



Einführung in CORBA

In's Feld gestreut



Große Software-Projekte verwenden heute schon standardmäßig **verteilte Architekturen**. Systeme wie CORBA ermöglichen es **mit Objekten** im Netzwerkwerk **so zu arbeiten**, als ob Sie auf dem eigenen **Computer** laufen würden.

OLIVER MÜLLER

Verteilte Systeme erobern seit geraumer Zeit die Software-Landschaft. Kaum ein großes Projekt, das heute gestartet wird, setzt noch auf starre zentrale Architekturen. Die großen Denkkentralen à la Mainframe (=Großrechner) mit Ihren "dummen", also mit keinerlei Algorithmen und Prozessen ausgestatteten Terminals haben ausgedient.

Doch nicht nur Großprojekte in der Wirtschaft setzen auf verteilte Architekturen. Wer sich für Linux und eben-

so für die Programmierung des Desktops GNOME interessiert, wird dort schon über die Begriffe wie ORBit, Gnorba oder Bonobo gestolpert sein, denn auch GNOME's basiert auf CORBA.

■ Warum CORBA?

Heute setzt die Software-Welt auf Client-Server-Systeme und Multitier-Architekturen (=mehrschichtige Systeme). Grundgedanke ist das Verteilen der Objektmodelle. Die Anwendungslogik wird bei diesem Ansatz nicht mehr in einem zentralen Rechner gekapselt, sondern auf mehrere Systeme in einem Netzwerk verteilt. Klarer Vorteil: Fällt ein Computer aus, kann der Rest des Netzwerks in gewissem Umfang weiter arbeiten. Außerdem kann schnell ein Ersatzsystem die Aufgaben des ausgefallenen Systems übernehmen. Früher stand beim Ausfall des Mainframes erst einmal alles still, denn wer hatte schon einen zweiten Großrechner als Ersatz-System?

Die Basis für einen solchen verteilten Software-Ansatz sind Systeme wie

- DCOM (Distributed Component Object Model) von Microsoft,

- RMI (Remote Method Invocation) von Java,
- DCE (Distributed Computing Environment) der OSF (Open Software Foundation) und- CORBA (Common Object Request Broker Architecture) der OMG (Object Management Group).

Solche Systemansätze lassen es zu, dass Software nicht mehr nur auf einem Computer laufen kann, sondern überein Netzwerk verteilt ist. Wenn Sie bislang Objekte in Ihren Programmen verwendet haben, liefen diese ja immer in einer einzigen Anwendung. Eventuell haben Sie sie noch über shared Libraries oder DLLs in mehrere Teile zerlegt. Dies ändert jedoch nichts an der Tatsache, dass alles auf einem einzigen Computer (=in einem Adressraum) läuft. Durch verteilte Architekturen können Sie einzelne Komponenten (bzw. Objekte) Ihrer Software auf verschiedene Computer verteilen (=mehrere Adressräume).

Wann immer Sie eine Komponente oder ein Objekt in einem verteilten System verwenden wollen, müssen Sie sich über dessen Lage im Netzwerk keine Gedanken machen. Sie verwenden es einfach. Die Verwaltung übernimmt das verteilte System, zum Beispiel CORBA.

■ Gekonnt verteilt

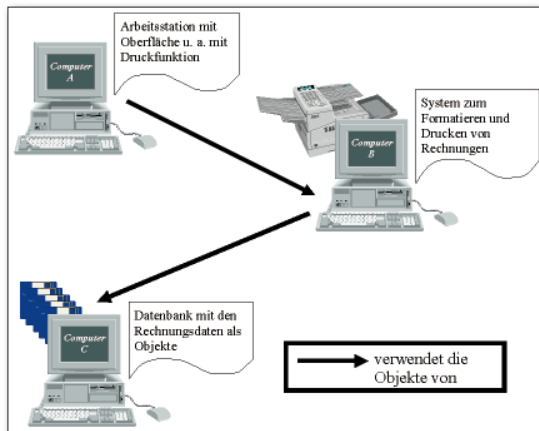
In einem verteilten Objektsystem können Objekte quer über das Netzwerk verstreut auf verschiedenen Computern

QUICK INDEX

- ▶ **Warum CORBA?**
Über CORBA können Objekte im Netzwerk verteilt werden.
- ▶ **Gekonnt verteilt**
Das Funktionsprinzip von CORBA.
- ▶ **Kuck mal, wie man spricht**
Über Skeleton und Stub interagieren CORBA-Clients und -Server.
- ▶ **Schnittmuster**
Mit Hilfe der IDL definieren Sie Schnittstellen für CORBA-Objekte.
- ▶ **Perfekter Entwurf**
Schnittstellendefinitionen in IDL sind Klassendeklarationen in C++ und Java sehr ähnlich.

INFORMATIONEN RUND UM CORBA

Die aktuellsten Informationen zu CORBA erhalten Sie direkt bei der Object Management Group (OMG). Diese finden Sie im Internet unter <http://www.omg.org>.



EIN VERTEILTES SYSTEM: Objektzugriff systemübergreifend.

liegen. Trotzdem können Sie von anderen Hosts im Netz genutzt werden.

Computer A übernimmt das Drucken von Rechnungen. Er bereitet dazu Informationen aus einer Datenbank auf, formatiert eine Rechnung und schickt diese an den Drucker. Hierzu existiert das Objekt *Drucker* in seiner Software. Eine Arbeitsstation (Computer B) hält die Benutzeroberfläche für die Anwender bereit. So, wie hier das Drucken der Rechnung veranlasst wird, verwendet die Software von Computer B das Drucker-Objekt von Computer A, als ob es lokal verfügbar wäre. Computer A könnte sogar von einem Computer C die Objekte aus der Datenbank mit den Rechnungsinformationen beziehen.

Damit diese Computer alle über das Netzwerk auf die jeweiligen Objekte zugreifen können, muss zwangsläufig ein gewisser Aufwand für die Verwaltung betrieben werden. Auf Fragen wie "Wie stelle ich eine Verbindung zwischen den Objekten her?" oder "Wenn eine Methode eines entfernten Objekts gerufen wird, wie wird das über das Netzwerk umgesetzt?" müssen Antworten gefunden werden. Das Positive an verteilten Objektsystemen ist, dass Sie sich als Programmierer darum nicht kümmern müssen.

Bei objektorientierten verteilten Systemen existiert hierzu der ORB (Object Request Broker). Der ORB ist die Kernkomponente des verteilten Systems. Er vermittelt zwischen den verteilten Objekten. Er sorgt dafür, dass Methodenaufrufe an das richtige verteilte Objekt geschickt und die Ergebnisse zum Aufrufer zurückgeleitet werden. Der ORB ist damit eine Schicht, die sich über das Netzwerk spannt und als Kommunikationsplattform für die verteilten Objekte dient.

Wie eine solche Verwaltung aussieht, definiert die Object Management Architecture (OMA) der OMG. Grundsätzlich existieren vier Komponenten:

Application Objects: Dies sind die Objekte der Anwendung(en), die verteilten System angesprochen werden können bzw. mit diesem interagieren. Die Applikationsobjekte sind der zentrale Punkt in Ihrer verteilten Software. Sie implementieren die Anwendung als solche.

Common Facilities: Diese Komponente enthält vorgefertigte Objekte, die die Application Objects unabhängig von ihrer eigentlichen Bestimmung benötigen. Beispiele hierfür sind Objekte zur Fehlerbehandlung oder zum Ansprechen von Peripherie (z. B. Chipkartenterminal oder Drucker).

Object Services: Diese Dienste werden zum Betrieb des verteilten Systems benötigt. Sie koordinieren und steuern die Interaktionen zwischen den Objekten. Die OMG hat für CORBA eine ganze Reihe von Diensten zur Sicherheit, persistenten Speicherung, Ereignisbehandlung etc. definiert. Diese Dienste müssen Sie selbst

nicht implementieren, sie sind schon in ihr CORBA-System integriert.

Object Request Broker: Wenn die drei zuletzt genannten Komponenten die Ziegelsteine einer Mauer sind, dann ist der ORB der Mörtel dazwischen. Der ORB hält das gesamte System zusammen und ermöglicht die Interaktion zwischen den Komponenten und Objekten. Mit anderen Worten: Über den ORB interagieren die anderen Komponenten des CORBA-Modells miteinander.

■ Kuck mal, wie man spricht

Wenn Sie in Java mit der Anweisung `import` ein Package einbinden, werden damit die Klassen inklusive deren Implementierung in das Programm eingebunden. Den selben Effekt haben Sie

MAPPING VON DATENTYPEN

Die Datentypen von C++ und Java müssen auf CORBA-Datentypen umgesetzt werden, da CORBA ein von der Programmiersprache unabhängiges System ist.

In diesem Artikel finden Sie die wichtigsten Datentypen. Für weitere Informationen zum Mapping von C++ und Java sei auf die folgenden Internetseiten verwiesen:

Für C++: <http://www.omg.org/technology/documents/formal/c++.htm>

Für Java: http://www.omg.org/technology/documents/formal/java_language_mapping_to_ogm_idl.htm

und http://www.omg.org/technology/documents/formal/ogm_idl_mapping_to_java_language.htm

BOA VERSUS POA

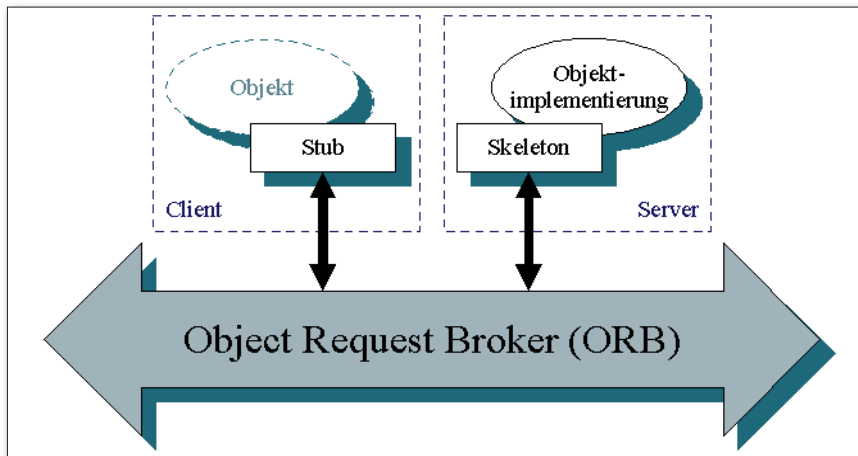
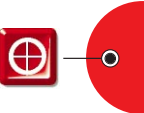
Zwischen dem Server-Objekt und dem ORB kommt der *Object Adapter* zum Einsatz. Beide Seiten nutzen den Adapter für nicht-anwendungsspezifische Interaktionen. Der Server kann beispielsweise Authentifizierungsdaten über den Client beim ORB abfragen. Ein anderes Beispiel wäre es, wenn der ORB ein Server-Objekt startet, weil ein neuer Client-Auftrag eingetroffen ist, das Server-Objekt aber inaktiv war.

Die ersten CORBA-Spezifikationen der OMG enthielten den *Basic Object Adapter* (BOA). Dieser sollte die Basisdienste für den Adapter abdecken.

Dieser Basisadapter stellte sich jedoch nicht als die glücklichste Lösung heraus. Viele Hersteller von CORBA-Systemen entwickelten aufgrund des wesentlich höheren Bedarfs an weiteren Funktionen eigenständig neue Features und fügten diese ihren BOAs hinzu.

Dies hatte allerdings auch zur Folge, dass die einzelnen CORBA-Systeme zunehmend zu einander inkompatibel wurden. Wenn Sie also beispielsweise ein System mit INPRISE's VisiBroker implementierten und nachträglich ein System anbinden wollten, das mit IONA Orbix realisiert wurde, konnten Sie nur hoffen, dass diese ORBs sich vertrugen. Dies war jedoch meist nicht der Fall. Kurzum: Es mussten in der Praxis separate Schnittstellen zwischen diesen Teilsystemen geschaffen werden.

Dieses Manko veranlasste die OMG in CORBA 2.0 den *Portable Object Adapter* (POA) einzuführen. Dieser führte viele der zusätzlich benötigten Features in den Standard ein. Damit wurde die Interoperabilität zwischen den ORBs verbessert. In der Regel sollten Sie davon ausgehen können, dass POA-Systeme miteinander zusammenarbeiten können.



VERWENDEN EINES ENTFERNTEN CORBA-Objekts.

in C++, wenn Sie mit der Präprozessoranweisung `#include` eine Header-Datei einbinden und beim Linken die entsprechende Library zum Programm hinzubinden. Die in der Library enthaltenen Klassenimplementierungen werden damit automatisch in das Programm aufgenommen.

Bei der Verwendung von CORBA soll aber genau das nicht passieren. Die Implementierung soll nicht in das (Client-)Programm eingebunden werden. Stattdessen soll nur eine Schnittstelle in den Client einfließen. Die Implementierung selbst wird nur im Server existieren.

Wenn Sie jetzt allerdings ohne weiteres die `import` bzw. `#include`-Anweisung außen vor lassen, oder die Libraries nicht einbinden bzw. (in Java) nicht verfügbar machen, kommt es zu einem Compilierungsfehler. Ihr Client-Programm würde also gar nicht erst erstellt.

Im Client-Programm muss also eine Art stellvertretendes Objekt geschaffen werden, das alle Anfragen über den ORB an den Server weiterleitet. Im Server-Programm muss die Implementierung des "echten" Objekts mit einer Schnittstelle ausgestattet werden, durch die der Server auf die Anfragen des Clients reagieren kann.

Der Client erhält hierzu eine spezielle Klasse, die eine Schnittstelle für das CORBA-Objekt bereitstellt. Diese Klasse heisst *Stub*. Ein Stub-Objekt stellt auf der Seite des Client die Me-

thoden des Objekts auf dem Server eins zu eins dar. Diese Methoden machen jedoch nichts anderes, als die Nachrichten an den Server weiterzuleiten und dessen Ergebnisse zurück zu liefern.

Auf der Server-Seite wird die Implementierungsklasse, von der das eigentliche Objekt dann instanziiert wird, von einer Klasse abgeleitet, die die CORBA-Funktionalität beisteuert. Diese Klasse nennt sich *Skeleton*. Das Skeleton ist eine abstrakte Basisklasse. Es stellt lediglich die Methoden virtuell zur Verfügung, die über CORBA verfügbar sein sollen. Erst mit dem Vererben der Eigenschaften des Skeletons an die Implementierungsklasse, werden diese virtuellen Methoden konkretisiert und mit "Leben" gefüllt.

Beide Klassen, sowohl Stub, als auch Skeleton (und die interne Klasse, auf der Sie basieren), werden von den Entwicklungstools der CORBA-Umgebung automatisch generiert. Lediglich die Implementierungsklasse müssen Sie selbst schreiben, wobei viele kommerzielle CORBA-Implementierungen Ihnen hier auch Assistenten an die Hand liefern, die für Sie einen Rahmencode generieren. Sie sparen sich so Tipparbeit (und Tippfehler), denn Sie müssen "nur" noch die generierten "Methodenhüllen" mit den entsprechenden Operationen und Algorithmen füllen.

■ Schnittmuster

Stub und Skeleton werden von den Entwicklungstools Ihrer CORBA-Umgebung automatisch generiert. Es werden Klassen erzeugt, die die Schnittstelle für Ihre konkrete Implementierung darstellen. Die Methoden, die Sie später implementieren wollen, werden hier bereits in Ihrer Aufrufsyntax festgelegt. Damit die CORBA-Tools aber überhaupt "wissen", was Sie für Methoden wünschen, müssen Sie das den Werkzeugen mitteilen. Zu diesem Zweck existiert die IDL. IDL steht für "Interface Definition Language". Es ist eine sehr an C++ und Java angelehnte Sprache, durch die Sie Klassenschnittstellen definieren können. Sie definieren Ihre Klassenschnittstelle in einer Textdatei mit der Endung `.idl` (=IDL-Datei). Diese Datei übergeben Sie dem IDL-Compiler Ihres CORBA-Systems. Dieser erzeugt dann Stub und Skeleton sowie die Basisklasse, von der beide geerbt haben.

OUT-PARAMETER

Die out-Parameter von IDL werden in C++ auf Referenzen (&) gemapped. `out long` wird demnach in C++ zu `long&`. IDL-in-Parameter, die in C++ Zeiger sind, werden zu const-Zeigern gemapped. `in wstring` wird damit zu `const wchar_t*`.

In Java werden Objekte ohnehin "by reference" übergeben. Somit erübrigt sich die Frage, wie Objekte als out-Parameter dargestellt werden. Bei primitiven Typen wie Javas `int` ist die Frage jedoch berechtigt, da primitive Typen in Java immer "by value" übergeben werden. Hier werden Holder-Klassen eingeführt, die das Mapping umsetzen. Aus dem IDL-Ausdruck `out long` wird in Java `Int-Holder`.

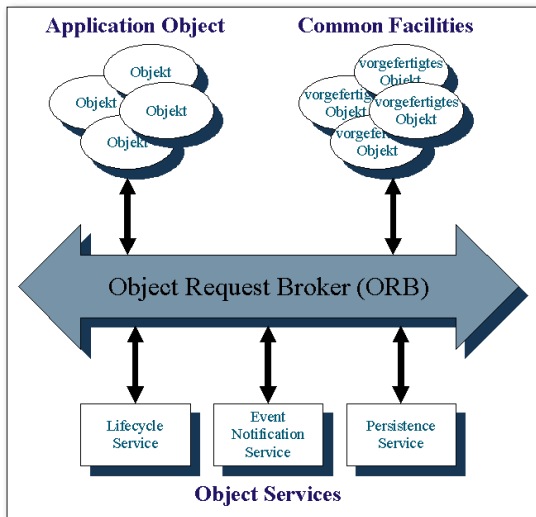
FRÜHES UND SPÄTES BINDEN

In diesem Artikel wurde bislang nur das "frühe Binden" angesprochen. Der Client wird mit einem statischen Stub versehen, der bereits beim Compilieren und Linken eingebunden wird. Damit ist klar festgelegt, welche Dienste das Server-Objekt bereitstellt. In der Vielzahl der Fälle ist das absolut ausreichend.

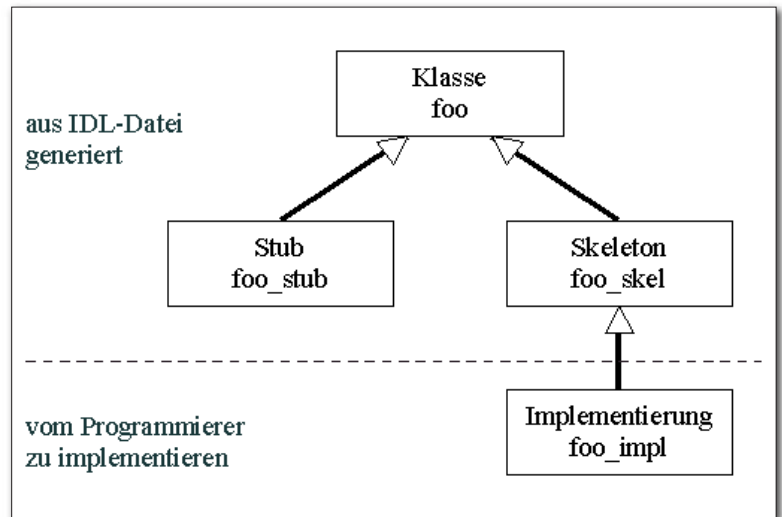
Allerdings ist auch "spätes Binden" möglich, bei dem erst zur Laufzeit dynamisch die Schnittstellen von CORBA-Objekten eingebunden werden. Über einen Loader-Mechanismus ermöglicht es der ORB, dem die Schnittstelle des Server-Objekts bekannt ist, die Schnittstelle dynamisch nachzuladen.

LISTING 1: HANDELSWARE.IDL

```
interface Handelsware {
    void bestand_erhoehen(in unsigned long menge);
    unsigned long bestand_vermindern(in unsigned long menge);
    unsigned long bestand_ausgeben();
}
```



DER AUFBAU VON CORBA.



VERERBUNGSBEZIEHUNG zwischen den CORBA-Klassen.

Auf diese Weise wird sichergestellt, dass Client-Stub und Server-Skeleton immer identische Schnittstellen verwenden. Außerdem wird Ihnen bei diesem Prozess Arbeit abgenommen, denn Sie müssen die Klassendeklarationen nicht selbst eintippen.

Das Modul mit der Stub-Klasse können Sie ohne weiteres in das Client-Programm einbinden. Von der Skeleton-Klasse leiten Sie anschließend noch eine konkrete Klasse (=Implementierungsklasse) ab. Beides müssen Sie dann in Ihren CORBA-Server einbinden.

Sowohl in Ihren CORBA-Client, als auch in Ihren Server müssen Sie die CORBA-Library einbinden bzw. die betreffenden Packages importieren, damit alles reibungslos funktioniert.

■ Perfekter Entwurf

Da Sie über IDL die Schnittstelle festlegen, über die Client und Server zusam-

menarbeiten, steht die IDL im Mittelpunkt der CORBA-Programmierung. Nur wer IDL beherrscht, kann wirklich mit CORBA programmieren.

IDL sieht wie C++- oder Java-Code aus. Da es sich bei IDL jedoch um eine Sprache zur reinen Beschreibung von Schnittstellen handelt, ist sie auf das Deklarieren von Klassen, Typen und Methoden reduziert. Sie können keine komplette Klasse implementieren – das erledigen Sie im Server in Ihrer eigenen Programmiersprache (C++ oder Java).

Eine Beispiel-Interface-Definition in IDL finden Sie in Listing 1 "Handelsware.idl". Diese IDL-Datei definiert die Schnittstelle für die Klasse *Handelsware*. Sie soll den (Lager-)Bestand einer Ware in einem Handelsunternehmen repräsentieren. Von dieser Klasse würden Objekte für jede der in der Firma gehandelten Waren angelegt. Es existieren Methoden zum Ermitteln und Verändern des aktuellen Bestandes. Beim Erniedrigen wird ein Wert zurückgeliefert, der angeben soll, ob wirklich der Bestand um die angegebene Menge reduziert werden konnte. Es ist nämlich sehr unsinnig in einem Lager negative Bestände zu führen.

Wie Sie sehen wird die Interface Definition anstatt mit dem Schlüsselwort *class* mit *interface* eingeleitet. Dies ist eine markante Eigenschaft von IDL.

Die Methoden werden wie in C++ oder Java deklariert. Sie müssen lediglich darauf achten, dass Sie nur die Da-

tentypen verwenden, die in IDL gültig sind. Näheres erläutert Ihnen zu diesem Thema der Textkasten "Mapping von Datentypen".

Auffallend ist bei den Methoden das Schlüsselwort *in*. Dies gibt an, dass der betreffende Parameter ein Eingabeparameter ist. Dieser Wert wird also vom Server nur gelesen. Er wird nicht verändert. Wenn Sie also beispielsweise einem Parameter vom Typ *in long* eine Variable mit dem Wert fünf übergeben, ist nach dem Aufruf der Methode der Inhalt der Variable immer noch fünf ("call by value"). Parameter, die statt *in* das Schlüsselwort *out* in Ihrer Deklaration enthalten, sind für die Ausgabe bestimmt. Es ist also egal, welchen Wert die Variable vor dem Aufruf der Methode hatte, er wird mit einem Ergebnis überschrieben. Der Wert wird auch von CORBA nicht an den Server übergeben und darf folglich im Server nicht beachtet werden. Der Server gibt jedoch in dieser Variable einen Wert aus. Dieser Wert kann im Client über die entsprechende beim Methodenaufruf übergebene Variable ausgelesen werden (eine Art One-Way-"call by reference").

Last but not least gibt es noch das Schlüsselwort "inout", über das eine Variable sowohl als Eingabe-, als auch als Ausgabeparameter deklariert werden kann. Damit sind die Möglichkeiten von IDL noch nicht ausgeschöpft. Insbesondere das Typmapping ist sehr umfangreich und sehr interessant, kann jedoch im Rahmen dieses Beitrags nicht umfassend behandelt werden. Wir empfehlen Ihnen deshalb, die Homepage der OMG zu besuchen.

UR

DIE WICHTIGSTEN DATENTYPEN

Die wichtigsten Datentypen

C++	Java	IDL
void	void	void
unsigned char	boolean	boolean
unsigned wchar_t	char	wchar
unsigned char		char
unsigned char	byte	octet
short	short	short
long	int	long
LongLong	long	long long
float	float	float
double	double	double
wchar_t*	String	wstring
char*	string	