenmerkmale. Geld kann darauf gutgeschrieben oder davon abgebucht werden. Für die Konten können Dispositionskredite eingeräumt werden. Das traditionelle "Überziehen" von Konten ist auch bei unserer Minibank möglich. Lediglich bei den Zinsen ist unsere Minibank großzügig, es werden keine verlangt.

Die entsprechenden Methoden für diese Operationen definiert innerhalb der Klasse *Konto* die Datei *Konto.idl*, die in Listing 1 wiedergegeben ist. Die Be-

LISTING 3: DER POA-SERVER

```
// ORB initialisieren
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
CORBA::Object_var poaobj = orb->resolve_initial_references("RootPOA");
PortableServer::POA_var poa = PortableServer::POA::_narrow(poaobj);
PortableServer::POAManager_var mgr = poa->the_POAManager();

// Server-Objekte erzeugen und
// Referenz-Strings in Datei schreiben
CORBA::Object_var ref;
CORBA::String_var str;
PortableServer::ObjectId_var oid;
ofstream refFile;

Konto_impl *Konto1 = new Konto_impl;
oid = poa->activate_object(Konto1);
refFile.open(REFFILE "1.ref");
ref = poa->id_to_reference(oid.in());
str = orb->object_to_string(ref.in());
refFile < str.in() < endl;
refFile.close();

// ... dasselbe für Konto2 und Konto3

// ORB starten
mgr->activate();
orb->run();
```



träge werden an die betreffenden Methoden als in-Parameter übergeben, da Sie vom Server nur gelesen werden müssen. Die Methoden erklären sich im Wesentlichen von selbst. Beim *Abbuchen()* gilt: Diese Methode gibt über einen Wert vom Typ *boolean* zurück, ob das Abbuchen des Betrags vom Konto möglich war. Wenn nicht, wurde der Dispokredit - sofern über *setDispokredit()* eingerichtet - bereits ausgereizt.

Diese IDL-Datei definiert die über CORBA verfügbaren Methoden. Damit daraus die Basisklassen (u. a. Stub und Skeleton) geniert werden, müssen Sie diese Datei dem IDL-Kompiler übergeben. Wollen Sie Ihr CORBA-System über BOA organisieren, verwenden Sie einfach das Kommando

```
idl Konto.idl
```

Für eine POA-Implementierung verwenden Sie stattdessen

```
idl -poa -no-bao Konto.idl
```

Der IDL-Kompiler von MICO erzeugt daraus die Dateien *Konto.cc* und *Konto.h*. Beide Dateien definieren sowohl das Skeleton, als auch den Stub. Sie binden also das Modul *Konto.cc* sowohl beim Server, als auch beim Client ein.



Über die Option `-c++-suffix=cpp` des IDL-Compilers wird statt *Konto.cc* die Datei *Konto.cpp* generiert.

In den Dateien *Konto_impl.h* und *Konto_impl.cpp* wird die Klasse *Konto_impl* deklariert und implementiert. Sie erbt bei BOA von der Skeleton-Klasse *Konto_skel*. Bei POA heißt diese Klasse *POA_Konto*.

Dort werden nun die Methoden mit Leben gefüllt. Die Vorgänge sind recht einfach. Die Methoden operieren mit arithmetischen Ausdrücken über den Attributen *Kontostand* und *Dispokredit*.

Die Datentypen der CORBA-Methoden beginnen jeweils mit dem Namespace *CORBA*. Sie sind einfache Typdefinitionen der IDL-Datentypen auf C++-Datentypen. Statt *CORBA::ULong* könnten Sie durchaus auch *unsigned long* verwenden, allerdings ist die erste Form wesentlich weniger fehleranfällig.

Damit Sie bei den CORBA-Datentypen keine Fehler machen, übernehmen Sie einfach die Deklarationen der Methoden aus der Klasse *Konto*. Diese finden Sie nach der Generierung via IDL in der Datei *Konto.ccbzw. Konto.cpp*. Sie müssen hier lediglich das Schlüsselwort *virtual* weglassen, da Sie die Methoden wirklich implementieren und wieder eine abstrakte Basisklasse schaffen wollen.

Mehr ist nicht zu tun, als die Methoden wirklich zu implementieren. Dies geschieht nicht anders, als bei "normalen" Klassen. Der Rest der CORBA-Programmierung spielt sich jetzt in den Servern und Client-Programmen ab.

■ Verteilt serviert

Ein Server, der den BOA (Basic Object Adapter) des ORBs nutzt, beginnt zunächst mit der Initialisierung von ORB und BOA. In *server_main.cpp* sind dies die ersten beiden Anweisungen in der *main*-Funktion.

Zunächst wird der ORB durch *CORBA::ORB_init()* auf den Weg gebracht. Es werden dabei die Kommandozeilenargumente *argc* und *argv* übergeben. Es ist möglich, über die Befehlszeile Optionen an den ORB zu übergeben. Versteht *ORB_init()* eine Kommandooption, wird diese berücksichtigt und aus *argv* entfernt. Nach dem selben Prinzip funktioniert dies auch bei der BOA-Initialisierung, die über die Methode *BOA_init()* des zuvor erzeugten ORB-Objekts erfolgt.

Jetzt werden drei *Konto_impl*-Ob-

LISTING 4: DER POA-CLIENT

```
// ORB initialisieren
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

// Objekte initialisieren aus Referenzdateien
CORBA::Object_var obj;
Konto_var Konto[3];

obj = orb->string_to_object("file://" REFFILE "1.ref");
Konto[0] = Konto::_narrow(obj);

// ... dasselbe für Konto[1] und Konto[2]

// mit den Objekten arbeiten
Konto[0]->setDispokredit(6000);
Konto[1]->setDispokredit(3000);
// ...
```

jekte instanziiert. *Konto_impl* wird deshalb verwendet, da nur diese Klasse aus dem Reigen der CORBA-Klassen überhaupt einen Sinn zur Instanziierung macht. Die Methoden, die die Operationen bereitstellen, sind nur dort implementiert. Von jedem Objekt wird über die Methode *object_to_string()* des ORBs die Objektreferenz in einen String konvertiert. Diesen Referenz-String kann jeder Client verwenden, um das CORBA-Objekt dieses Servers anzusprechen. Sie können sich diese Referenz wie eine Telefonnummer vorstellen, die innerhalb des laufenden CORBA-Systems eindeutig ist. Durch das Übertragen in einen String wird quasi eine Visitenkarte gedruckt, welche die Telefonnummer enthält. Wenn jetzt ein Client diese Visitenkarte erhält, kann er das CORBA-Objekt "anrufen" und mit diesem interagieren. Diese Referenz-Strings schreibt der Server in eine Datei, die unter Linux in diesem Beispiel */tmp/konto.objids* und unter Windows *C:\konto.objids* heißt.

NAMING SERVICE

Objektreferenzen in Strings zu packen und über Dateien auszutauschen ist zwar CORBA-Standard, aber nicht sehr elegant. Naming Services (NS) lösen diesen Nachteil komfortabel. Jeder ORB, der etwas auf sich hält, bietet Ihnen einen solchen Service.

Bei einem laufenden CORBA-Naming-Service registrieren sich die verteilten Objekte. Über eine klar definierte ID können die Clients die Objektreferenzen direkt beim Naming Service über das Netzwerk abholen. String-Referenzen und umständlicher Datenaustausch über Dateien ade!

Dieser Luxus hat jedoch seinen Preis. Die Naming Services der einzelnen ORBs sind zueinander nicht kompatibel. Solange Sie also nur einen ORB eines einzigen Herstellers einsetzen, können Sie Objekt-IDs per NS problemlos quer durchs Netzwerk schicken, wie Sie wollen. Sowie aber ein anderer ORB dazu kommt, wird die Sache kompliziert. Zwischen den ORBs müssen Sie wieder auf den CORBA-Standard zurückgreifen. Dann heißt es wieder Objektreferenzen per Strings und Dateien durch die Netzwerkwelt zu schaffen.



TIPP Nachdem der Server gestartet wurde existiert diese Datei. Wenn Sie einen Client auf dem selben Computer starten, kann dieser darauf zugreifen und die Objekte ansprechen. Starten Sie einen Client auf einem anderen Host im Netzwerk, müssen Sie diese Datei zuvor auf diesen Host kopieren.

Damit Verbindungen zu diesen CORBA-Objekten möglich sind, müssen noch `impl_is_ready()` des BOA aufgerufen und der ORB gestartet werden.

Klientel

Der Client (`client_main.cpp`) beginnt zunächst wie der Server. Er initialisiert ebenfalls ORB und BOA. Jetzt werden die Referenz-Strings aus der Datei zurückgelesen. Aus jedem String wird eine Objektreferenz des Typs `CORBA::Object_var` erzeugt. Über die statische Methode `_narrow()` der Klasse `Konto` kann der Client jetzt `Konto`-Instanzen (genauer: Stubs) erzeugen. Der Datentyp `Konto_var` bezeichnet in CORBA nichts anderes, als einen Zeiger auf ein Objekt der Klasse `Konto`.

TIPP Bitte beachten Sie, dass Sie unbedingt die Datei `konto.objids` vom Server-Host auf den Computer, auf dem der Client läuft, übertragen müssen, wenn Server und Client nicht auf dem gleichen Host laufen!

`Konto` ist übrigens die Klasse, die lediglich die Methoden virtuell deklarierte. Diese ist geradezu perfekt als Stub.

Dank der Polymorphie kann eine abstrakte Basisklasse ohne weiteres als Stellvertreter für eine konkrete Implementierung dienen.

Jedesmal, wenn ein solches Objekt verwendet wird, wird damit das Objekt auf dem Server angesprochen. Ein Methodenaufruf baut also eine Verbindung zum Server auf und führt die Methode dort aus.

Das Arbeiten mit den erzeugten Objekten funktioniert jetzt genauso, wie bei Objekten, die Sie lokal, also nicht über CORBA, erzeugt haben. Sie arbeiten wie gewohnt mit den Zeigern auf diesen Objekte. Methodenaufrufe erreichen Sie ganz einfach über den Operator `"->"`.

TIPP Wenn Sie den Client mehrfach hintereinander starten, sehen Sie, dass sich die Kontostände immer wieder neu anpassen. Die Objekte existieren also wirklich auf dem Server und nicht auf dem Client, denn die neuen Kontostände gehen nicht verloren.

Kontaktfreudiger

Der POA (Portable Object Adapter) ist wesentlich besser geeignet, wenn Sie mit anderen ORBs zusammenarbeiten wollen. So können Sie die POA-Implementierung der kleinen Bank problemlos mit der Java-Variante des Beispiels aus dem nächsten Beitrag zusammenarbeiten

LISTING: KONTO.IDL

```
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();
};
```

lassen. Mit anderen Worten: Ihr C++-POA-Client kann mit dem Java-Server und der Java-Client mit dem C++-POA-Server zusammenarbeiten. Probieren Sie das ruhig einmal aus.

Der Aufwand zur Initialisierung des POA ist größer als der beim BOA. Der POA wird nämlich ähnlich wie ein anderes CORBA-Objekt per Referenz vom ORB geholt. Nachdem der ORB initialisiert wurde, wird über die Methode `resolve_initial_references()` eine Objektreferenz geholt. Über die `_narrow`-Methode von `PortableServer::POA` wird dann ein POA-Objekt erzeugt. Außerdem benötigt man einen `POAManager`, der über das POA-Objekt per `the_POAManager()` angefordert wird.

Die `Konto_impl`-Objekte müssen nun explizit per POA aktiviert werden. Dies erreichen Sie über die POA-Methode `activate_object()`. Auch das Erzeugen eines Referenz-Strings ist bei POA etwas aufwendiger. Sie müssen zunächst die Objekt-ID, die `poa->activate_object()` zurücklieferte über die POA-Methode `id_to_reference()` in eine Objektreferenz wandeln. Diese können Sie dann wie gewohnt über die ORB-Methode `object_to_string()` in einen String verpacken.

Um den ORB zu starten, müssen Sie zunächst über den `POAManager` per `activate()` den POA aktivieren. Danach können Sie den ORB starten.

Die Objektreferenzen schreibt der POA-Server nun in drei einzelne Dateien. Ansonsten ändert sich an der Funktion selbst nichts.

Der Client ändert sich vom Prinzip her nur wenig. Die Objektreferenzen werden jetzt aus einzelnen Dateien bezogen. Statt direkt die Referenz-String anzugeben, werden an `string_to_object()` lokale URLs `"file:///..."` übergeben. Die Programme zu MICO und CO finden Sie auf der CD.

MICO IN DIE IDE EINBINDEN

Wenn Sie Ihre CORBA-Anwendungen nicht mit `make` bzw. `nmake` auf dem Kommandoprompt verwalten wollen, können Sie MICO auch in die Entwicklungsumgebung von Visual C++ einbinden. So schreiben Sie direkt in der IDE MICO/CORBA-Programme, ohne sich um weiteres kümmern zu müssen.

Zuerst teilen Sie der IDE mit, wo MICO in Ihrem System residiert. Im Dialogfeld *Optionen*, das Sie über das Menü *Extras/Optionen* öffnen, tragen Sie die MICO-Verzeichnisse im Reiter *Verzeichnisse* ein. Über die Auswahlliste *Verzeichnisse anzeigen für* können Sie festlegen, welche Verzeichnisse aufgelistet werden. Unter *Include-Dateien* fügen Sie diese Einträge hinzu:

```
C:\MICO\win32-bin\include\windows
C:\MICO\win32-bin\include
```

Bei der Auswahl *Bibliothekdateien* tragen Sie den Pfad

```
C:\MICO\win32-bin\lib
```

und bei *Ausführbare Dateien*

```
C:\MICO\win32-bin
```

ein.

Bei CORBA-Projekten müssen Sie unter dem Menü *Projekt/Einstellungen* noch im Reiter *Linker* in der Zeile *Objekt-/Bibliothek-Module* die beiden Libraries `mico232.lib` und `wsock32.lib` eintragen.

Last but not least integrieren Sie die IDL-Dateien Ihres Projekts in den Build-Prozess der IDE. Klicken Sie hierzu mit der rechten Maustaste auf die IDL-Datei und tragen Sie als Build-Kommando

```
idl -c++-suffix=cpp $(InputPath)
```

ein. Bei den Ausgabedateien geben Sie folgende Zeilen an:

```
$(InputName).h
$(InputName).cpp
```