
Prüfer: Prof. Dr. Rothermel
Betreuer: Dr. Cora Burger
Mitbetreuer: Dipl.-Inform. Fritz Hohl

Begonnen am: 01. März 1996
Beendet am: 31. August 1996

CR-Klassifikation: C.2.4, D.4.4

Studienarbeit Nr. 1540

**Einbindung des MELODY-Traders
in das Agentensystem zur Vermitt-
lung von CORBA-Diensten**

Gerald Vogt

Zusammenfassung

Mit dem Wachstum der Netzwerke wächst beständig die Zahl konkurrierender Dienst-Anbieter. Dies macht es jedoch immer schwieriger, alle Dienst-Anbieter aufzufinden und aus dem Angebot den besten auszuwählen. In dieser Situation versuchen Vermittlungsdienste zu helfen. Sie verwalten Listen von Dienst-Anbietern und ermöglichen es Klienten, Anfragen mit verschiedenen Kriterien zu stellen. Die Bedeutung von Vermittlungsdiensten wächst immer mehr. Ein Trader wurde als Teil des Melody-Projektes der Universität Stuttgart entwickelt.

In dieser Arbeit wird eine neue Schnittstelle für Melody implementiert, die als Erweiterung zur bestehenden DCE-Schnittstelle den Zugriff auf Melody über CORBA ermöglicht. Es werden Fragen der Entwicklung der Schnittstellenbeschreibung aufgegriffen und ein Weg wird gezeigt, wie diese mit derjenigen von DCE verbunden werden kann. Darüber hinaus wird die Integration der CORBA Schnittstelle erläutert, um so sie so besser verständlich zu machen und damit bei zukünftigen Änderungen zu helfen.

Mit Hilfe dieser neuen Schnittstelle wird Melody im Agentensystem Mole verfügbar gemacht, das entwickelt wird, um die Benutzung mobiler Agenten in verteilten Systemen zu erforschen. Dazu muß es möglich sein, Melody von Agenten aufzurufen, die in Java programmiert sind. Zu diesem Zweck wird eine Java Schnittstelle erstellt, die Melody über CORBA benutzt. Daraufhin wird ein einfacher Agent vorgestellt, der die grundsätzliche Machbarkeit der Aufgabe zeigt.

Abstract

With the growth of the networks the number of concurrent service providers of different kind increases constantly. But this makes it harder to find all of them and to choose the best one. Trading services try to help here. They manage lists of service providers and allow clients to access these list applying different criteria. The importance of such services increases more and more. A trader was implemented as part of the Melody project at the University of Stuttgart.

In this work a new interface to Melody is implemented that allows access to Melody through CORBA in addition to the existing DCE-interface. This work covers questions of the design of the interface description for CORBA and shows a way how to combine it with the one for DCE. Moreover the integration of the CORBA interface in the existing program is explained for better understanding of the implementation and thereby helping with future changes and extensions.

Using this new interface Melody is made available in the agent system Mole that is developed to research the use of mobile agents in distributed systems. Therefore it must be possible to call Melody from agents that are programmed in Java. For that purpose a Java interface is built that accesses Melody via CORBA and afterwards a simple agent is introduced that shows the general feasibility of that task.

Aufgabenstellung

Trader dienen dazu, zwischen Klienten und Server zu vermitteln. Eine derartige Funktionalität wird auch in Systemen benötigt, in denen Agenten, die aktiven Softwareeinheiten zukünftiger verteilter Systeme, nach Diensten eines elektronischen Marktes suchen (z.B. Datenbankdienste, Reisebüros, Warenhauskataloge).

Im Projekt MELODY wurde ein Trader erstellt, der beliebige Dienste vermitteln kann, u.a. auch solche, die in CORBA-IDL spezifiziert werden (CORBA-IDL ist eine Beschreibungssprache für Schnittstellen, die sich wachsender Bedeutung erfreut). Außerdem steht durch eine Implementierung mit JAVA ein Agentensystem mit mobilen Agenten und ortsfesten Systemagenten zur Verfügung.

Im Rahmen der Studienarbeit sollen einerseits geeignete Abbildungsfunktionen implementiert werden, um den MELODY-Trader als Systemagenten in das Agentensystem einbinden zu können. Außerdem sollen Mechanismen erstellt werden, um die beiden Aufrufmöglichkeiten für CORBA-Dienste (dynamic invocation, Aufruf über Stubs) für Agenten zugänglich zu machen.

Aufgabenstellung

Vorwort

Diese Studienarbeit beschäftigt sich mit der Einbindung von Melody in Java und dem Agentensystem Mole, um so den in Java geschriebenen Agenten den Zugriff auf Melody zu ermöglichen. Im ersten Teil sollen deshalb, nach einer Einleitung in die grundsätzliche Problematik in Kapitel 1, zunächst die wesentlichen beteiligten Komponenten dieser Arbeit kurz vorgestellt werden. Dies sind CORBA in Kapitel 2, Mole in Kapitel 3 und Melody in Kapitel 4. Darauf hin werden in Kapitel 5 generelle Überlegungen zur möglichen Integration beschrieben. Eine erste Möglichkeit wird im folgenden Kapitel untersucht: Die direkte Einbindung. Bei der direkten Einbindung werden die Methoden von Melody direkt von Java aus aufgerufen. Leider stellte sich heraus, daß diese Lösung technisch nicht machbar ist. Sie wird in Kapitel 6 beschrieben.

Deshalb mußte auf eine indirekte Einbindung ausgewichen werden. Aus diesem Grund wurde eine CORBA-Schnittstelle für Melody implementiert. Da in Melody bereits eine Schnittstelle zu DCE vorhanden war, wurde an vielen Stellen versucht, dabei in ähnlicher Weise vorzugehen. Den Ausgangspunkt bildet die DCE-Schnittstellenbeschreibung. Sie enthält in einer CORBA vergleichbaren Form die Zugriffsmöglichkeiten auf Melody. Da sich DCE und CORBA-IDL recht ähnlich sind, wurde versucht durch Einsatz von verschiedenen Textmakros, beide zu kombinieren. Wie dies geschehen ist, wird in Kapitel 7 näher erläutert.

In den beiden darauf folgenden Kapiteln 8 und 9 werden dann die weiteren notwendigen Schritte für die Integration von CORBA gezeigt. Kapitel 8 beschreibt, wie notwendige Konvertierungsmethoden für Melody entwickelt wurden. Diese Methoden haben die Aufgabe, Datenstrukturen von Melody in Datenstrukturen von CORBA umzuwandeln. In Kapitel 9 werden darauf die restlichen Aspekte der Integration näher erläutert. Dabei werden die verschiedenen Teile der Seite des Klienten und des Servers näher betrachtet. Der Hauptgedanke bei diesen Teilen war, ähnlich wie dies schon bei DCE der Fall ist, sowohl den Klienten als auch den Server von den Problemen des Datenaustausches zu befreien, indem eine definierte Schnittstelle vorgegeben wird, die vom Server angeboten und vom Client genutzt wird. Es mußten deshalb Teile implementiert werden, die die Umwandlung von der Klientenschnittstelle nach CORBA bzw. von CORBA zum Server hin übernehmen.

Die beiden nächsten Kapitel beschäftigen sich damit, wie Melody ebenfalls von Java-Programmen und ganz besonders von Mole-Agenten aufgerufen werden kann. Der *Java-Trader User Agent* aus Kapitel 10 stellt genau genommen eine Portierung des bereits vorhandenen TUAs in C++ dar. Der TUA ist das Objekt, mit Hilfe dessen der Klient auf Melody zugreift. Er muß die Daten vom Klienten nach CORBA umwandeln. Für Java wurde ein TUA mit analoger Funktionalität zu C++ entwickelt. Darüber hinaus wurde durch Nachbildung von C++-Zugriffsmethoden eine weitere Annäherung zwischen C++ und Java erreicht, so daß sich die Programmierung für den Klienten in beiden Fällen sehr ähnlich ist.

Das Kapitel 11 beschäftigt sich mit dem Zugriff für Mole-Agenten. Das Agentensystem Mole wird entwickelt, um mobile Agenten – also Agenten, die ihren Ort wechseln – näher untersuchen zu können. Die Agenten von Mole sollten die Möglichkeit erhalten, auf Melody zuzugreifen. Da sie in Java implementiert sind, ist dies bereits mit den Ergebnissen von Kapitel 10 möglich. Doch aus verschiedenen Gründen, die in diesem Kapitel erklärt werden, wurde zusätzlich noch ein ortsfester Systemagent implementiert, über den der Austausch mit Melody abgewickelt werden kann.

Den Abschluß in Kapitel 12 macht der Ausblick.

Bei genauer Betrachtung der Aufgabenstellung zu dieser Studienarbeit und der tatsächlich vorliegenden Arbeit wird man sicher leicht herausfinden, daß eine größere Diskrepanz zwischen beiden besteht. Dies liegt daran, daß die Aufgabenstellung während der Bearbeitung eine doch recht bemerkenswerte Dynamik zu Tage legte. Die Aufgabe hatte zwei Teile: Zunächst die (direkte) Einbindung von Melody in Mole, um CORBA-Dienste zu vermitteln, und danach eine Schnittstelle für Mole zu implementieren, über die der Zugriff auf die vermittelten CORBA-Dienste ermöglicht wird.

Doch bereits zu Beginn der Arbeit erschien von der Firma Orbix mit OrbixWeb eine CORBA-Schnittstelle für Java auf dem Markt. Dieses OrbixWeb bietet für Klienten unter Java alles notwendige an, um aus CORBA-IDL Stubs erzeugen zu lassen und auf CORBA-Objekte zu zugreifen. Da damit bereits der zweite Teil meiner Aufgabe als Produkt erhältlich war, wurde Abstand davon genommen, hier eine eigene, nur sehr eingeschränkte Lösung zu implementieren.

Auch der erste Teil konnte in der ursprünglich vorgesehenen Weise nicht erstellt werden. Es war geplant, die Melody-Einbindung in Mole direkt über die in Kapitel 6 dargestellten *native*-Methode zu machen. Dieser Ansatz wäre vom Umfang her wesentlich geringer ausgefallen, als die jetzt realisierte Integration von CORBA. Doch leider stellte sich heraus, daß er aus technischen Gründen nicht machbar ist. Dies bedingte eine indirekte Einbindung, die über die schon länger in Melody geplante CORBA-Schnittstelle geschaffen wurde, was aber eigentlich als eine getrennte Studienarbeit gedacht war.

Dank gebührt an dieser Stelle Dr. Cora Burger, die neben der Betreuung meiner Arbeit auch mit verschiedensten Vorbereitungen für meine Diplomarbeit beschäftigt war, Ernő Kovacs, der durch die Entwicklung der Aufgabe bedingt, zu einem wesentlichen Betreuer der Arbeit wurde und Fritz Hohl, der wiederum als eigentlicher Mitbetreuer, nicht beleidigt war, daß er nur wenig davon mitbekam, allen anderen, denen Dank gebührt, und die sonst immer vergessen werden, insbesondere denen, die in den letzten, arbeitsreichen Wochen immer noch Geduld mit mir hatten. Der Firma ISA Informationssysteme Stuttgart GmbH, deren Rechner ich einige

Male während der Entstehung dieser Ausarbeitung nutzte, und die mir so eine größere Flexibilität in der Zeiteinteilung bescherten. Den Machern des Word-nach-MIF-Filters, die es mir zudem ermöglichten, einige Male auch zu Hause zu arbeiten. Und schließlich noch Earl Grey, dessen mit Bergamottöl aromatisierter Tee zu einem unverzichtbaren Begleiter bei einige langen Arbeitstagen avancierte.

Inhalt

Zusammenfassung ***i***

Abstract ***iii***

Aufgabenstellung ***v***

Vorwort ***vii***

KAPITEL 1

Einleitung ***1***

Vermittlung **1**

Client/Server **2**

Objekte **3**

Agenten **3**

KAPITEL 2

*CORBA – Common Objekt Request Broker
Architecture* ***5***

Der Object Request Broker (ORB) **6**

OMG Interface Definition Language **9**

KAPITEL 3	<i>Mole – Mobile Agenten</i>	11
KAPITEL 4	<i>Melody</i>	15
KAPITEL 5	<i>Die Melody Integration in Mole</i>	17
KAPITEL 6	<i>Die direkte Anbindung von Melody nach Java</i>	19
	Von Java nach C	19
	Von C nach C++	23
	Von Java nach C++	24
	Machbarkeitstests für die Einbindung	25
KAPITEL 7	<i>Eine CORBA-Schnittstelle für Melody</i>	27
	Beschreibung der Trader Schnittstelle in CORBA IDL	28
	Die Umsetzung	30
	<i>Die Basistypen</i>	30
	<i>Zeichenketten</i>	30
	<i>Die Typ-Definition</i>	32
	<i>Die Umwandlung von definierten Typnamen</i>	34
	<i>Die Definition von Stringtypen</i>	35
	<i>Die Deklarationen von Listen</i>	35
	<i>Benutzung von Strukturzeigern</i>	37
	<i>Die Unions</i>	39
	<i>Die Deklaration der Operationen</i>	40
	Die Übersetzung der CORBA-IDL	41
KAPITEL 8	<i>Die CORBA-Melody Konvertierung</i>	45
	Der vorhandene Aufbau in Melody	46
	Die Integration der CORBA-Schnittstelle	47
	Die CORBA-Konvertierungsmethoden	50
	<i>Die Konvertierungsmethoden von Listen</i>	53
	Die Umwandlung der einzelnen Datentypen	56
	<i>Die Umsetzung der einfachen Datentypen</i>	56
	<i>Die Umsetzung der Strings</i>	57
	<i>Die Umwandlung von Listen</i>	59
	<i>Die Umsetzung von Unions</i>	60
	<i>Die Umsetzung der OPTSTRUCPs</i>	61
	Problemlösungen	62

KAPITEL 9	<i>Der CORBA-Objekt-Adapter</i>	65
	Der CORBA-TUA	66
	<i>Die Klasse TUAgent_oo</i>	66
	<i>Konvertierung der Aufrufe</i>	67
	<i>Der Verbindungsaufbau – prepareConnection</i>	69
	Das Frontend	70
	Die Objekt-Adapter-Klassen	72
	<i>Der allgemeine Objekt-Adapter TRD_OA</i>	72
	<i>Der serverseitige Objekt-Adapter TRD_OA_TSA</i>	73
	Die CORBA-Objekt-Adapter-Klassen	74
KAPITEL 10	<i>Das Java Interface</i>	77
	Die Übersetzung der Schnittstellenbeschreibung	78
	Der JAVA Trader User Agent	81
	Die Java Zugriffsklassen	82
	<i>Die Portierung der Zugriffsklassen</i>	83
	<i>Die Verwendung von Enums</i>	85
	<i>Die Verwendung von Unions</i>	85
	<i>Rückgabe von Strings in Referenzparametern</i>	87
	<i>Die Umsetzung von String-Vergleichen</i>	87
	<i>Abbildung von Listen</i>	88
	<i>Verwendung von Zugriffsmethoden innerhalb von Zugriffsmethoden</i>	90
	Beispielumsetzung für das Java Interface	90
KAPITEL 11	<i>Der Melody Agent</i>	93
	Der Melody Agent	94
	Die Melody Nachrichten	96
	<i>Die Klasse MELODYMessage</i>	96
	<i>Die Klasse MELODYListRequest</i>	98
	<i>Die Klasse MELODYPing</i>	99
	<i>Die Klasse MELODYReply</i>	99
	<i>Die Klasse MELODYListReply</i>	100
	<i>Die Klasse MELODYError</i>	100
	Verwendung der Nachrichtenklassen im Melody Agenten	102
KAPITEL 12	<i>Ausblick</i>	105

Abbildungen

ABBILDUNG 2-1.	Object Management Architecture (OMA)	5
ABBILDUNG 2-2.	Aufbau der OMA	6
ABBILDUNG 2-3.	Prinzip eines ORB Requests	7
ABBILDUNG 2-4.	Schnittstellen zum ORB	7
ABBILDUNG 3-1.	Mole	12
ABBILDUNG 4-1.	Architektur des Traders	15
ABBILDUNG 7-1.	Integration der CORBA-Schnittstelle	28
ABBILDUNG 7-2.	Die Schnittstellenbeschreibung	29

Die Vernetzung der Rechner in der Welt schreitet immer weiter voran. Ein Zeichen dafür ist, daß *Internet* zu einem allgemeinen Schlagwort geworden ist. Ein Anschluß an das *Internet* ist für die Allgemeinheit erschwinglich geworden. Und mit der wachsenden Zahl an privaten Nutzern wächst auch die Zahl der – zum Teil kommerziellen – Anbieter von Diensten unterschiedlichster Art. Dadurch wird das *Internet* immer größer.

Doch schon seit längerem ist dieses Netzwerk unübersichtlich geworden. Die Zahl der Rechner, der Dienste und der erreichbaren Personen ist mittlerweile so groß, daß es nicht mehr möglich ist, einen halbwegs vollständigen und konsistenten Überblick angeschlossener Ressourcen zu bekommen. Im Gegenteil sogar wird eine Suche nach Informationen oder Diensten oftmals sehr beschwerlich, da man kaum wissen kann, wo man sinnvollerweise die Suche beginnen soll oder nach welcher Strategie gefundene Dienste ausgewählt werden sollen.

Vermittlung

Durch diese Entwicklung sind in den letzten Jahren eine Vielzahl von *Meta-Diensten* entstanden, die versuchen eine Übersicht über angebotene Dienste zu erstellen. Zum Beispiel *archie* bietet dem Benutzer die Möglichkeit, Software und allgemeine Daten zu suchen, die auf der großen Zahl von *FTP-Servern* rund um die Welt abgelegt sind, wobei die Anfragen sich allerdings auf recht einfache Suche nach Dateinamen beschränken.

Ein anderes Beispiel ist die Verwendung von *Search Engines* für die Suche nach Seiten auf *WWW-Servern*. Durch die explosionsartige Ausweitung von *WWW* wird einerseits zwar eine sehr große Menge von Informationen angeboten, andererseits ist es sehr schwierig, bestimmte Informationen zu finden. Oftmals ist der Weg zur gesuchten Information beschwerlich, wenn nicht die genaue Adresse der *WWW-Seite* bekannt ist. Auch hier werden verschiedene Dienste (deren Zahl selber langsam unübersichtlich wird) im *Internet* angeboten, die bei der Suche

behilflich sein möchten. Das Spektrum der Suche reicht von einfacher Suche nach Titeln von Seiten bis zur Suche nach Stichworten innerhalb von Seiten.

Diese beiden Beispiele machen deutlich, wie wichtig eine automatisierte Vermittlung von Diensten in Rechnernetzen für einen menschlichen Nutzer ist, der ansonsten völlig überfordert wäre und kaum optimale Entscheidungen treffen könnte. Der vermittelnde Dienst hilft dem Benutzer also auf zwei Weisen: erstens kennt er die Rechner – allgemeiner die Orte –, die den gesuchten Dienst anbieten. Zweitens kann er dem Benutzer bei der Auswahl der Dienste behilflich sein, indem er aus der Gesamtzahl der Dienstanbieter ungeeignete verwirft und die verbleibenden Angebote nach einer Gewichtung sortiert. Er übernimmt also eine Filterfunktion für den Benutzer.

Client/Server

Die bisher beschriebenen Formen des Informationsaustausches zwischen dem Erbringer eines Dienstes und dem Benutzer entsprechen einem *Client-Server-Verhältnis*. In diesen Fällen fragt ein *Client* – in den bisherigen Fällen der menschliche Benutzer – über eine geeignete Schnittstelle bei einem *Server* – das den Dienst erbringende Program – an und bekommt dann die Antworten auf seinem Bildschirm angezeigt.

Dieses Verhältnis zwischen Dienstbringer und Dienstanfrager beschränkt sich allerdings nicht auf Anfragen menschlicher Benutzer, sondern wird oft als grundlegendes Konzept bei der Programmierung von verteilten Anwendungen benutzt. Durch die Aufgabenverteilung zwischen den beteiligten Komponenten wird eine klare, einfache Struktur der Anwendung gefördert, die jedoch auch zum einschränkenden Faktor werden kann. Im Allgemeinen bedienen sich die Anwendungen dazu des *Remote Procedure Calls (RPC)*, um einzelne Funktionen des *Servers* aufzurufen. Während des Aufrufes wird allerdings meist die Bearbeitung beim *Client* angehalten bis schließlich das Ergebnis zurückübertragen wird. Andere Ansätze sind möglich und werden zum Teil auch verwendet.

Für die Zuordnung eines *Servers* zu einem *Client* gibt es verschiedene Strategien. Der einfachste Weg dabei ist, den zu verwendenden Dienstbringer einmal bei der Erstellung der Anwendung fest vorzugeben. Doch eine solche Lösung bietet keinerlei Flexibilität, um auf verschiedene Dienstbringer oder auf Fehlerfälle zu reagieren. Die Verwendung einer Vermittlungskomponente an dieser Stelle bietet sich deshalb an, die zur Laufzeit einen (oder mehrere) Server auswählt. Dazu offerieren die Server ihre Dienste dem Vermittler, indem sie ihm mitteilen, welchen Dienst sie erbringen. Dabei können weitere Angaben – zum Beispiel der Preis der Leistung – gemacht werden. Wird nun ein Dienst gesucht, so wird eine Anfrage an den Vermittler gestellt – eventuell unter Angabe weiterer Randbedingungen, wie zum Beispiel Suche nach einem möglichst günstigen Dienst –, der aus seinen vorhandenen Angeboten einige auswählt und dem Client übergibt.

Generell bleibt aber bei Client/Server immer die Aufgabenverteilung bestehen. Es ist nicht vorgesehen, daß zwei Kommunikationspartner einmal die Rollen tauschen oder ein völlig anderes Verhältnis eingehen. Die Arbeit wird vom *Client* initiiert und vom *Server* ausgeführt. Dazu

kommt meistens noch, daß sie nicht mobil sind, also nicht den Ort ihrer Ausführung wechseln können und damit fest an ihren einmal gewählten Ort gebunden sind.

Objekte

Bei der Verteilung von Aufgaben im *Client/Server*-Modell sind die beteiligten Komponenten meist vollständige Programme, die miteinander in Kontakt treten und über besondere Kommunikationsschnittstellen ihre Daten austauschen. Doch die Arbeitsverteilung läßt sich auch auf tieferer Ebene vorstellen. Ein Ansatz mit einer sehr feinen Granularität ist die Betrachtung von *Objekten*. Dieser Ansatz entsteht aus der objekt-orientierten Programmierung.

Objekte stellen im Prinzip Teile der realen Welt dar, die auf die Ebene der Software abgebildet werden, oder entstehen durch weitere Abstraktion. Ihre Eigenschaften setzen sich aus zwei Teilen zusammen. Die *Attribute* sind die gespeicherten Daten innerhalb eines Objektes und repräsentieren somit seinen aktuellen Zustand. Die *Methoden* sind die Operationen, die auf einem Objekt angewendet werden können, und so den Zustand verändern können.

Durch Kapselung von Daten und Operationen bleibt die konkrete Implementierung des Objektes verborgen und somit kann es unabhängig von anderen Objekten betrachtet werden. Konzeptionell lassen sich zwei Ansätze unterscheiden: einerseits können Objekte als *aktive* Elemente aufgefaßt werden, die durch Nachrichten kommunizieren, aufgrund derer Methoden ausgeführt werden. Andererseits können Objekte auch als passive Elemente dargestellt werden, die Methoden bereitstellen. Letzterer Fall ähnelt dann dem *Client/Server*-Fall.

Objekte sind Datenkapseln. Bedingt durch diese Eigenschaft ist es möglich, Objekte auf verschiedene Knoten eines Netzwerkes zu verteilen und über Nachrichten kommunizieren zu lassen. Doch spätestens mit der Verteilung taucht auch wieder die Vermittlungsproblematik auf: wie findet ein Objekt ein anderes und welches soll gewählt werden, wenn mehrere Objekte eine bestimmte Funktionalität anbieten. Da dies bei der Verteilung auf der Stufe von Objekten ein sehr grundlegendes Problem ist, muß diese Aufgabe vom unterliegenden System übernommen werden. Dies wird noch deutlicher, wenn es den Objekten möglich wird zu migrieren, also den Ort ihrer Ausführung zu verlagern. Nachrichten an migrierte Objekte müssen immer noch korrekt ankommen.

Da Objekte zunächst einmal vom Grundgedanken relativ kleine Einheiten darstellen, bietet sich die Möglichkeit zur Migration auch sehr an. Die Daten, die beim Ortwechsel übertragen werden müssen, halten sich in einem überschaubaren Rahmen im Gegensatz zum *Client/Server*-Modell, wo im Prinzip ganze Programme verlagert werden müssten.

Agenten

Agenten stellen den dritten Ansatz für die Betrachtung von verteilter Bearbeitung in Netzwerken dar. Allerdings muß zunächst einmal festgestellt werden, daß der Bereich der Agenten nicht genau definiert ist. Es existieren verschiedene Definitionen und Kategorisierungen von

Agenten, die es schwierig machen, genau zu beschreiben, was einen Agenten besonders ausmacht. Vom Begriff des Agenten herkommend ist ein Agent ein Stellvertreter für eine andere Person, der für diese bestimmte Aufgaben übernimmt. Dabei ist der Agent zwar an Weisungen des Auftraggebers gebunden, kann aber innerhalb dieses Rahmens selbständig und unabhängig arbeiten.

Diese grobe Beschreibung eines Agenten soll an dieser Stelle einmal ausreichen. Damit lassen sich Agenten grob zwischen Client/Server und Objekte einordnen. Sie stellen zwar meist nur Teilkomponenten einer gesamten Anwendung dar, sind aber für gewöhnlich komplexer als einzelne Objekte aus der objekt-orientierten Programmierung und sind vielleicht aus mehreren Objekten zusammengesetzt.

Eine besondere Stärke der Agenten, die im Rahmen dieser Arbeit interessant sind, ist ihre Mobilität. Da Agenten selbständig eine Aufgabe erledigen sollen, bietet es sich an, ihnen die Möglichkeit zu geben, ihren Ausführungsort zu wechseln. Durch diesen Ansatz wird es Agenten erlaubt, an einen Ort zu wechseln, wo sie ihre Aufgabe am besten erledigen können. Grundsätzlich läßt sich hier die Fähigkeit nennen, zu benötigten Daten im Netzwerk zu migrieren. Als Beispiel kann an einen Agenten gedacht werden, der für seinen Auftraggeber bestimmte Daten aus einer Datenbank im Netzwerk besorgen soll. Statt wie in anderen Ansätzen üblich alle Daten über das Netzwerk zu übertragen und dann erst die interessanten Daten herauszufiltern, wechselt der Agent zur Datenbank, bearbeitet die Daten dort lokal und wechselt wieder zurück. Dadurch kann die Menge der übertragenen Daten drastisch reduziert werden.

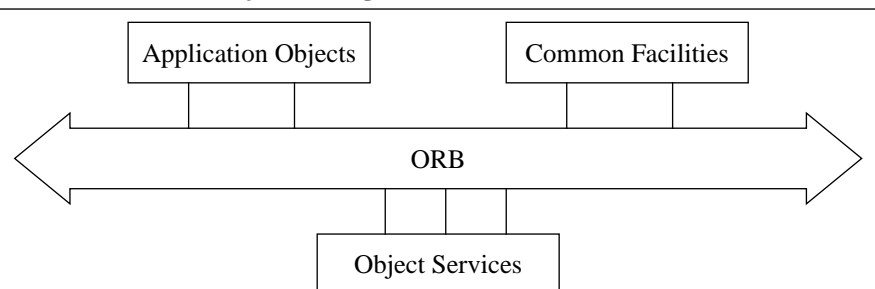
Die große Besonderheit bei Agenten ist die Möglichkeit zu selbständigen Entscheidungen. Ein Agent entscheidet normalerweise selbständig, wann und wohin er migrieren möchte und nicht das unterliegende System, daß vielleicht zum Lastausgleich Verschiebungen vornehmen möchte.

Aber trotz Selbständigkeit und Mobilität hängt ein Agent immer noch von anderen Agenten und Dienstbringern ab. Schließlich ist er für die Erfüllung seiner Aufgabe meist auf Hilfe anderer angewiesen. Gibt es eine große Zahl an Dienstbringern, die oftmals von einer sehr unterschiedlichen Qualität sind, benötigt der Agent Auswahlkriterien, nach denen er seinen Partner auswählen kann. Wenn dagegen in einem anderen Szenario es überhaupt nur sehr wenige Dienstbringer gibt, die einen benötigten Dienst anbieten, dann muß der Agent zunächst einmal einen im Netzwerk finden. Beide Fälle machen deutlich, wie wichtig auch in Agentensystemen ein Vermittlungsdienst ist, um so den Agenten bei solchen zeitraubenden und komplexen Dingen wie der Partnersuche zu helfen und sie teilweise davon zu befreien. Damit kann sich der Agent besser auf seine eigentliche Aufgabe „konzentrieren“, wird also von seiner Programmierung her einfacher.

CORBA – Common Objekt Request Broker Architecture

Die *Object Management Group, Inc. (OMG)* ist eine internationale Organisation, die sich zum Ziel gesetzt hat, objekt-orientierte Technologien zu fördern. Sie hat über 500 Mitglieder, davon zahlreiche Hersteller, Softwarefirmen und Benutzer. Unter anderem sollen Richtlinien und Spezifikationen helfen, einen gemeinsamen Rahmen für die Entwicklung von Anwendungen zu schaffen, um so grundlegende Ziele wie Wiederverwendbarkeit oder Portabilität von objekt-orientierter Software in verteilten, heterogenen Umgebungen zu erreichen. Dazu wurde eine *Object Management Architecture (OMA)* definiert, die die konzeptionelle Grundlage aller weiteren Spezifikationen und Standardisierungsbemühungen von OMG darstellt.

ABBILDUNG 2-1. Object Management Architecture (OMA)

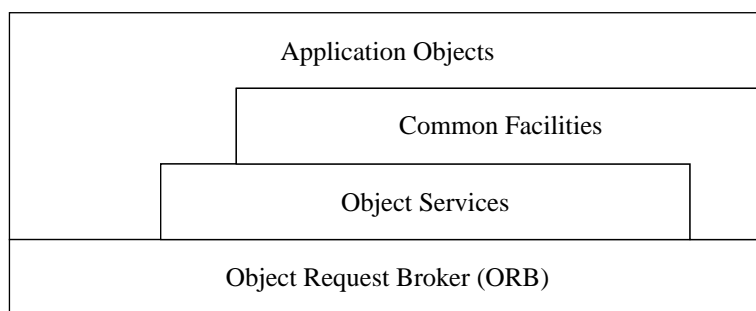


Die OMA besteht aus vier wesentlichen Komponenten:

- *Object Request Broker (ORB)*, dieser ermöglicht es den einzelnen Objekten andere Objekte in verteilten Systemen anzusprechen. Dazu ruft ein Objekt den ORB mit den zu übertragenden Parametern für den Empfänger auf. Der ORB muß nun in der Lage sein, diesen Aufruf – auch zwischen heterogenen Systemen – an ein passendes Empfängerobjekt zu übertragen und dort die richtige Methode aufzurufen. Nach Beendigung des Aufrufs wird das Ergebnis dem initiierten Objekt zurückgegeben, das seine Arbeit wie gewohnt fortsetzt. Der ORB stellt also das Herzstück des gesamten Systems dar.

- *Object Services* definieren die grundlegende Funktionalität für die Erstellung und Benutzung von Objekten wie zum Beispiel die Funktionen, um Objekte zu erstellen, zu kopieren und wieder zu löschen, oder um Beziehungen zwischen Objekten festzulegen. Die Dienste dieser Kategorie müssen von allen Systemen, die die OMA implementieren wollen, angeboten werden.
- *Common Facilities* sind dagegen optional. Die Dienste dieser Kategorie sind nicht so grundlegend wie die Dienste der Object Services, enthalten aber wichtige Komponenten, die von vielen Anwendungen geteilt werden können. Möglichkeiten zur Verarbeitung von elektronischer Post ist nur ein Beispiel für einen solchen Dienst. Durch die Definition dieser Dienste besteht damit ein standardisierter Zugriff unabhängig von der jeweiligen Implementierung oder vom unterliegenden System.
- *Application Objects* sind schließlich die Objekte der jeweiligen Anwendungen und sind somit nicht genauer spezifiziert. Diese Objekte implementieren die Funktionalität der Anwendung und bedienen sich dabei der Dienste der drei anderen Komponenten.

ABBILDUNG 2-2. Aufbau der OMA



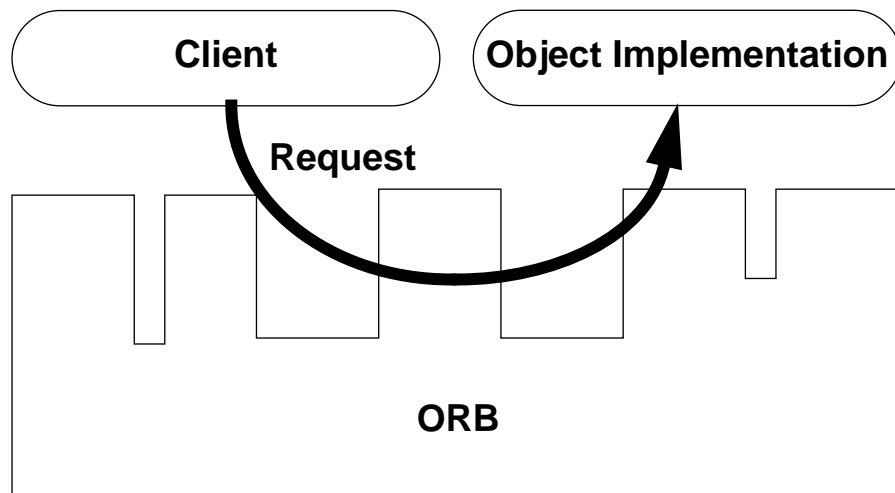
Das Objekt Modell, das CORBA zugrunde liegt, stellt eine einheitliche Darstellung der Objektkonzepte und Terminologie zur Verfügung. Dabei handelt es sich nicht um ein abstraktes Modell, das nicht direkt realisiert ist, sondern um ein konkretes Modell, daß in verschiedener Weise von einem abstrakten Modell abweicht. Es definiert zum Beispiel, wie Parameter übergeben werden oder wie Datentypen spezifiziert sind. Aufgrund dieses Objekt Modells können dann Anwendungen und andere Dienste implementiert werden. Im Einzelnen wird dort beschrieben, was Objekte sind, wie die Aufträge für Objekte aussehen, wie Objekte erzeugt und wieder gelöscht werden können, wie die Datentypen genau aufgebaut sind, wie Attribute, Operationen und Schnittstellen von Objekten spezifiziert werden und das Ausführungsmodell, das einem Objekt zugrunde liegen muß.

Der Object Request Broker (ORB)

Der ORB hat die Aufgabe, Aufträge für Objekte entgegen zu nehmen und sie an das entsprechende Objekt weiterzugeben.

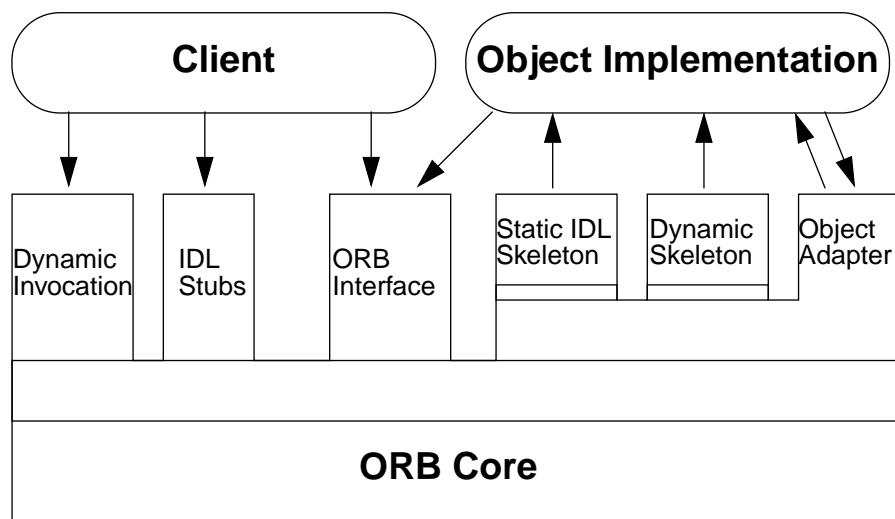
Er ist damit zuständig, eine passende Implementierung des Objektes zu finden, diese Implementierung gegebenenfalls vorzubereiten und die notwendigen, zu übertragenden Daten ent-

ABBILDUNG 2-3. Prinzip eines ORB Requests



sprechend so aufzubereiten, daß sie korrekt empfangen werden können, auch wenn dazwischen Systemgrenzen überschritten werden müssen. Dadurch wird die Schnittstelle, die ein Dienstnutzer sieht, völlig unabhängig vom Ort, wo sich das Objekt befindet, wie es implementiert ist oder sonstigen Aspekten, die nicht die Schnittstelle betreffen. Der ORB muß also in der Lage sein, sowohl mit den Klienten als auch mit der Implementierung eines Objektes zu kommunizieren. Aus diesem Grund gibt es verschiedene Schnittstellen zum ORB, wie sie in Abbildung 2-4 dargestellt sind.

ABBILDUNG 2-4. Schnittstellen zum ORB



Zunächst einmal lassen sich zwei grundsätzliche Arten von Schnittstellen für die Objektaufrufe auf beiden Seiten unterscheiden: die dynamische Schnittstelle und die Schnittstelle über Funktionsrümpfe. Bei letzterer Art wird aus der Schnittstellenbeschreibung eines Objektes für den Klienten ein *IDL Stub* und für die Objekt Implementierung ein *IDL Skeleton* erzeugt. Diese bei-

den Teile verstecken die Details des unterliegenden ORBs, indem sie die Schnittstellen aus der Beschreibung zur Verfügung stellen. Der Klient ruft eine Operation des Stubs im Prinzip genau so auf, wie er sie aufrufen würde, wenn er das Objekt direkt ansprechen könnte. Innerhalb des Stubs werden die Parameter und alle weiteren Informationen für den ORB aufbereitet und ihm schließlich übergeben. Nun überträgt er die Daten zum Empfänger und ruft die Operation durch das zugehörige IDL Skeleton auf. Dieses Skeleton hat die gleiche Aufgabe wie der Stub auf der anderen Seite – nur in umgekehrter Richtung. Er nimmt die Daten vom ORB, zerlegt sie wieder in die ursprünglichen Parameter und ruft nun die Operation der Implementierung auf. Im Prinzip sollte dieser Operationsaufruf gleich sein, egal ob er über den ORB oder lokal erfolgte. Der Ansatz mit IDL Stub bzw. Skeleton erlaubt es also dem Klient bzw. der Objekt Implementierung, die Schnittstelle (mehr oder weniger) unabhängig vom unterliegendem ORB zu verwenden.

Dieser Ansatz ist zwar relativ einfach und bequem, erlaubt aber nur wenig Flexibilität. Aus der Schnittstellenbeschreibung werden Programmteile erzeugt, die fest eingebunden werden. Bei Änderung der Schnittstelle müssen diese Teile wieder neu erzeugt werden. Weiterhin macht dieser Ansatz auch die Zugriffe auf eine Menge von Objekten mit ähnlicher aber nicht gleicher Schnittstelle komplexer. In diesem Fall muß für jedes Objekt ein neuer Stub und ein neues Skeleton eingebunden werden.

Aus diesen Gründen existiert die zweite Art des Operationsaufrufes über die dynamischen Schnittstellen (*dynamic invocation & dynamic skeleton*). Diese beiden Schnittstellen sind für alle Realisierungen von ORBs identisch. Sie bieten den Klienten die Möglichkeit durch verschiedene Aufrufe des *Dynamic Invocation Interfaces* einen Operationsaufruf zusammzusetzen und der Objekt Implementierung auf der anderen Seite den ankommenden Aufruf wieder in seine Einzelteile zu zerlegen und schließlich die passende Operation aufzurufen.

Dadurch wird die Flexibilität der Aufrufe erhöht und es ist möglich, auf sich ändernde Schnittstellen zur Laufzeit einzugehen. Darüber hinaus wird es sogar möglich, völlig unabhängig von einer Schnittstellenbeschreibung zu werden, wenn diese Informationen vom CORBA-System abgefragt werden. Der Preis für diese Flexibilität ist allerdings die Notwendigkeit, alle Daten für den ORB selber aufzubereiten. Damit wird der Programmierer zusätzlich mit diesen Fragen belastet.

Die Entscheidung, ob der statische, aus der Schnittstellenbeschreibung erzeugte oder der dynamische Ansatz gewählt wird, ist für beide Seiten – also sowohl Klient als auch Objekt Implementierung – unabhängig zu treffen. Es ist also auch möglich, daß ein Klient über das Dynamic Invocation Interface auf ein Objekt zugreift, daß seinerseits ein IDL Skelton verwendet. Es ist die Aufgabe des ORBs, alle Aufrufe korrekt zu übermitteln. Wie sich aus der bisherigen Beschreibung vermuten läßt, kann sich ein erzeugter IDL Stub oder ein IDL Skeleton durchaus der dynamischen Schnittstelle bedienen und alle dafür notwendigen Schritte unternehmen. Die erzeugten Programmteile sind dann also Aufsätze auf die dynamischen Schnittstellen.

Die beiden verbleibenden Schnittstellen des ORBs sind das *ORB Interface* und die *Object Adapters*. Das ORB Interface ist eine Schnittstelle, die direkt die Funktionalität des ORBs anspricht. Welche Funktionen angeboten werden, ist für alle ORBs gleich. Da jedoch die meiste Arbeit über andere Schnittstellen erledigt wird, gibt es nur wenige Operationen, die hier angesiedelt sind. Die meisten Zugriffe beschäftigen sich mit Objekt Referenzen, über die

Objekte identifiziert werden können. Sowohl Klienten als auch die Objekt Implementierungen nutzen diese Möglichkeiten gleichermaßen.

Der Hauptweg, durch den eine Objekt Implementierung auf Dienste des ORBs zugreift, ist der Objekt Adapter. Ein Objekt Adapter erzeugt und interpretiert Objektreferenzen, aktiviert und deaktiviert Objekt Implementierungen und einzelne Objekte und bewerkstelligt die Methodenaufrufe der Objekte. Es kann verschiedene Objektadapter geben, die verschiedene Funktionalität bieten, und so für eine gewisse Gruppe von Objekten besonders geeignet sein können. Alle ORBs sollten einen *Basic Objekt Adapter (BOA)* unterstützen, der für die meisten Objekt Implementierungen ausreichend sein sollte.

OMG Interface Definition Language

Die OMG Interface Definition Language (IDL) ist die Sprache, in der die Schnittstellen beschrieben werden, die Klienten aufrufen und die von den Objekt Implementierungen bereit gestellt werden. In dieser Sprache werden alle Datenstrukturen beschrieben, die benötigt werden, sowie alle möglichen Operationen mit ihren Parametern. Es handelt sich aber bei IDL nicht um eine Programmiersprache sondern um eine reine Beschreibungssprache. Die Programmierung der beteiligten Softwarekomponenten erfolgt weiterhin in einer herkömmlichen Programmiersprache. Um eine einheitliche Verwendung der IDL Konstrukte in Programme zu gewährleisten, werden Abbildungen in die jeweilige Programmiersprache definiert. Dadurch können alle Datenstrukturen und Aufrufe passend dargestellt werden und somit auch benutzt werden.

OMG IDL lehnt sich an die lexikalischen Regeln von C++ an, führt aber weitere neue Schlüsselwörter ein. Für eine Beschreibung von OMG IDL sei auf die entsprechende Literatur verwiesen. Die im Zusammenhang mit dieser Arbeit interessanten Konstrukte werden an entsprechender Stelle erläutert, soweit sie sich nicht aus dem Zusammenhang erschließen.

Mole ist ein Agentensystem, das in der Abteilung Verteilte Systeme des Instituts für Verteilte und Parallele Höchstleistungsrechner (IPVR) der Universität Stuttgart entwickelt wird. Es bietet eine Umgebung an, in der die Ausführung der Agenten erfolgt und die die notwendige Systemfunktionalität bereitstellt. Sie bewerkstelligt zum Beispiel die Kommunikation zwischen Agenten oder ermöglicht es den Agenten zu einem anderen Ort zu migrieren.

Agenten in Mole werden (bislang ausschließlich) in *Java* implementiert. *Java* ist für die Verwendung in heterogenen Netzwerken entwickelt worden. Der vom *Java Compiler* erzeugte *Java Bytecode* wird interpretiert und ist deshalb überall dort lauffähig, wo ein Interpreter vorhanden ist. Damit ist ein *Java Programm* portabel einsetzbar. Von dieser Möglichkeit wird in WWW-Browsern rege Gebrauch gemacht, um nicht nur statische Informationen anzuzeigen, sondern auch während der Anzeige Animationen ablaufen zu lassen oder Interaktionen mit dem Benutzer zu erlauben. Dazu wird das entsprechende Programm zum Benutzer übertragen und dort ausgeführt.

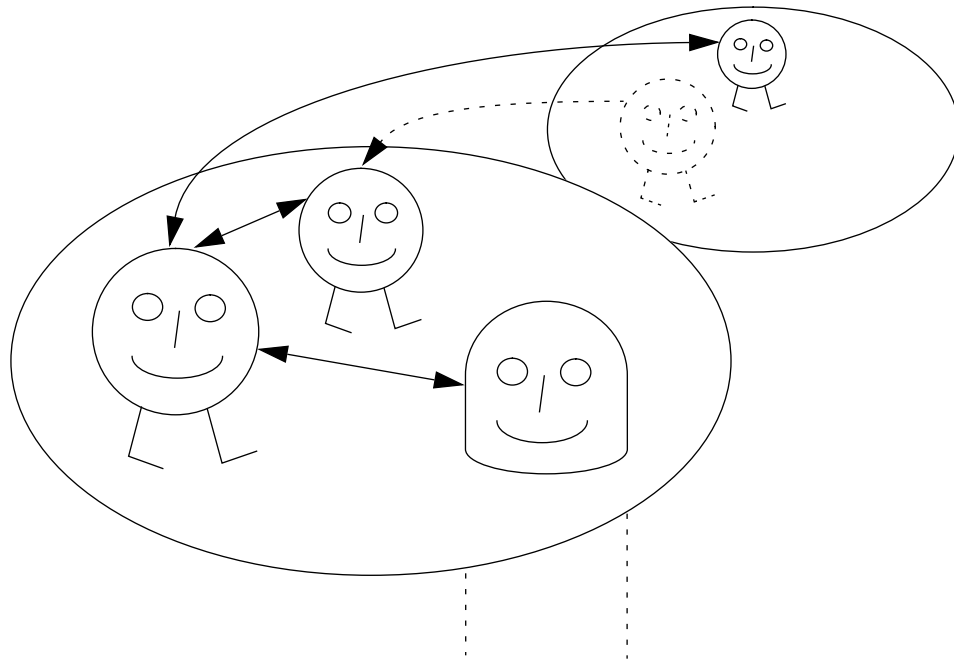
Bedingt durch die Portabilität und die Verwendung in offenen Netzwerken wird auch gleich der zweite wichtige Aspekt von *Java* deutlich: die Sicherheit. Da der Benutzer in seinem WWW-Browser ohne genaue Kontrolle fremde Programme ausführt, muß sichergestellt werden, daß ein Programm keinen Schaden anrichten kann. Deshalb wurde bei der Entwicklung von *Java* Sicherheitsaspekten ein großer Raum bereitgestellt. Diese beiden Eigenschaften von *Java* machen deutlich, daß *Java* als Implementierungssprache für Agenten durchaus geeignet ist.

Ein Agentensystem unter Mole besteht aus dem Ausführungssystem, das die notwendige Umgebung für die Agenten bereitstellt, und den Agenten selber. Die Umgebung der Agenten besteht aus verschiedenen Orten, die auf unterschiedliche Rechner verteilt werden können. Sie stellen die Plätze dar, wo sich die einzelnen Agenten aufhalten, wo sie miteinander kommunizieren und wo sie schließlich auch alle anderen Aufgaben erledigen. Dazu stellt ein Ort die Funktionalität bereit. Aufgaben, die mit einem Ort verbunden sind, sind z.B. die Initialisierung zu Beginn, die Registrierung und Verwaltung der am Ort befindlichen Agenten und die Mechanismen für den Nachrichtenaustausch oder die Migration. Der *lokale Agent* ist ein Systemagent

des Ortes, der das Agentensystem gegenüber den anderen Agenten repräsentiert und so Zugriffsmöglichkeiten auf die Funktionalität des Ortes bietet.

Das unterliegende System implementiert das Agentensystem. Die wichtigste Aufgabe ist natürlich zunächst einmal die Ausführung der Agenten. Es steuert und kontrolliert, wie der Programmcode der Agenten ausgeführt wird. Der Nachrichtenaustausch zwischen Agenten wird bewerkstelligt. Insbesondere bei globalem Austausch zwischen verschiedenen Orten müssen dazu auch weitere Systemdienste benutzt werden.

ABBILDUNG 3-1. Mole



Es gibt zwei verschiedene Arten von Agenten: die *Systemagenten* und die *Benutzeragenten*. Die Systemagenten ermöglichen, wie der Name bereits ahnen läßt, den Zugriff auf Systemdienste außerhalb des Agentensystems. Ein Systemagent kann zum Beispiel den Zugriff auf eine Datenbank anbieten. Damit wird schon deutlich, daß Systemagenten im Hinblick auf das Gesamtsystem Aufgaben erfüllen, die vom Aspekt der Sicherheit besondere Bedeutung haben, da sie Zugriff auf Ressourcen außerhalb des Agentensystems haben müssen, was nicht mehr unter der Kontrolle des Agentensystems stattfindet. Deshalb sind nur Systemagenten dazu in der Lage. Sie stellen ihren Dienst letztendlich unter der Verantwortung des Anbieters eines Ortes zur Verfügung. Aus diesem Grund – ganz abgesehen davon, daß die benötigten Systemressourcen nicht überall verfügbar sind – können Systemagenten nicht migrieren, sondern sind an ihren Ort fest gebunden.

Die zweite Art von Agenten – die *Benutzeragenten* – sind nun die „normalen“ Agenten ohne spezielle Rechte, die sich zur Erfüllung von Aufgaben im Agentensystem aufhalten. Da sie den Ort wechseln können und damit die Herkunft eines Agenten nicht sicher ist, unterliegen sie Restriktionen, durch die die Sicherheit aller Agenten und des Agentensystems gewährleistet werden soll. Sie können damit auch nicht auf Ressourcen außerhalb des Agentensystems

zugreifen, sondern sind bezüglich dieser auf die Hilfe von Systemagenten angewiesen, die diese Dienste anbieten. Benutzeragenten können mit anderen Agenten kommunizieren, können neue Agenten erzeugen und auch zu anderen Orten migrieren.

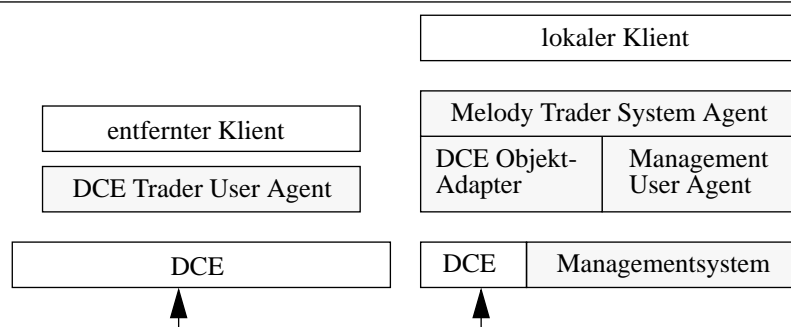
Alle Agenten haben einen einheitlichen Aufbau. Um ihre Aufgaben erfüllen zu können, müssen sie dem Agentensystem einige Methoden anbieten, über die sie angesprochen werden. Dazu gehören zunächst einmal Methoden, über die ein Agent „aktiviert“ und „deaktiviert“ werden kann. Sowohl wenn ein neuer Agent gestartet wird, als auch wenn ein alter Agent nach der Migration an einem Ort ankommt, wird er durch eine bestimmte Methode aufgerufen. Dort kann er alle notwendige Vorbereitungen für die weitere Arbeit treffen. Wenn ein Agent zu einem anderen Ort wechselt oder aus dem Agentensystem entfernt wird, werden weitere Methoden des Agenten aufgerufen, um ihm die Gelegenheit zu geben, notwendige Aufräumarbeiten durchzuführen.

Aber die wohl wichtigste und vermutlich meist benutzte Methode, die ein Agent anbieten muß, ist die Nachrichtenschnittstelle. Dort bekommt er alle Nachrichten übergeben, die an ihn verschickt werden. Er muß diese Nachrichten dann geeignet interpretieren und entsprechend bearbeiten. Für diese Arbeit wird er sich verschiedener Dienste des Ortes bedienen, um zum Beispiel selbst Nachrichten zu verschicken.

Vermittlung von Diensten in großen, offenen, verteilten Systemen ist das Thema, mit dem sich das Projekt *MELODY* beschäftigt. Das Melody-System unterstützt Dienstbringer wie Dienstnutzer bei der Auswahl der geeigneten Partner. Es deckt drei wichtige Bereiche ab: das Management verteilter Dienste – die Grundlage, um die Anwendungen in großen, verteilten Systemen überhaupt verwalten zu können –, die Vermittlung von Diensten von den Erbringern zu den Nutzern und schließlich die Kooperation zwischen Vermittlungskomponenten.

Das Managementsystem von Melody bildet den Grundstock für die Verwaltung von verteilten Diensten. Es ermöglicht den Zugriff auf dynamische Informationen der verschiedenen Teilkomponenten der Anwendung. Dazu wird festgelegt, welche Attribute, Ereignisse oder Aktionen zu diesem Zweck benötigt werden. Das Managementsystem verwaltet diese Teile und stellt sicher, daß der Austausch auch über die Grenzen der einzelnen Rechner hinweg zwischen den relevanten Komponenten funktioniert.

ABBILDUNG 4-1. Architektur des Traders



Der Trading-Dienst von Melody ist nun zuständig für die Vermittlung. Er bekommt die Angebote der Dienstanbieter und verwaltet diese in einer Datenbank. Dabei orientiert sich der Trader

am jeweiligen *Kontext* des Angebots. Durch verschiedene Kontexte wird ein Kontextrraum aufgebaut, der die Menge der Angebote strukturiert. Im Kontext werden die zugehörigen Dienste gespeichert.

Die Unterscheidung der verschiedenen Dienste im System geschieht durch die *Diensttypen*. Der Diensttyp beschreibt die Art des Dienstes. Er schließt Auswahlregeln und weitere Attribute mit ein, die bei der Vermittlung benötigt werden. Durch die Verwendung der Diensttypen werden die Dienste allgemein benutzbar. Die Diensttypen werden dabei hierarchisch in einem azyklischen Graphen verwaltet. Somit ist es möglich von vorhandenen Diensttypen neue Typen abzuleiten oder auch mehrere zu einem neuen zusammenzufassen. Mit diesen Informationen können die vorhandenen Dienstangebote gesammelt und strukturiert abgelegt werden, so daß der weitere Zugriff effizienter und effektiver gestaltet werden kann.

Sucht nun eine Anwendung einen Dienst, so stellt sie eine Anfrage an den Trader, der wiederum aus den vorhandenen Angeboten einen oder mehrere Dienste auswählt. Wichtigstes Kriterium dabei ist natürlich der Diensttyp, der den benötigten Dienst beschreibt. Um die Menge der möglichen Antworten weiter einzuschränken, können weitere Kriterien aufgestellt werden. Nach diesen können verschiedene Angebote geordnet werden. Dadurch kann eine genauere Auswahl der in Frage kommenden Dienste vorgenommen werden. Außerdem können dynamische Attribute berücksichtigt werden, die vom Management bereitgestellt werden. Zusammen mit dem Managementsystem ist der Trader auch in der Lage, die Dienstgüte für den vermittelten Dienst auszuhandeln und entsprechend vom Erbringer reservieren zu lassen.

Der letzte Bereich von Melody betrifft die Koopertation von Tradern. Da in großen und offenen System die Menge der zu verwaltenden Dienste zu groß wird, müssen Zuständigkeitsbereiche für Trader geschaffen werden. Damit entstehen also an verschiedenen Stellen Trader, die teilweise unter recht unterschiedlichen Zielsetzungen geführt werden können. Um nun für einen Trader auch Dienste außerhalb seiner eigenen Domäne zugänglich zu machen, ist Kooperation zwischen Tradern unumgänglich. Das Kooperationsmodul von Melody ist für diesen Teil zuständig.

Klienten können den Trader über eine einheitliche Schnittstelle nutzen: den *Trader User Agent (TUA)*. Dieser bietet alle notwendigen Funktionen an, um Diensttypraum, Kontextrraum und alle Dienstangebote zu verwalten. Für alle Methode des TUAs existieren entsprechende Methoden des *Trader System Agent (TSA)*, dem eigentlichen Trader. Diese Trennung von TUA und TSA erlaubt es, daß vom verwendeten Kommunikationsprotokoll abstrahiert wird und weder Klient noch TSA davon beeinflußt werden. In der Ausgangsversion für diese Studienarbeit, existierte nur die DCE-Schnittstelle, wenn man mal von der Möglichkeit des lokalen Zugriffs auf den TSA absieht. Auf der Seite des Klienten hat sie die Aufgabe, ankommende Aufrufe des TUAs entgegenzunehmen und über DCE zu übertragen. Auf der Seite des TSA übernimmt der DCE Objekt-Adapter alle notwendigen Anpassungen. Er empfängt die Daten für einen Aufruf über DCE und paßt sie so an, daß der Aufruf an den TSA erfolgen kann. Wie zu sehen ist, ändert sich weder für den Klienten noch für den Trader selbst die Schnittstelle.

Melody ist in C++ implementiert, das Agentensystem dagegen in Java. Damit wird deutlich, daß auf dem Weg von Melody ins Agentensystem einige Hürden zu überwinden sind. Nicht nur muß der Programmfluß zwischen beiden Sprachen bewerkstelligt werden, auch müssen zahlreiche komplexe Datenstrukturen übergeben werden.

Generell sind zwei verschiedene Varianten möglich: Zum einen kann daran gedacht werden, Melody direkt über geeignete Mechanismen einzubinden, so daß letztendlich Melody und das Agentensystem in einem Prozeß eine Einheit bilden. Zum anderen können Melody und Mole auch getrennt betrachtet werden. Der Datenaustausch geschieht dann über einen Kommunikationskanal. Dafür kann wiederum ein eigenes Protokoll definiert werden oder es wird auf ein bereits vorhandenes zurückgegriffen. Da die Implementierung eines eigenen Protokolls sehr aufwendig ist, wurde diese Möglichkeit nicht in Betracht gezogen.

Es liegt klar auf der Hand, daß eine direkte Einbindung von Melody gewisse Vorteile in sich birgt. Der Datenaustausch ist schneller und zuverlässiger, weil keine weiteren Komponenten oder Übertragungswege benutzt werden müssen, die eine Verzögerung bewirken oder möglicherweise eine zusätzliche Fehlerquelle darstellen (z.B. bei Ausfall der Verbindung). Andererseits führt dieser Weg aber auch zu möglichen anderen Problemen. Eine direkte Integration bedeutet, daß mehrere recht große Teile vereinigt werden müssen. Bereits Melody selbst stellt ein sehr großes Programmpaket dar. Es benötigt zusätzlich noch einige andere Teile wie zum Beispiel DCE. Java auf der anderen Seite ist ein weiteres umfangreiches Paket, daß zudem als Interpretersystem grundsätzlich anders aufgebaut ist. Schließlich kommt noch das Agentensystem hinzu. Alles gemeinsam bildet ein sehr großes Paket, das möglicherweise aufgrund seiner Größe schwer zu handhaben ist (zum Beispiel bei der Fehlersuche) oder eventuell wegen verschiedener Abhängigkeiten nicht lauffähig ist.

Deshalb hat auch der „Umweg“ über CORBA oder ein anderes Protokoll seine Vorzüge. Die beteiligten Komponenten können als einzelnes betrachtet werden und sind kleiner und damit besser zu handhaben. Durch eine klare Strukturierung von eigentlicher Funktionalität und Kommunikation können alle Teile unabhängig voneinander implementiert und getestet werden.

Abhängigkeiten zwischen Melody und Java bzw. Mole beschränken sich auf den eng umfaßten Teil des Nachrichtenaustausches. Durch die Verwendung von CORBA statt eines selbstentworfenen Protokolls wird darüber hinaus ein standardisiertes Modell zugrunde gelegt. Der Zugriff auf Melody beschränkt sich nicht nur auf das Agentensystem, sondern ist über den ORB jedem zugänglich, der entsprechend der Schnittstellenbeschreibung und des passenden Mappings seiner Programmiersprache CORBA nutzen kann. Die Bereitstellung eines CORBA-Interfaces für Melody wäre also in jeder Hinsicht interessant und wünschenswert.

In den folgenden Kapiteln sollen nun beide Varianten – die direkte und die indirekte Einbindung – näher beleuchtet werden. Zunächst wird darauf eingegangen, wie eine direkte Integration aussehen könnte. Die notwendigen Schritte von Java über C nach C++ werden betrachtet und erläutert. Leider zeigten sich bereits bei den ersten Machbarkeitstests einer Komplettintegration mit allen beteiligten Komponenten große Probleme, an denen dieser Weg schließlich scheiterte.

Deshalb wurde die CORBA-Variante die Wahl dieser Studienarbeit. In den folgenden Kapiteln wird ausführlich auf die einzelnen Schritte eingegangen, die dazu notwendig waren. Es soll nicht nur einfach die vorhandene Implementierung beschrieben, sondern auch der Weg dorthin beleuchtet werden, um so ein besseres Verständnis für spätere Änderungen zu schaffen.

Die direkte Anbindung von Melody nach Java

Die direkte Anbindung erscheint auf den ersten Blick nicht ganz einfach. Melody ist in C++ implementiert. Damit muß ein Mechanismus gefunden werden, wie die Sprachbarriere zwischen C++ und Java überwunden wird und so sowohl der Kontrollfluß als auch der Datenfluß darüber stattfindet. Allerdings kann vereinfachend festgestellt werden, daß für die direkte Anbindung es nicht notwendig ist, von C++ aus Java-Methoden aufzurufen, da der Kontrollfluß immer nur an einer definierten Stelle von Java nach C++ übergehen muß und am Ende wieder zurückkehrt.

Java bietet als Konstrukt, das den Zugriff auf Funktionen außerhalb des Java-Systems ermöglicht, das Schlüsselwort *native* an, mit dem eine Methode deklariert werden kann. Eine solche Methode muß dann nicht notwendigerweise in Java formuliert sein, sondern kann in einer anderen Programmiersprache implementiert sein. Bei der verwendeten Version von Java war C bislang die einzige Sprache, die direkt unterstützt wurde. Andere Sprachen wie C++ sollen folgen. Bis dahin aber muß für die direkte Einbindung von Melody der Umweg über C gewählt werden.

Von Java nach C

Um von Java aus eine Funktion in C aufrufen zu können, müssen zunächst einmal verschiedene Fragen geklärt werden, bevor deutlich wird, wie die Abbildung einer als *native* deklarierten Funktion nach C aussieht:

- Wie geschieht die Benennung der Funktion? Java bietet *Packages*¹ und Klassen zur Modularisierung, während in C alle Funktionen global sind, weshalb die Benennung Eindeutigkeit sicherstellen muß.

1. *Packages* in Java sind eine Sammlung abhängiger Klassen. Sie sind mit einer Klassenbibliothek vergleichbar und bieten die Möglichkeit einer weiteren Strukturierung des Namensraums.

- Wie geschieht der konkrete Aufruf? Wie werden die Parameter übergeben, welche Parameter werden übergeben und wie wird das Resultat wieder zurückgegeben?
- Wie werden die verschiedenen Datentypen übergeben? Es gibt verschiedene Datentypen in Java und in C. Wie werden sie umgesetzt? Was geschieht zum Beispiel mit Klassen?
- Wie wird die geschriebene C-Funktion in Java integriert? Bei Java handelt es sich um eine interpretierte Sprache. Damit muß der geschriebene C-Code zur Laufzeit hinzugeladen werden, wenn nicht der Interpreter selbst verändert werden soll.

Im folgenden soll nun exemplarisch auf diese Fragen eingegangen werden. Für eine vollständige Darstellung sei auf die entsprechende Literatur verwiesen. [13]

Im *Java Development Kit (JDK)* wird zusammen mit dem Interpreter und Bytecode-Compiler ein weiteres Tool namens *javah* ausgeliefert, mit dem die notwendigen Dateien für die Anbindung von C-Funktionen automatisch generiert werden können. Dazu muß zuerst einmal die Klasse, in der eine *native* Methode deklariert wurde, mit dem Java Compiler übersetzt werden. Wenn dies erfolgreich geschehen ist, kann *javah* mit dem Klassennamen aufgerufen werden, analog wie beim Interpreter in der Form „*javah package.classname*“. Dazu muß selbstverständlich die *CLASSPATH* Umgebungsvariable entsprechend gesetzt sein. *javah* erzeugt bei erfolgreichem Aufruf eine Datei „*package_classname.h*“, die Deklarationen für alle *native* Methoden der Klasse enthält. Ein Blick in diese Include-Datei hilft sehr gut dabei festzustellen, wie die implementierten C-Funktionen heißen müssen und welche Parameter sie bekommen.

Deshalb soll hier ein einfaches Beispiel gezeigt werden:

Java: `package melo;`

```
class Third extends Object {
    native void gateInit();
    native void gateTakeint(int a);
    native int gateGiveint();
    native void gateDelete();

    Object dummy;
    int ptr;

    // Hier können die Methoden wie gewohnt verwendet
    // werden.
    Third() { gateInit() }
    protected void finalize() throws Throwable {
        gateDelete();
    }
    void takeInt(int a) { gateTakeint(a); }
    int giveInt() { return gateGiveint(); }
    ...
}
```

Durch Aufruf von „*javah melo.Third*“ wird die Datei „*melo_Third.h*“ erzeugt, die in Teilen hier wiedergegeben wird:

```
C:    typedef struct Classmelo_Third {
        struct Hjava_lang_Object *dummy;
        long ptr;
    } Classmelo_Third;
    HandleTo(melo_Third);

    extern void melo_Third_gateInit(
        struct Hmelo_Third *);
    extern void melo_Third_gateTakeint(
        struct Hmelo_Third *, long);
    extern long melo_Third_gateGiveint(
        struct Hmelo_Third *);
    extern void melo_Third_gateDelete(
        struct Hmelo_Third *);
```

Die erzeugte Datei enthält zwei wesentliche Teile: die Typdefinition und die Funktionsdeklarationen. Im ersten Teil wird ein Abbild der Java-Klasse definiert, die alle Variablen der Klasse enthält. Klassen werden dabei in den Funktionen nicht direkt sondern über sogenannte *Handles* verwendet. Das Makro *HandleTo* erzeugt aus dem übergebenen Namen den Klassenhandle „*struct Hmelo_Third*“. Der zweite Teil zeigt nun, wie die C-Funktionen zu implementieren sind. Alle Methoden einer Klasse erhalten als ersten Parameter immer den *Handle* der Klasse. Die weiteren Parameter entsprechen dann den normalen Parametern in Java, wobei die Datentypen entsprechend angepaßt werden (z.B. aus Java *int* wird C *long*). Gleiches gilt für den Rückgabewert.

Sowohl der Name der Datei als auch der Präfix des Funktionsnamens wird immer aus dem Namen des *Packages* und der Klasse abgeleitet. In Zweifelsfällen hilft immer ein Blick in die von *javah* erzeugte Datei.

Die Implementierung der C-Funktionen kann nun nach Belieben erfolgen. Um auf die Variablen der Klassen oder anderer Java-Objekte, deren *Handle* bekannt ist, zugreifen zu können, gibt es das Makro *unhand*, das einen Zeiger auf die erzeugte Struktur mit den Klassenvariablen enthält. Nach obigem Beispiel geschieht ein Zugriff auf *ptr* etwa so:

```
C:    void
    melo_Third_gateInit(struct Hmelo_Third *this)
    {
        unhand(this)->ptr = ...
    }
```

Der genaue Aufbau der Strukturen kann wieder der erzeugten Include-Datei entnommen werden.

Bevor nun die implementierten C-Funktionen in einem Java-Programm angesprochen werden, sind noch mehrere Schritte notwendig. Ein zusätzliches C-Modul muß erzeugt und eingebunden werden. Dies geschieht durch nochmaligen Aufruf von *javah*, diesmal aber zusätzlich mit der Option „*-stubs*“. Der Aufruf „*javah -stubs melo.Third*“ erzeugt die Datei „*melo_Third.c*“, die alles nötige enthält. Damit sind die Teile für den nächsten Schritt vorhanden.

Da es sich bei Java um eine interpretierte Sprache handelt, können die Funktionen nicht direkt in ein ausführbares Programm hinzugelinkt werden, denn das laufende Programm ist ja immer der Java Interpreter, der nicht verändert werden kann. Deshalb müssen die Funktionen zur Laufzeit dynamisch hinzugeladen werden, wozu die beiden oben erzeugten bzw. erstellten C-Dateien entsprechend für dynamisches Laden übersetzt werden müssen. Dazu muß (zumindest auf Solaris 2.4) beim Übersetzen zusätzlich die Option „-Kpic“ gesetzt werden. Die einzelnen so übersetzten Module können dann wie in folgendem Beispiel zu einer dynamischen Library zusammengefaßt werden.

```
cc -I/usr/local/java/include -c -Kpic \
    melo_Third.c ThirdImpl.c
cc -dy -G -z text -o libthird.so melo_Third.o \
    ThirdImpl.o
```

Damit wird die Library *libthird.so* generiert. Diese Library muß nun im Java Programm explizit geladen werden, bevor das erste Mal auf eine der Funktionen zugegriffen wird. Java bietet dazu in der *System* Klasse die Methode *loadLibrary* an. Am besten wird der Aufruf in den statischen Initialisierungsteil der entsprechenden Klasse gestellt, der ausgeführt wird, sobald die Klasse geladen wird. Dadurch kann sichergestellt werden, daß sich die Library vor dem ersten Methodenaufruf im Speicher befindet. Für obiges Beispiel könnte dies etwa so passieren:

```
Java: class Third ... {
    static {
        System.loadLibrary("third");
        ...
    }
}
```

Die Library wird dabei unter dem Namen „*libthird.so*“ gesucht. Der *Dynamic Loader* unter Solaris sucht seine Libraries immer in den normalen Systempfaden sowie in den Verzeichnissen, die in der Umgebungsvariablen *LD_LIBRARY_PATH* angegeben werden. Damit er sie findet, muß das Verzeichnis, in dem sich die Library befindet, in den Pfad mit aufgenommen werden. Im einfachsten Fall geschieht dies durch Hinzunahme des aktuellen Verzeichnisses etwa wie folgt (csh):

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH} : .
```

Damit sind alle Vorkehrungen getroffen und das Javaprogramm sollte in der Lage sein, die C-Funktionen zur Laufzeit zu laden und dann wie normale Java Methoden aufzurufen. Die gerufenen C-Funktionen unterliegen nicht den Beschränkungen, die normalerweise in Java aus Sicherheitsgründen vorliegen. Durch diese Schnittstelle kann also beliebige Funktionalität eingebunden werden. Es muß allerdings besonders darauf geachtet werden, daß keine der übergebenen Datenstrukturen zerstört werden, weil sonst das Java Laufzeitsystem abstürzen könnte. Es ist deshalb große Sorgfalt angesagt. Damit ist das erste Teilziel der direkten Integretation erreicht. Über die *native* Schnittstelle von Java ist es möglich, C-Funktionen aufzurufen. Nun muß in einem zweiten Schritt noch eine geeignete Abbildung von C nach C++ stattfinden.

Von C nach C++

Für den zweiten Schritt von C nach C++ sind noch einige wenige Dinge zu beachten. Da C und C++ die gleichen Datentypen haben und auch sonst ja C komplett in C++ verwendbar ist, beschränkt sich die Problematik auf die Abbildung von C-Funktionen in entsprechende Methodenaufrufe eines C++-Objektes, da ja das Ziel der kompletten Einbindung ist, ein C++-Objekt – nämlich den *Melody Trader User Agent* – durch ein Java-Objekt mit vergleichbarer Schrittstelle zugänglich zu machen.

C++ bietet die Möglichkeit C Funktionen zu definieren, indem sie in einen Block gestellt werden, dem „extern "C"“ vorangestellt wird. In diesem Block kann auch auf vorhandene C++-Klassen und Methoden zugegriffen werden. So können Interface-Funktionen geschrieben werden, die, von C aus aufgerufen, mit C++ Objekten arbeiten. Durch eine geeignete Benennung von Funktionen für Konstruktor und Destruktor lassen sich alle Zugriffe von C aus bewerkstelligen.

In Fortführung des Beispiels aus dem vorigen Kapitel soll dies hier exemplarisch erläutert werden:

```
C++: class Third {
public:
    Third();
    ~Third();
    void takeint(long);
    long giveint();
};

extern "C" {
    Third *Third_init() { return new Third(); }
    void Third_delete(Third *p) { delete p; }
    void Third_takeint(Third *p, long a)
    {
        p->takeint(a);
    }
    long Third_giveint(Third *p)
    {
        return p->giveint();
    }
}
```

Die vier Interface-Funktionen stellen den Zugriff auf die C++-Klasse her. Da alle Methodenaufrufe sich immer nur auf ein Objekt beziehen können, muß jedesmal der Zeiger auf das Objekt, das angesprochen wird, übergeben werden. *Third_init* legt ein neues Objekt der Klasse an und gibt den Zeiger darauf zurück. *Third_delete* zerstört das Objekt wieder. Die beiden anderen Funktionen reichen die Argumente einfach an die entsprechenden Methoden weiter. Damit kann das Objekt vollständig von C aus genutzt werden.

Von Java nach C++

Als letztes müssen nun die beiden bisherigen Teilschritte zusammengefügt werden. In Java werden für alle Methoden des C++-Objektes *native* Methoden deklariert. Diese Methoden können von Java aus entweder direkt verwendet werden oder nochmals von entsprechenden Java Methoden umfaßt werden. Im Konstruktor des Java Objektes sollte der passende Konstruktoraufufruf des C++-Objektes erfolgen. Analoges sollte in der *finalize* Methode geschehen, um den Speicher wieder freizugeben, wenn ein Java Objekt vom *Garbage Collector* gelöscht wird.

Um eine Zuordnung von einem Java-Objekt zu seinem C++-Pendant festzuhalten, muß im Java-Objekt noch zusätzlich Speicher reserviert werden, in dem der Zeiger auf das C++-Objekt abgelegt werden kann. Im Beispiel wurde dazu die Variable *ptr* angelegt, die die passende Größe hat, einen Zeiger aufzunehmen. Damit fehlt im Beispiel nur noch die Implementierung der *native* Funktionen in C. Sie müssen nur die Parameter weiterreichen und dabei jeweils den Zeiger des C++-Objektes aus der Objektvariablen *ptr* entnehmen.

```
C:   void
      melo_Third_gateInit(struct Hmelo_Third *this)
      {
          unhand(this)->ptr = Third_init();
      }
      void
      melo_Third_gateDelete(struct Hmelo_Third *this)
      {
          Third_delete(unhand(this)->ptr);
      }
      void
      melo_Third_gateTakeint(struct Hmelo_Third *this,
                           long a)
      {
          Third_takeint(unhand(this)->ptr, a);
      }
      long
      melo_Third_gateGiveint(struct Hmelo_Third *this) {
          return Third_giveint(unhand(this)->ptr);
      }
  }
```

Im Beispiel wurden nur Parameter vom Typ *long* verwendet. Bei komplexeren Datentypen und -strukturen wären an diese Stelle noch Konvertierungen nötig, die eine Abbildung in die entsprechenden C++-Strukturen gewährleisten. Da aber die im nächsten Abschnitt beschriebenen Machbarkeitstest zeigten, daß die direkte Einbindung von Melody nicht möglich ist, wurde diese Problematik nicht näher untersucht. Doch läßt sich annehmen, daß, falls alle vom Melody *Trader System Agent* verwendeten Klassen auf entsprechende Java Klassen abgebildet werden, sich diese Problematik auf die Konvertierung der Grundtypen beschränken läßt.

In den letzten Kapiteln wurde gezeigt, wie eine Einbindung eines C++-Objektes nach Java möglich ist und wie sie gestaltet werden kann. Da Java nur eine Schnittstelle nach C anbietet, wurde noch ein zusätzlicher Zwischenschritt eingeschoben, in dem von C aus die Methoden

von C++ aufgerufen werden können. Alles in allem gestaltet sich aber die Durchbindung nicht zu schwierig, wenn man vom Umstand absieht, daß bei Änderungen an den Methoden in C++ sich Änderungen an vier weiteren Stellen – dem *extern "C"*-Block, den *native* Funktionen in C und Java sowie gegebenenfalls den Methoden in Java – nach sich ziehen. Ein Mechanismus zur Automatisierung wäre deshalb wahrscheinlich wünschenswert.

Machbarkeitstests für die Einbindung

Nachdem einer Einbindung auf „sprachlicher Ebene“ nichts mehr im Wege stand, konnte damit begonnen werden zu testen, inwiefern alle beteiligten Komponenten zusammen spielen. Dazu wurde Melody komplett für das dynamische Laden übersetzt und in mehrere *shared Libraries* verteilt.

Für den ersten Test wurde ein altes „Testprogramm“ von Melody *TRD_init* probeweise durch das Agentensystem aufgerufen. *TRD_init* wurde früher benutzt, um in Melody grundlegende Dienstypen bzw. -kontexte zu initialisieren. Mittlerweile ist diese Funktionalität in den Trader mit integriert, doch das Programm ist immer noch für Tests geeignet, ob der Trader angesprochen werden kann.

Um das Programm verwenden zu können, wurde die *main* Funktion umbenannt und das Programm ebenfalls in eine *shared Library* übersetzt. Der Aufruf erfolgte von einem einfachen Agenten aus Mole über eine simple, parameterlose *native* Methode.

Doch bereits dieser erste, noch sehr einfache Testlauf schlug fehl. Das Programm stürzte beim ersten Aufruf von DCE innerhalb des Teils von *TRD_init* mit einem „*assertion failed*“ ab. Eine weitere Analyse des Problems, bei der schrittweise einzelne Komponenten entfernt wurden, brachte Aufschluß über die Ursache des Problems. Schon einer der ersten Schritte brachte eine große Überraschung: bei bloßer Entfernung des Aufrufes von *TRD_init* stürzte das Java Laufzeitsystem trotzdem noch mit einem „*Segmentation fault*“ ab. Das bedeutete, daß schon allein die Anwesenheit von *TRD_init* bzw. eines Teils davon im Hauptspeicher ausreichte, um Java zum Absturz zu bringen. Weitere Reduktion der geladenen *shared Libraries* führte schließlich dazu, daß das Problem in der *Multi-Thread (MT) Library libthread.so* lokalisiert werden konnte. Sobald diese Library von einem Javaprogramm geladen wurde, ohne daß auch nur eine Funktion davon aufgerufen wurde, stürzte es mit einem *Segmentation fault* ab. In einigen Testszenerarien konnte der Absturz in Funktionen aus der Library festgestellt werden, obwohl sie eigentlich nie verwendet wurde!

Eine eingehendere Betrachtung der Symboltabelle des Java Interpreters führte zum Ergebnis, daß Java einige Funktionen verwendet, die in der Dokumentation als nicht für die Verwendung in Multi-Thread Applikationen tauglich eingestuft werden. Weiterhin wurde einige Symbole gefunden, die ebenfalls in der MT Library vorhanden waren. Dies ließ vermuten, daß der Java Interpreter nicht für multi-threaded Anwendungen übersetzt war. Eine Anfrage in der Newsgroup *comp.lang.java* brachte schließlich die Bestätigung, daß die MT Library mit Java unverträglich ist. Java implementiert seine Threads mit einem eigenen Paket, das in Konflikt mit den System Threads steht.

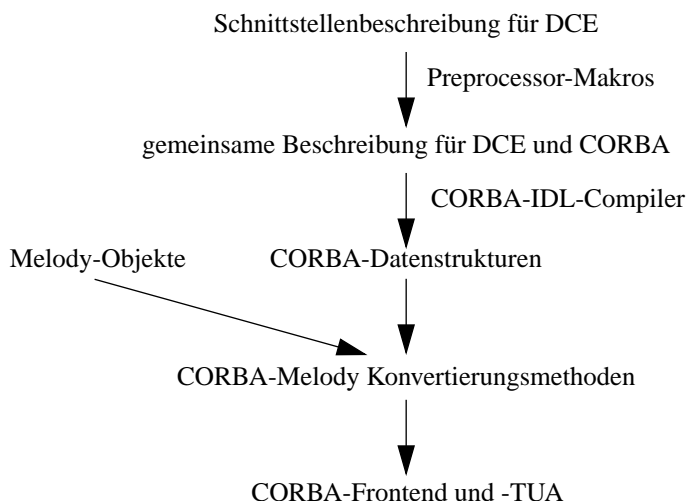
Da aber sowohl Melody als auch DCE von den System Threads regen Gebrauch machen, bedeutete dies das (vorläufige) Ende des Versuchs einer direkten Einbindung. Nach Auskunft von Sun soll in einer späteren Version von Java auch die MT Library unterstützt werden, doch bis dahin mußte ein anderer Weg beschritten werden.

Eine CORBA-Schnittstelle für Melody

Nachdem der Versuch, Melody direkt aus Java anzusprechen, fehlgeschlagen war, mußte nach einem alternativen Weg gesucht werden. Melody benutzte bislang DCE, um mit anderen Anwendungen zu kommunizieren. Da DCE selbst für den Datenaustausch ausschied und ein Herunterbrechen der DCE-Aufrufe auf eine tiefere Ebene extremen Aufwand bedeuten würde, mußte Melody über einen anderen Weg angesprochen werden. Da schon seit längerem geplant war, für Melody eine CORBA-Schnittstelle zu integrieren und von Orbix ein Java-Mapping von CORBA in einer Beta-Version vorlag, wurde diese Variante als nächstes betrachtet. Verschiedene Test zeigten, daß eine Implementierung – trotz einige Probleme mit der Orbix-Software – erfolgreich sein könnte. Die Erfahrungen der vorgeschalteten Test floß in die spätere Realisierung voll ein. Aus diesem Grund soll auf eine ausführliche Erläuterung an dieser Stelle verzichtet werden. Statt dessen soll die tatsächliche Realisierung motiviert und beschrieben werden, um die Hintergründe besser zu erklären und für spätere Änderungen und Erweiterungen Hilfestellungen zu geben.

Die folgenden Kapitel beschreiben die einzelnen Schritte. Zunächst werden die Anpassungen beschrieben, die notwendig waren, um die bereits vorhandene Schnittstellenbeschreibung für DCE auch für CORBA nutzbar zu machen. Darauf aufbauend konnten die Konvertierungsmethoden geschrieben werden, um die CORBA-Strukturen auf die internen Melody-Objekte und umgekehrt abzubilden. Mit der Realisierung des *Trader System Agent(TSA)-Frontends* und des *Trader User Agents(TUA)* für CORBA wird die CORBA-Schnittstelle für C++ vollendet. In den weiteren Kapiteln wird beschrieben, wie Melody über das Java Interface angesprochen werden kann. Diese Informationen bilden die Grundlage für die Implementierung des TUAs unter Java. Zu guter Letzt wird der System Agent für Mole beschrieben, der unter Verwendung des TUAs die Traderanfragen von anderen Agenten entgegennimmt. Mit diesem Schritt ist das Ziel erreicht: Agenten in Mole können über den Systemagenten und CORBA auf Melody zugreifen.

ABBILDUNG 7-1. Integration der CORBA-Schnittstelle



Beschreibung der Trader Schnittstelle in CORBA IDL

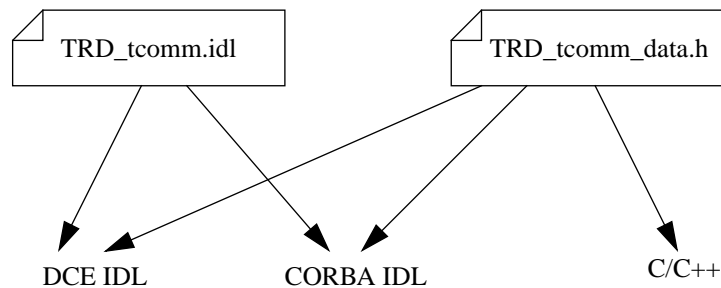
Um ein Objekt über CORBA anzusprechen, muß eine Schnittstellenbeschreibung in CORBA IDL vorliegen, in der steht, welche Methoden mit welchen Parametern aufgerufen werden können. Diese Beschreibung wird verwendet, wenn z.B. die Parameter für die Übertragung codiert und decodiert werden. Ein *IDL Compiler* ist in der Lage, aus dieser Information bereits einige Definitionen und Funktionen zu generieren, die diese Übertragung bewerkstelligen und so den Programmierer von diesen Aufgaben befreit wird.

Melody verwendet bereits für den Datenaustausch DCE. Jetzt sollte also zusätzlich auch CORBA hinzukommen. Beide ähneln sich in Bezug auf ihre Verwendung dahingehend, daß sie beide eine Schnittstelle in einer *IDL (Interface Definition Language)* verwenden, aus der benötigte Datenstrukturen und Funktionen erzeugt werden. Der Unterschied zwischen beiden liegt allerdings in der Syntax und einigen Ausdrücken. DCE IDL ist vollständig an C angelehnt und erlaubt quasi beliebige C Strukturen, u.a. werden auch Zeiger mitberücksichtigt. CORBA baut dagegen auf C++ auf, erweitert die Syntax aber um einige Konstrukte, kennt dafür keine Zeiger und ist an manchen Stellen restriktiver.

Da aber die Syntax trotzdem sehr ähnlich ist und sich die Differenzen relativ problemlos umgehen lassen, wurde versucht beide Schnittstellen zu verbinden. Dadurch ließ sich unnötige Duplizierung ähnlicher Teile vermeiden und Fehlerquellen können so ausgeschlossen werden.

In der zugrunde gelegten Version von Melody bestand die Schnittstellenbeschreibung aus zwei Dateien. *TRD_tcomm.idl* enthielt die Deklarationen der Schnittstellenfunktionen und benutzte die zweite Datei *TRD_tcomm_data.h*, in der die verwendeten Datenstrukturen beschrieben wurden. Diese zweite Datei ist von zentraler Bedeutung, da sie sowohl die Grundlage für die Schnittstellenbeschreibung darstellt als auch in Melody selber benutzt wurde, um aus diesen

ABBILDUNG 7-2. Die Schnittstellenbeschreibung



Datenstrukturen die benötigten Klassen abzuleiten. Durch diese Zusammenfassung wurde ebenfalls unnötige Codeduplizierung vermieden. Entsprechend sollte sich nun auch die CORBA-Beschreibung einpassen. Die Datei dient also in der endgültigen Version als Grundlage für DCE IDL, CORBA IDL und C bzw. C++. Da die Syntax in allen Fällen ähnlich ist, bot sich eine Realisierung mit Preprocessor Makros an, die immer die passenden Konstrukte erzeugen.

Bevor auf die verschiedenen Konstrukte und die entsprechenden Makros eingegangen wird noch ein Wort vorweg zur Benennung der Datenstrukturen. Alle Datenstrukturen heißen sowohl für DCE als auch für C++ immer gleich. Die Implementierung von Melody beruht auf der Tatsache, daß die vom DCE IDL Compiler erzeugten Datenstrukturen genau mit den C Datenstrukturen übereinstimmen¹. Die vom CORBA IDL Compiler erzeugten Datenstrukturen sind dagegen an vielen Stellen verschieden von C, weshalb für diese eine Namenskonvention geschaffen werden mußte, um Konflikte auszuräumen. In der vorliegenden Arbeit wurde zur eindeutigen Benennung der CORBA Strukturen der Suffix „_oo“ gewählt, der an die entsprechenden Namen angehängt wird. Aus der C Struktur *STInsert_struct* wird damit die CORBA-Struktur *STInsert_struct_oo*. Dadurch ist der Name leicht herzuleiten und ebenfalls gut zu erkennen.

Und noch eine Anmerkung zu den Makros: Leider sind nicht alle Preprozessoren für C, C++, CORBA IDL und DCE IDL gleich, obwohl sie in einfacheren Fällen durchaus identisch arbeiten. Doch in einigen Makros war es notwendig, Namen zu konkatenieren, was leider verschieden gelöst werden mußte. Die Konkatenation ist zum Beispiel notwendig, wenn ein Makro an einen übergebenen Namen einen Suffix wie „_oo“ anhängen soll.

Angenommen das Makro „name“ soll dies übernehmen, so soll bei Verwendung von „name(struct)“ als Ergebnis „struct_oo“ erzeugt werden. Eine einfache Definition der folgenden Art ist aber nicht möglich, da der Preprocessor die Makrodefinition wortweise untersucht.

```
#define name(x) x_oo
```

Dieses Makro erzeugt immer „x_oo“ unabhängig vom tatsächlichen Parameter. Um die Konkatenation zu bewerkstelligen, muß also ein Trenner eingeschoben werden, der später aber wieder vollständig verschwindet. Unter C gilt die Regel, daß ein Kommentar „/* ... */“ immer völlig entfernt wird. Dies kann bei der Makrodefinition genutzt werden, indem statt „x_oo“ nun „x/

1. Abgesehen von der zusätzlichen Attributierung mit „[string]“ und ähnlichem, die einfach wegfällt

`**/_oo`“ geschrieben wird. `x` kann erkannt und ersetzt werden, der Kommentar wird später gelöscht.

```
C:    #define name(x) x/**/_oo
```

Diese Methode kann unter C++ nicht benutzt werden, da hier die Kommentare durch ein Leerzeichen ersetzt werden. Dafür ist der Konkatenationsoperator „##“ vorgesehen. Dieser dient bei Ersetzung der Makroparameter als Trennzeichen, wird aber später inklusive vor- oder nachstehenden Leerzeichen gelöscht. Für C++ lautet also die korrekte Definition:

```
C++: #define name(x) x ## _oo
```

Der DCE IDL Compiler selbst benutzt die C Konvention. Abweichend von der CORBA V2.0 Spezifikation und der Aussage des Handbuches von Orbix, das einen vollen C++-Preprocessor verspricht, unterstützte der Orbix IDL Compiler sowohl die C als auch die C++-Konvention. Zu allem Überfluß aber erkannte er das „##“ Token in ignorierten Teilen und meldete dafür Fehlermeldungen und auch sonst hatte er Schwierigkeiten mit mehrzeiligen Makrodefinitionen! Wie dies umgangen werden konnte, wird später beschrieben.

Die Umsetzung

Auf den folgenden Seiten wird erläutert, wie die CORBA-Schnittstellenbeschreibung in die bestehenden Dateien integriert wurde. Diese Erläuterungen beschränken sich dabei auf die tatsächlich benutzten Konstrukte und wollen keineswegs eine allgemeingültige Umsetzung vorstellen. Bei der Umsetzung für C++ oder DCE wurde fast immer exakt die bereits vorhandene Vorlage verwirklicht. Die Korrektheit oder Zweckmäßigkeit dieser Deklarationen wurde deshalb meist nicht betrachtet.

Die Basistypen

Die von Melody benutzten Basistypen ließen sich alle direkt ohne Veränderung nach CORBA übernehmen. CORBA kennt als Basistypen für Ganzzahlen *short* und *long*, die zusätzlich durch Voranstellen von *unsigned* als vorzeichenlos deklariert werden können. Melody verwendet genau diese Typen². Da die Datentypen auf der Zielarchitektur *Solaris* denselben Wertebereich umfassen, ist an dieser Stelle also nichts weiter zu beachten.

Zeichenketten

Es gibt zwei verschiedene Arten von Strings: bei den einen ist eine maximale Länge vorgegeben, die anderen können theoretisch beliebig lang werden. Für beide Arten gibt es verschiedene Deklarationen. Unter C bzw. C++ werden Zeichenketten immer als Zeichen-Array abgelegt. Bei Deklaration der Arraygröße der Strings wird eine maximale Länge festgelegt. Beliebige lang Strings müssen als Haldenobjekte dynamisch angelegt und über Zeiger angesprochen werden.

2. Der Basistyp *char* kam nur im Zusammenhang mit Zeichenketten vor.

DCE benutzt zusätzlich noch das Schlüsselwort „*[string]*“, das vor die C++-Deklaration gestellt wird.

CORBA verwendet eine gänzlich andere, eigene Syntax, die eine komplexere Umsetzung nötig macht. Strings werden durch einen eigenen Typ deklariert, wobei gegebenenfalls noch die maximale Länge spezifiziert werden kann.

Beispiel zu Strings fester maximaler Länge:

```
C++: char text[12];
DCE: [string] char text[12];
CORBA: string<11> text;
```

Da die Form der CORBA-Deklaration sich in der Reihenfolge von Länge und Name unterscheidet, mußte das entsprechende Makro eine komplette Stringdeklaration erzeugen, statt nur einzelne Teile umzusetzen. Als weiterer Punkt muß noch beachtet werden, daß die Länge in der C++ und DCE-Deklaration immer das abschließende Null-Byte mit einschließt, was bei CORBA nicht der Fall ist, weshalb die Länge dort um eins reduziert werden kann.

Das Makro *STRING(name,len)* wird wie folgt definiert³:

```
C++: char name[len]
DCE: [string] char name[len]
CORBA: string<len-1> name
```

Damit kann durch Verwendung von *STRING(text,12)* ein String namens „*text*“ deklariert werden, der maximal 11 Zeichen zzgl. dem abschließenden Null-Byte aufnehmen kann. Da die Umsetzung von den C++/DCE-Grundlagen ausging und alle Konstanten in entsprechender Weise definiert sind, wurde darauf verzichtet, das Makro in einer vielleicht einleuchtenderen Weise ohne Berücksichtigung des Null-Bytes zu definieren.

Für Strings beliebiger Länge sind die Deklarationen zunächst einmal einfacher, da der Längenparameter wegfällt. C++ und DCE verwenden statt der Array-Schreibweise einen Zeiger. Bei CORBA wird die Längenangabe einfach weggelassen.

Beispiel für einen String beliebiger Länge:

```
C++: char *text;
DCE: [string] char *text;
CORBA: string text;
```

Die nur geringen Unterschiede und der immer am Ende stehende Name hätten es erlaubt, ein parameterloses Makro für das Anfangsstück zu schreiben. Aus Konsistenzgründen im Hinblick

3. Das voranstehende „*#define STRING(name,len)*“ wird hier und im Folgenden für eine kompaktere, übersichtlichere Darstellung weggelassen.

auf das *STRING*-Makro und aus Gründen, die im Folgenden ersichtlich werden, wurde aber ebenfalls ein Makro mit dem Namen als Parameter definiert.

Eine genaue Betrachtung der obenstehenden Deklarationen zeigt, daß ihre Mächtigkeit nicht gleich ist. Da C++ und DCE Zeiger verwenden, gibt es einen wichtigen Sonderfall – nämlich den *NULL*-Zeiger –, der in der Deklaration von CORBA nicht berücksichtigt ist. Eine Umsetzung von *NULL*-Strings als String der Länge null würde eine Doppeldeutigkeit im Falle von CORBA bedeuten, die nicht eindeutig aufgelöst werden könnte. Deshalb mußte als zusätzliches Hilfsmittel noch eine *union* verwendet werden.

Das Makro *VARSTRING(name)* wird wie folgt definiert⁴:

```
C++: char *name
DCE: [string] char *name
CORBA: union name/**/_union switch (boolean) { \
        case TRUE: string name;
        case FALSE: boolean dummy;
    } name
```

Da eine CORBA-Union immer einen eigenen Typ deklarieren muß, wurde aus dem Stringnamen der Typname durch Anhängen von „*_union*“ erzeugt. Die Eindeutigkeit bleibt aber gewährleistet, weil der Typ im gleichen Namensraum wie der String selbst angelegt wird. Als Diskriminante reicht ein *boolean* völlig aus. Ist sie auf *TRUE* gesetzt, so ist ein String beliebiger Länge gesetzt. Bei *FALSE* wird ein *NULL*-Zeiger dargestellt. Da unter CORBA eine Variante nicht leer sein kann, mußte ein Dummy-Eintrag vorgenommen werden, der nicht verwendet wird. Damit kann durch *VARSTRING(text)* ein String namens „*text*“ deklariert werden, der eine beliebige Länge hat, oder auch gar nicht definiert ist.

Die Typ-Definition

In *TRD_tcomm_data.h* werden zahlreiche neue Datentypen deklariert, die bei der Übertragung benutzt werden. Beim Großteil von ihnen handelt es sich um Strukturen. Dazu kommen noch einige Aufzählungstypen sowie ein paar wenige Stringtypen und Unions. In diesem Kapitel soll zunächst einmal auf den grundlegenden Rahmen für die Typ-Deklaration der Strukturen und Aufzählungstypen eingegangen werden. Beide sind vom umschließenden Rahmen her gleich, weshalb folgende Ausführungen in gleicher Weise gelten, auch wenn nur Strukturen dargestellt werden.

Zunächst einmal ein Beispiel für eine Typdefinition für C++ oder DCE:

```
typedef struct struc {
    ...
} struc_type;
```

4. Auf die besonderen syntaktischen Gesichtspunkte der *union* in CORBA wird bei ihrer Umsetzung später genauer eingegangen.

Diese Anweisung beschreibt den Aufbau einer neuen Struktur namens „*struc*“, deren genauer Inhalt hier nicht von Interesse ist. Durch die *typedef*-Anweisung wird dieser Struktur gleichzeitig noch ein neuer Typname „*struc_type*“ zugewiesen. Dieser Typname kann nach dieser Definition immer als abkürzende Beschreibung der Struktur dienen. Er ist damit in seiner Bedeutung sowohl unter C++ als auch unter DCE (und auch unter CORBA) gleich. Anders ist dies jedoch beim Namen der Struktur. Unter DCE ist dieser Name nur mit expliziter Angabe von „*struct*“ gültig, d.h. um ihn verwenden zu können, muß „*struct struc*“ geschrieben werden. Die gleiche Schreibweise ist auch unter C++ möglich. Sie ist aber nicht zwingend vorgeschrieben. Dort kann der Strukturname auch selbständig als „*struc*“ auftreten.

Eine wichtige Verwendung dieses Typnamens ist bei der Erstellung von rekursiven Datenstrukturen, wenn also in einer Datenstruktur ein Zeiger auf ein Objekt der gerade definierten Struktur benötigt wird. Verkettete Listen sind hier ein gutes Beispiel. Da eine Struktur immer direkt benannt werden kann, ist manchmal gar keine extra Typdefinition notwendig. Allerdings muß dann auf die unterschiedliche Handhabung zwischen C++ und DCE Rücksicht genommen werden. Andererseits können auch unbenannte Strukturen erstellt werden, in denen nach dem Schlüsselwort *struct* kein Name genannt wird. Stattdessen wird sie nur über den definierten Typnamen benutzt.

Bei CORBA ist die Syntax einer Typdeklaration strikter festgelegt. Sie hat immer eine Form wie im folgenden Beispiel:

```
CORBA:struct struc {  
    ...  
};
```

Analog zu C++ kann der Name „*struc*“ nach seiner Deklaration sowohl mit dem Schlüsselwort *struct* als auch ohne auftauchen. Die CORBA-Syntax erlaubt allerdings keine unbenannten Strukturen, so daß auch bei Typ-Definitionen mit *typedef* dort zusätzlich ein Name definiert wird.

Melody definiert für seine Datenstrukturen eine Reihe von Typnamen, die für unbenannte Strukturen oder Aufzählungstypen verwendet werden, z.B.:

```
typedef struct {  
    ...  
} STInsert_struct;
```

Diese Definition ist für DCE und C++ gleichbedeutend. Doch leider entspricht sie nicht der CORBA Syntax, weil die Struktur unbenannt ist. Eine Struktur muß benannt werden. Doch allein schon die Benennung würde für CORBA ausreichen, um den definierten Namen alleine ohne *struct* verwenden zu können. Und einen zweiten „Dummy“-Namen zu definieren, der nie verwendet wird, würde zu einer unnötigen Flut an Bezeichnern führen, die kaum wünschenswert sein kann.

```
CORBA:struct STInsert_struct {  
    ...  
};
```

Diese Lösung wäre für CORBA und C++ zufriedenstellend. Doch in C bzw. DCE müßte jetzt die Struktur immer mit Schlüsselwort geschrieben werden. Aus diesem Grund ist eine Abweichung von der ursprünglichen Definition nicht praktikabel, möchte man nicht einige Stellen im Programmcode ändern. Und wenn bereits die DCE Definition in der alten Form aufrecht erhalten werden muß, so kann dies auch für C++ geschehen. Damit bleibt für den bestehenden Teil alles gleich, während für den neuen CORBA-Teil ein anderer Aufbau gewählt wird.

Was muß nun für eine allgemeine Umsetzung getan werden? Ein Vergleich der beiden obigen Definitionen gibt darauf bereits eine erste Antwort:

- Der Typname muß bei CORBA vor dem Körper der Struktur kommen, sonst danach, und
- CORBA benötigt keinen *typedef*, weil der Strukturname bereits ausreicht und keine unbenannten Strukturen möglich sind.

Um Fehler bei späteren Änderungen zu vermeiden, wäre es eigentlich wünschenswert gewesen, ein einziges Makro für die gesamte Strukturdefinition zu haben, um so immer den Typnamen korrekt setzen zu können. Doch in Anbetracht der Tatsache, daß der Körper einer Strukturdeklaration beliebig groß werden kann, ist dies nicht möglich. Stattdessen ist eine Aufteilung in mindestens zwei Teile nötig. In der gewählten Realisierung sieht eine endgültige Strukturdeklaration folgendermaßen aus⁵:

```
TYPEDEF struct TC(STInsert_struct) {  
    ...  
} TD(STInsert_struct);
```

Zu beachten ist dabei, daß der Name in den beiden Makros identisch ist und das *typedef* Schlüsselwort ersetzt wurde durch ein gleichnamiges Makro. Nach den bisherigen Bemerkungen sollte klar sein, wie die passenden Makros aussehen müssen.

```
CORBA:#define TYPEDEF /* nichts */  
      #define TC(x)   x/**/_oo  
      #define TD(x)   /* nichts */  
  
sonst:#define TYPEDEF typedef  
      #define TC(x)   /* nichts */  
      #define TD(x)   x
```

CORBA benutzt kein *typedef* und hat seinen Typnamen an erster Stelle, alle anderen nutzen *typedef* und haben den Namen am Ende. Das *TC*-Makro hängt zusätzlich noch für CORBA den Suffix „_oo“ an den Namen, um – wie bereits früher erläutert – Namenskonflikte zu vermeiden.

Die Umwandlung von definierten Typnamen

Die Benennung von Strukturen für CORBA unterscheidet sich, wie bereits mehrfach erwähnt, von den Namen für C++ oder DCE, um Namenskonflikten vorzubeugen. Alle definierten Typen werden für CORBA um den Suffix „_oo“ erweitert. Damit muß bei Verwendung dieser Typen

5. Für Aufzählungstypen kann *enum* statt *struct* geschrieben werden.

auch die Endung korrekt verwendet werden. Um eine Struktur mit Namen „x“ zu benutzen, muß in der CORBA Version „x_oo“ erzeugt werden, während sonst nichts verändert werden muß. Aus diesem Grund wurde das Makro *T* wie folgt eingeführt:

```
CORBA:#define T(x) x/**/_oo
sonst:#define T(x) x
```

Dieses Makro muß für alle Typnamen verwendet werden, die in verschiedenen Versionen vorliegen. Es betrifft damit alle in *TRD_tcomm_data.h* definierten Namen, außer diejenigen Strukturen, die über Zeiger angesprochen werden, was später noch erläutert wird. Ein Beispiel sieht damit etwa so aus:

```
typedef enum TC(takeType) { ... } TD(takeType);
typedef struct TC(...) {
    T(takeType) tType;
} TD(...);
```

Die Definition von Stringtypen

Mit den bisher definierten Makros ist es möglich, auch Typnamen für Strings beliebiger und begrenzter Länge zu definieren. Die notwendigen Anweisungen seien hier gleich an einem Beispiel gegenübergestellt:

```
C++: typedef char APstat[ALENGTH];
DCE: typedef [string] char APstat[ALENGTH];
CORBA:typedef string<ALENGTH-1> Apstat_oo;
```

Es ist schon deutlich zu sehen, daß nach dem typedef wieder Definitionen folgen, wie sie bereits vom *STRING*-Makro erzeugt werden. Einzige Abweichung ist hier: der Typname muß noch für CORBA um den Suffix erweitert werden. Doch dies kann problemlos durch das *T*-Makro erfolgen. Damit sieht eine Stringtypdefinition folgendermaßen aus:

```
typedef STRING(T(APstat),ALENGTH);
```

Definitionen von Strings beliebiger Länge sind analog möglich, wurden aber nicht benötigt.

Die Deklarationen von Listen

Ein wesentliches Element der deklarierten Datenstrukturen in Melody sind Listen. Sie werden an verschiedenen Stellen verwendet. Innerhalb von Melody sind sie in Form von verketteten Listen realisiert. Da auch DCE-IDL Zeiger unterstützt, konnten diese Listen ohne Veränderung übernommen werden. Doch bei CORBA ist dies nicht möglich, denn CORBA kennt keine Zeiger! Auch lassen sich Listen grundsätzlich nicht über rekursive Datenstrukturen verwirklichen. Es ist also ein völlig anderer Weg notwendig. CORBA bietet für Listen das Konstrukt der *sequence* an, was auch für die Umsetzung verwendet wurde. Doch zuvor waren einige Änderungen an den bestehenden Datenstrukturen nötig.

In der Ausgangsversion von Melody sahen Listen etwa wie folgt aus:

```
typedef struct x_r {
    /* Inhalt eines Listenelementes */
    struct x_r *next;
} x_struct, *x_structP;
```

Leider war die Benennung der einzelnen Namen nicht einheitlich, was sich störend auf eine automatische Umwandlung auswirkte. An verschiedenen Stellen hieß zum Beispiel der *next*-Zeiger unterschiedlich. Zum Teil bestand der Inhalt der Liste nur aus einem Zeiger auf eine andere Struktur, zum Teil aus den Einträgen selbst. Bei letzterer Verwendung gestaltete sich zudem die Abbildung auf eine CORBA Sequence schwierig, da in diesem Fall für den Inhalt eine Zwischenstruktur eingeführt werden mußte, was eine zusätzliche Erhöhung der Komplexität einer einheitlichen Umsetzung bedeutet hätte. Aus diesem Grund wurde vor einer Umsetzung auf CORBA der grundsätzliche Aufbau der Listen überarbeitet und vereinheitlicht, so daß heute alle Listen den gleichen Aufbau haben:

```
typedef struct {
    /* Inhalt eines Elements */
} x_el_struct;
typedef struct x_r {
    x_el_struct elem;
    struct x_r *next_el;
} x_struct;
typedef x_struct *x_structP;
```

Dadurch wurde Verschiedenes erreicht:

- Der Inhalt der Liste wird in einer getrennten Struktur definiert und ist am Namen mit der Endung „*_el_struct*“ erkennbar und kann der entsprechenden Liste zugeordnet werden.
- Die Listenstruktur selber hat einen einheitlichen Aufbau und besteht nur noch aus zwei Teilen: dem Feld für die Inhaltsstruktur und in Zeiger auf den nächsten Eintrag.
- Der Name der Liste ist einheitlich festgelegt und aus dem Basisnamen ableitbar.
- Insgesamt entsteht also ein klares, einheitliches Namensschema für Listen.
- Durch die Trennung des Inhalts ist eine gute Abbildung auf CORBA möglich. Sie stellt sich jetzt denkbar einfach dar, wenn die Inhaltsstruktur wurde gemäß der bisherigen Beschreibung umgesetzt wurde:

```
CORBA:typedef sequence<x_el_struct_oo> x_struct_oo;
```

Wie jetzt bereits erkennbar ist, zerfällt der Aufbau einer Liste in der endgültigen Form in zwei Teile: Zunächst wird der Inhalt der Liste in einer Struktur mit dem entsprechenden Namen deklariert. Dies geschieht in der normalen Art und Weise, wie sie in diesem Kapitel beschrieben wird. Der Rest der Listendeklaration ist völlig einheitlich gelöst und läßt sich deshalb hervorragend durch ein einziges Makro realisieren! Die Definition des Makros *STRUCLIST(x)* sollte einfach zu verstehen sein:

```
C/DCE:#define STRUCLIST(x) \
    typedef struct x/**/_r { \
        x/**/_el_struct elem; \
        struct x/**/_r *next_el; \
```

```
        } x/**/_struc; \
        typedef x/**/_struc *x/**/_strucP
C++: #define STRUCLIST(x) \
        typedef struct x ## _r { \
            x ## _el_struc elem; \
            struct x ## _r *next_el; \
        } x ## _struc; \
        typedef x ## _struc *x ## _strucP
CORBA #define STRUCLIST(x) \
        typedef sequence<x/**/_el_struc_oo> \
            x/**/_struc_oo
```

Wegen der unterschiedlichen Art der Konkatenation muß zwischen C und C++ unterschieden werden, obwohl das Ergebnis nachher identisch ist! Mit diesem Makro und den vorher eingeführten stellt sich eine Deklaration einer Liste denkbar einfach und kompakt dar:

```
/* Deklaration des Inhalts in üblicher Form */
TYPEDEF struct TC(mesel_el_struc) {
    STRING(message, ERRMESLENGTH);
} TD(mesel_el_struc);

/* Die Liste selbst */
STRUCLIST(mesel);
```

Der Preis für die Einfachheit und Einheitlichkeit der Listendeklarationen ist, daß sich die Schreibweise sehr kompakt darstellt und es deshalb immer bewußt sein muß, daß sich hinter „*STRUCLIST(mesel)*“ die Deklarationen einer verketteten Liste mit den Felder *elem* und *next_el* verbirgt, so wie der Definition eines entsprechenden Zeigertyps. Nach kurzer Gewöhnung sollte aber das dahintersteckende Konzept verständlich sein. Darüberhinaus ist jede Liste genau in dieser Form aufgebaut. Dadurch sollte sich die Verwendung schnell vertraut sein, so daß die Vorteile dieser Regelung überwiegen.

Benutzung von Strukturzeigern

Für alle deklarierten Strukturen wird für C++ und DCE ein zusätzlicher Zeigertyp definiert, der sich aus dem ursprünglichen Namen durch Anhängen eines „*P*“ ableitet. Dieser Zeigertyp wird wieder in den Deklarationen anderer Strukturen verwendet. CORBA jedoch kennt keine Zeiger, weshalb die entsprechenden Zeigertypen nicht existieren. Für CORBA wurde deshalb die Struktur selbst statt eines Zeigers verwendet. Die Alternative, für CORBA den Zeigertypnamen als Synonym für die ursprüngliche Struktur zu definieren – dann könnte die Struktur immer über den Zeigertypnamen eingebunden werden –, wurde verworfen, weil dann erstens wieder eine Großzahl redundanter Namen definiert würde, zweitens das Bewußtsein verloren ginge, daß es sich bei CORBA nicht um einen Zeiger sondern um die Struktur selbst handelt und zu guter Letzt der CORBA-Suffix sowieso eine Makroverwendung nötig macht. Deshalb wurden für die Strukturzeiger ebenfalls neue Makros definiert.

Im einfachsten Fall wurde in Melody ein Strukturzeiger wie folgt benutzt:

```
typedef struct {
    ...
    DescriptionNode_structP sTDescriptionList;
    ...
} STInsert_struct;
```

Diese Deklaration sollte für DCE und C++ natürlich beibehalten werden, während für CORBA das letzte „P“ entfernt und stattdessen ein „_oo“ angehängt werden muß. Dies läßt sich durch ein sehr einfaches Makro *STRUCP(x)* erreichen:

```
C/DCE:#define STRUCP(x) x/**/P
C++: #define STRUCP(x) x ## P
CORBA:#define STRUCP(x) x/**/_oo
```

Doch genauso wie bei den unbegrenzten Strings NULL-Zeiger für COBRA nicht direkt modelliert werden konnte so auch hier. Ein übergebener NULL-Zeiger muß anders abgebildet werden. Doch trotzdem hat das *STRUCP*-Makro seine Berechtigung. Tatsächlich läßt es sich an vielen Stellen einsetzen: nämlich immer dann, wenn nicht auf eine einfache Struktur verwiesen wird, sondern auf ein Konstrukt, daß in der CORBA Abbildung auch den NULL-Zeiger schon einschließt. Dies ist bei Listen immer der Fall, denn ein NULL-Zeiger entspricht einer leeren Liste, die auch in CORBA dargestellt werden kann. In Fortführung des obigen Beispiels sieht eine Anwendung von *STRUCP* folgendermaßen aus:

```
TYPEDEF struct TC(STInsert_struct) {
    ...
    STRUCP(DescriptionNode_struct) sTDescriptionList;
} TD(STInsert_struct);
```

Dies reicht für einen Zeiger auf eine Liste völlig aus. Für den Fall, daß die Verwendung von *STRUCP* nicht mehr ausreicht, muß für CORBA ein anderer Weg beschritten werden. Ähnlich wie bei den unbegrenzten Strings wird wieder eine Union benutzt, die entweder die Struktur enthält oder leer bleibt. Um wieder Namenskonflikten vorzubeugen, mußte das entsprechende Makro aber anders aufgebaut werden, da aus dem Namen des Feldes der Name der Union abgeleitet werden sollte. Das Makro erzeugt deshalb eine vollständige Deklaration statt nur die Typbezeichnung im Fall von *STRUCP*, und bekommt sowohl den Namen als auch den Typ als Parameter. Die Realisierung ist wieder ähnlich wie bei den Strings. Das Makro *OPT-STRUCP(x,name)* ist folgendermaßen definiert:

```
DCE/C:#define OPTSTRUCP(x,name) x/**/P name
C++: #define OPTSTRUCP(x,name) x ## P name
CORBA:#define OPTSTRUCP(x,name) \
    union name/**/_union switch (boolean) {
        case TRUE: x/**/_oo name;
        case FALSE: boolean dummy;
    } name
```

Bei der Diskriminante TRUE ist die entsprechende Struktur gespeichert, sonst nur ein Dummy, da eine leere Variante nicht möglich ist. Der Name der Struktur in der Union wurde gleich

gewählt wie der ursprüngliche Name, um den Inhalt besser erkennbar zu machen. Ein Beispiel für die Verwendung sieht so aus:

```
TYPEDEF struct TC(valueEl_el_struct) {
    OPTSTRUCP(Value_struct, value);
} TD(valueEl_el_struct);
```

Das dritte und letzte Makro im Zusammenhang mit Strukturzeigern ist *STRUCP_PTR*. Es wird ausschließlich in *TRD_tcomm.idl* benutzt. Bei der Definition der Schnittstellenfunktionen mußten in Melody für die *OUT*-Parameter Zeiger auf die Strukturzeiger benutzt werden, um das Ergebnis zurückgeben zu können. Für CORBA wurde hier wieder direkt die Struktur verwendet. Der passende Mechanismus für die Datenrückgabe wird dann durch das entsprechende Mapping definiert. Da an dieser Stelle keine NULL-Zeiger übergeben wurden, stellt sich die Realisierung wieder äußerst einfach dar:

```
DCE/C: #define STRUCP_PTR(x) x/**/P *
C++: #define STRUCP_PTR(x) x ## P *
CORBA: #define STRUCP_PTR(x) x/**/_oo
```

Ein Beispiel für die Verwendung ist bei der Beschreibung der Deklaration der Schnittstellenfunktionen zu sehen.

Die Unions

Die Unions sind die letzten Datenstrukturen, die in der Liste der Umsetzung noch fehlen. Insgesamt werden in *TRD_tcomm_data.h* nur zwei Unions definiert, beide dazu lokal innerhalb einer anderen Struktur. Da sich zudem die Syntax zwischen DCE, COBRA und C++ in größerem Umfang unterscheidet, wurde auf eine Umsetzung mittels Makros verzichtet und alle drei Teile wurden getrennt angegeben. Die bereits vorgegebenen Teile für DCE und C++ sahen folgendermaßen aus⁶:

```
DCE: union switch (Value_Template value_template) u {
    case YES: ALT_struct alt;
    case NO: ALO_struct alo;
} value_el;
```

value_template ist der Name der Diskriminante. Der Name *u* wird zur Benennung des Uniontyps verwendet. *value_el* ist der Feldname für die Union innerhalb der umschließenden Strukturdeklaration. Die Umsetzung für C bzw. C++ entsteht aus dem DCE-IDL-Compiler durch Übersetzung der obenstehenden Deklarationen.

```
C/C++: struct {
    Value_Template value_template;
    union {
        ALT_struct alt;
        ALO_struct alo;
    };
};
```

6. *Value_Template* ist ein *enum* mit *YES* und *NO*.

```
    } u;  
  } value_el;
```

Der Aufbau hat sich leicht verändert. *value_el* ist eine Struktur geworden, die zwei Objekte enthält: die Diskriminante und eine unbenannte Union, die über den Feldnamen *u* angesprochen wird. Da eine C Union nicht diskriminiert ist, fällt die weitere Unterscheidung weg.

Für CORBA wurde eine Umsetzung in Anlehnung an DCE gewählt, da dort die Syntax ähnlich ist. Bei CORBA wird in einer Union nur der Typ der Diskriminante angegeben, ein Name für sie kann nicht gesetzt werden. Als zweiter Unterschied zu DCE ist die Position für den Namen der Union verschieden. Die verwendete Umsetzung sieht folgendermaßen aus:

```
CORBA:union u switch (Value_Template_oo) {  
    case YES: ALT_struct_oo alt;  
    case NO:  ALO_struct_oo alo;  
  } value_el;
```

Die Benennung der CORBA-Strukturen mit dem Suffix wurde direkt berücksichtigt.

Die Deklaration der Operationen

Der letzte Teil, der noch umgesetzt werden muß, ist die Deklaration der Operationen für DCE und CORBA. Dieser Teil findet sich in der Datei *TRD_tcomm.idl* und betrifft nicht mehr C++. Da die meisten Deklarationen die gleiche Form haben, soll die Umsetzung hier gleich durch ein Beispiel eingeführt werden. Zunächst einmal die vorgegebene Deklaration für DCE:

```
DCE:  TSError LISTTYPE(  
    [in]  STList_structP      listType,  
    [out] STListReturn_structP sTLRP,  
    [out] ErrorMessage_structP *errmsg  
  );
```

Das Attribut „*[in]*“ gibt an, daß im entsprechenden Parameter Daten an die Operation übergeben werden, während in den mit „*[out]*“ markierten Parameter Daten von der Operation zurückgegeben werden. Bei der Umsetzung nach CORBA sind verschiedene Dinge zu verändern. Zunächst einmal gibt es keine Zeiger und auch die Zeigertypnamen sind nicht definiert. Sie werden komplett durch die Strukturen selbst ersetzt, wobei noch der CORBA-Suffix berücksichtigt wird. Diese Umbenennung ist auch beim Rückgabewert notwendig. Als letzter Unterschied werden die Parameter statt mit „*[in]*“ bzw. „*[out]*“ mit „*in*“ bzw. „*out*“ markiert. Dies ergibt folgende CORBA Deklaration:

```
CORBA:TSResult_oo LISTTYPE(  
    in  STList_struct_oo      listType,  
    out STListReturn_struct_oo sTLRP,  
    out ErrorMessage_struct_oo errmsg  
  );
```

Für die meisten Teile sind bereits Makros definiert, die hier direkt verwendet werden können. Für den Rückgabewert kann das Makro *T(x)* den richtigen Namen erzeugen. Da für die Zeiger

keine NULL-Zeiger verwendet werden, kann das Makro *STRUCP(x)* bzw. *STRUCP_PTR(x)* zum Tragen kommen. Was noch fehlt, ist die Veränderung der Markierungen der Parameter als *in* oder *out*. Dies kann durch zwei einfache Makros geschehen, so daß das endgültige Ergebnis etwa so aussieht:

```
T(TSResult) LISTTYPE(  
    IN_ARG  STRUCP(STList_struct) listType,  
    OUT_ARG STRUCP_PTR(STListReturn_struct) sTLRP,  
    OUT_ARG STRUCP_PTR(ErrMessage_struct) errmsg  
);
```

Wobei das Makro *IN_ARG* entweder als „*[in]*“ für DCE oder „*in*“ für CORBA definiert wird, und *OUT_ARG* analog.

Damit ist die Umsetzung der Datenstrukturen und der Operationsdeklarationen in IDL abgeschlossen. Durch den Einsatz mehrerer Makros ist es möglich, aus einer einzigen Datei drei verschiedene Versionen zu erzeugen: für den C++-Compiler, für den DCE-IDL-Compiler und für den CORBA-IDL-Compiler. Durch diese Makros entspricht zwar das Erscheinungsbild in einigen Teilen nicht mehr der gewohnten C-Syntax, dafür werden alle Deklarationen – abgesehen von den Unions – immer nur ein einziges Mal beschrieben. Dadurch entfällt der sonst notwendige Abgleich bei Änderungen und eine Quelle für Inkonsistenzen und Fehler wird vermieden.

Andere Ansätze für die Zusammenfassung wären denkbar gewesen. Ein schon recht simpler Parser hätte eine automatische Umsetzung ebenfalls vornehmen und die entsprechenden Versionen erzeugen können. Die Lösung mit Makros läßt sich allerdings schneller realisieren und ist von der späteren Nutzung her auch einfacher. Es muß keine zusätzliche Datei erzeugt werden, da die korrekte Version durch einen Lauf vom Preprocessor vor der Übersetzung erzeugt wird. Dazu kommt noch, daß die Umsetzung durch den Preprocessor wesentlich schneller ist als ein zusätzlicher Übersetzerlauf.

Die Übersetzung der CORBA-IDL

Mit der Erstellung der Schnittstellenbeschreibung von Melody in CORBA-IDL ist ein wichtiger erster Schritt geschafft. Diese Schnittstellenbeschreibung wird nun dazu genutzt, um mit Hilfe eines IDL-Compilers alle notwendigen Datenstrukturen, Definitionen sowie die Stub-Routinen für Client- und Serverseite entsprechend des C++-Mappings zu erzeugen. Für diese Studienarbeit wurde die Software von *Orbix* benutzt, die einen ORB und einen IDL-Compiler gemäß CORBA-Spezifikation implementiert.

Der Orbix IDL-Compiler liest die Schnittstellenbeschreibung und erzeugt daraus standardmäßig 3 Dateien mit folgenden Namen: *TRD_tcommC.cc*, *TRD_tcommS.cc* und *TRD_tcomm.hh*. In *TRD_tcomm.hh* finden sich die Umsetzungen aller definierten Datenstrukturen. Sie ist damit für die im nächsten Kapitel beschriebenen Konvertierungsmethoden von zentraler Bedeutung, da sie den Ausgangspunkt der Überlegungen darstellt. Die beiden anderen Dateien enthalten die Stubs bzw. die Skeletons für Client- und Serverseite, unter anderem auch die Signaturen der

Schnittstellenmethoden, die an der Nahtstelle zwischen CORBA und Melody verwendet werden.

Leider gab es bei der Verwendung des Orbix IDL-Compilers einige Probleme, die einige zusätzliche Maßnahmen erforderlich machten. Der Preprocessor von Orbix hatte einige Schwierigkeiten mit den Makrodefinitionen, wie sie in den vorigen Seiten eingeführt wurden. Er war nicht in der Lage, mehrzeilige Makrodefinitionen korrekt zu interpretieren. Darüber hinaus meldete er immer Fehlermeldungen für den Konkatenationsoperator „##“, wenn er in Teilen auftrat, die durch ein umschließendes „*#ifdef ... #endif*“ eigentlich ignoriert werden sollten. Zu guter Letzt stellte sich heraus, daß der Preprocessor abstürzt, wenn der Konkatenationsoperator in den Makrodefinitionen für CORBA enthalten ist. Dieser letzte Punkt ließ sich noch durch Verwendung der C-Konvention für die Konkatenation umgehen, da der IDL-Preprocessor – abweichend vom ANSI Standard – einen normalen Kommentar nicht durch ein Leerzeichen ersetzt, sondern völlig entfernte.

Da sich die Verwendung des Operators für C++ aber nur schwer gänzlich umgehen ließ (der C++-Compiler entsprach ja dem ANSI Standard!) und außerdem mehrzeilige Makros auf jeden Fall zur Übersichtlichkeit und besseren Lesbarkeit beitragen, mußte eine andere Lösung gefunden werden. Aus diesem Grunde wurde vor der eigentlichen Übersetzung mit dem IDL-Compiler ein Durchlauf mit dem Standard-Preprocessor vorgeschoben, der diese Probleme nicht hat.

Die notwendigen Aufrufe mit den entsprechenden Parametern sind beide in das *Makefile* von Melody aufgenommen, so daß dieser Schritt normalerweise nicht auffallen sollte. Zur Vollständigkeit sei hier aber trotzdem etwas genauer darauf eingegangen. Die beiden Befehle, die momentan im Makefile verwendet werden, sind die folgenden:

```
/usr/ccs/lib/cpp -C -P -B -D__IDL -D__CORBA_IDL__
-DIDL_CMD TRD_tcomm.idl sun4.SunOS5/TRD_tcomm.idl

/opt/Orbix_2.0/bin/idl -c C.C -s S.C -B -N -R
-d melody_if -D__CORBA_IDL__ -DIDL_CMD sun4.SunOS5/
TRD_tcomm.idl
```

Mit dem ersten Kommando wird der Standard-Preprocessor für C aufgerufen. Da für die Konkatenation bereits, wie oben erläutert, die C Konvention verwendet wird, bereitet dieser Umstand keine weiteren Probleme. Die Definitionen der drei Symbole *__IDL*, *__CORBA_IDL__* und *IDL_CMD* steuern die Auswahl der benötigten Teile in der Schnittstellenbeschreibung so, daß die CORBA-Version erzeugt wird. Ersteres wird in den Dateien nicht verwendet und wurde nur zur Vollständigkeit mit angegeben. Es wird standardmäßig vom IDL-Preprocessor selbst vordefiniert und wurde einfach übernommen.

Die Ausgabe dieser Preprocessorläufe wird in das Unterverzeichnis geschrieben, in dem alle anderen übersetzten oder generierten Dateien von Melody auch gespeichert werden. Dadurch kann ein Umkopieren oder Umbenennen der Datei, die dem IDL-Compiler übergeben wird, vermieden werden. Insbesondere die umgangene Umbenennung ist von großem Vorteil, weil vom IDL-Compiler eine Kopie der übersetzten Datei im *Interface Repository* unter gleichen Namen angelegt wird. Wenn diese anders heißen würde, so wäre das Wiederauffinden und die korrekte Zuordnung möglicherweise erschwert.

Die weiteren Optionen für den Preprocessor beeinflussen ihn so, daß das Ergebnis aus dem Lauf direkt im Compiler verwendet werden kann. Mit der Option „-B“ wird die Unterstützung von einzeiligen C++-Kommentaren mit „//“ eingeschaltet, da sie sonst Fehlermeldungen verursachen würden. Durch „-C“ wird er angewiesen, alle Kommentare nicht herauszulöschen, sondern ungefiltert wieder auszugeben. Damit gehen durch den Preprocessor die Kommentare nicht verloren. Die Bewandnis dieser Einstellung hat einen einfachen Grund: Der IDL-Compiler kopiert die Datei später in sein *Interface Repository*. Die Kommentare können dort später vielleicht helfen, die einzelnen Definitionen besser zu verstehen. Die letzte Option „-P“ gibt an, daß in der Ausgabe die sonst üblichen Zeilenkontrollinformationen nicht mit ausgegeben werden sollen, da diese beim folgenden Compileraufruf nicht verstanden werden. Dadurch geht zwar leider die Zuordnung der Fehlermeldungen zu den Zeilennummern in den Originaldateien verloren, sie ist aber über die erzeugte Zwischendatei immer noch indirekt möglich.

Die, durch diese „Vorbehandlung“ erzeugte Datei wird direkt als Eingabe im IDL Compiler verwendet, der damit von allen Aufgabe der Makro-Erweiterung befreit wurde. Wie bereits aus dem Aufruf ersichtlich ist, werden für den Compilerlauf nochmals dieselben Symbole wie im Schritt zuvor definiert. Diese Definitionen sind natürlich genaugenommen überflüssig, da in der Zwischendatei ja bereits alle Definitionen ausgewertet und damit überhaupt nicht mehr vorhanden sind. Doch um später bei einem verbesserten IDL-Preprocessor nicht alle Argumente wieder neu einfügen zu müssen, wurden die Definitionen trotzdem beibehalten..

Die beiden Option „-c“ und „-s“ werden benutzt, um die Dateiendungen für die erzeugten Dateien zu ändern. Standardmäßig verwendet der IDL-Compiler immer die Endung „.cc“. Da diese Endung aber nicht der sonst in Melody üblichen Bezeichnung „.C“ entspricht, wurde der Client-Stub und das Server-Skeleton durch diese beiden Optionen so angepaßt. Das Argument zu diesen Optionen wird einfach an den Stamm der verwendeten IDL-Datei angehängt. Die Endung „.hh“ für das erzeugte Header wurde beibehalten, da bereits der DCE IDL-Compiler eine Datei „TRD_tcomm.h“ erzeugt.

Die restlichen Optionen sind mehr oder weniger IDL-Compiler spezifisch. Mit „-B“ werden vom IDL-Compiler Funktionen für den *Basic Object Adapter* erzeugt, der für die Verwendung in Melody völlig ausreicht. Durch „-N“ wird der Compiler angewiesen, auch den Inhalt der Dateien, die durch „#include“ eingebunden werden, mit zu berücksichtigen. Normalerweise wird die „#include“-Anweisung nur in eine entsprechende Anweisung in C++ übersetzt. Dies ist sinnvoll bei Referenzen auf andere Objekte, wurde aber in Melody nicht benutzt. Die letzten beiden Optionen des Compileraufrufes beschäftigen sich mit dem *Interface Repository*. Ohne diese Optionen würde normalerweise keine Kopie der Schnittstellenbeschreibung dort abgelegt und auch der erzeugte Programmcode würde diesen Sachverhalt nicht berücksichtigen. Durch die Option „-R“ wird dieser Teil aktiviert. Um die verschiedenen IDL-Files von verschiedenen Projekten besser trennen zu können, wurde der IDL-Compiler zur weiteren Strukturierung mit der Option „-d melody_if“ angewiesen, die Datei in ein Unterverzeichnis des Repositories names „melody_if“ abzulegen.

Dirch die beiden beschriebenen Befehle werden aus den beiden Dateien *TRD_tcomm.idl* und *TRD_tcomm_data.h*, die nach der in diesem Kapitel beschriebenen Vorgehensweise umgewandelt wurden, die drei Datei *TRD_tcommC.C*, *TRD_tcommS.C* und *TRD_tcomm.hh* erzeugt. Diese Dateien bilden die Grundlage für die Konvertierungen der erzeugten Datenstrukturen in die von Melody selbst benutzen, die im nächsten Kapitel beschrieben werden.

Die CORBA-Melody Konvertierung

Die im letzten Kapitel vom IDL-Compiler erzeugten Dateien sind der Ausgangspunkt für die jetzt folgenden Überlegungen. Die enthaltenen Definitionen und Deklarationen beschreiben vollständig die Schnittstelle, die von CORBA aus angeboten wird, um später auf Melody zugreifen zu können. Es finden sich die Signaturen der Zugriffsmethoden, wie sie in der Schnittstellenbeschreibung beschrieben sind. Alle verwendeten Datenstrukturen wurde dabei entsprechend des CORBA-C++-Mappings in C++-Strukturen und -Klassen umgewandelt. Zusätzlich enthalten viele diese Klassen noch sehr hilfreiche Zugriffsmethoden, die die Verwendung wesentlich vereinfachen und auch für das korrekte Speichermanagement zuständig sind. Eine genaue Beschreibung des C++-Mappings findet sich in der CORBA-Spezifikation der OMG [11] oder in den Handbüchern von Orbix [7]. In diesem Kapitel wird deshalb nicht näher darauf eingegangen. Stattdessen werden die erzeugten Programmteile als gegeben vorausgesetzt.

Zunächst wird auf den vorgefundenen Aufbau von Melody eingegangen, also wie die Datenstrukturen bislang dort verwendet wurden. Im nächsten Schritt werden Alternativen aufgezeigt, wie sich die notwendigen Konvertierungsmethoden in diesen Aufbau grundsätzlich integrieren lassen und welcher Ansatz schließlich realisiert wurde. Von diesem Ansatz wird zunächst die Grobstruktur der Methoden erläutert, bevor dann schließlich die genauen Details erklärt werden. Den Abschluß des Kapitels bildet ein benötigter Workaround, der aufgrund eines Bugs im erzeugten Programmcode notwendig wurde.

Mit Abschluß der Beschreibung in diesem Kapitel kann dann der letzte Schritt der CORBA-Schnittstelle für Melody geschehen: die Implementierung des Schnittstelle zwischen CORBA und Melody sowie dem *Trader User Agent*.

Der vorhandene Aufbau in Melody

In der Version von Melody, die in dieser Arbeit der Ausgangspunkt war, war das DCE-Interface bereits integriert. Um die Handhabung zu vereinfachen und unnötige Codeduplizierung zu vermeiden, waren alle Deklarationen der Strukturen, die bei der Übertragung benutzt wurden, in eine Datei zusammengefaßt worden: *TRD_tcomm_data.h*. Diese Datei wird vom DCE IDL-Compiler gelesen, um – ähnlich wie bei CORBA auch – notwendige Deklarationen und Stub-Routinen zu erzeugen. Andererseits sind die dort enthaltenen Datenstrukturen auch die Basisklassen für entsprechende Klassen von Melody. Dabei wurde eine besondere Eigenschaft dabei ausgenutzt. Da DCE IDL quasi alle möglichen C-Datenstrukturen direkt unterstützt, u.a. auch Zeiger, stellt sich die Umwandlung von DCE Strukturen in die Melody-Klassen denkbar einfach dar: Es genügt an vielen Stellen ein einfaches Kopieren des Speichers, um die Daten zu transferieren. Nur für Daten, die über Zeiger referenziert werden, muß noch etwas mehr Aufwand getrieben werden.

Anhand der Parameterklasse für die Diensttypeinfügung *STInsert* sollen kurz ein paar Aspekte dieser Implementierung erläutert werden. Diese Klasse wird als Parameter dem *Trader Service Agent* übergeben, wenn ein neuer Diensttyp eingefügt werden soll. Sie enthält alle notwendigen Informationen für einen Diensttyp, angefangen vom Namen des Typs bis hin zu allen Attributen und Auswahlregeln. Die Basisklasse für diese Melody-Klasse ist eine Struktur, die aus der bereits erwähnten Include-Datei entnommen wird. Ihr Name leitet sich aus dem Namen der Melody-Struktur durch Anhängen von „_struc“ her. Sie sah (und sieht nach der Anwendung der Makros auch jetzt immer noch) so aus:

```
C++: typedef struct {
        char name[STLENGTH];
        char *stInterface;
        ...
        STTypeNamesNode_structP sTSupertypeNames;
        ...
    } STInsert_struct;
```

Diese Struktur ist die Basisklasse für Melody-Klasse *STInsert*:

```
C++: class STInsert : public STInsert_struct
    {
        ...
        STInsert(STInsert_struct*);
        ...
    }
```

Der hier gezeigte Konstruktor wird benutzt, um aus der von DCE empfangenen Datenstruktur ein neues Objekt herzustellen. Da die Klasse *STInsert* keine zusätzlichen eigenen Attribute mehr definiert und damit von Speicheraufbau und -größe völlig gleich ist mit der Basisklasse, ist diese Aufgabe äußerst einfach: Es genügt im Prinzip ein einfaches Kopieren des Speichers mit einer Funktion wie *memcpy*. Nur bei Zeigern ist noch ein weiterer Schritt notwendig. Bei dem String *stInterface* muß ein neues Haldenobjekt angelegt und der Inhalt des Strings umkopiert werden. Bei dem zweiten Zeiger *sTSupertypeNames* ist es noch einfacher. Es handelt sich um einen Zeiger auf eine andere Struktur, für die diese Ausführungen in gleicher Weise gelten.

Es genügt für die korrekte Konvertierung, den Konstruktor der Klasse *STTypeNamesNode* mit dem Parameter *STSupertypeNames* aufzurufen, um so das Objekt aus der Übertragungsdatenstruktur zu erstellen. Der entsprechende Teil könnte also etwa so aussehen:

```
C++:  STInsert::STInsert(STInsert_struct *my_struct)
      {
        if (my_struct != NULL) {
            memcpy(this, my_struct, sizeof(STInsert_struct));

            if (my_struct->STInterface != NULL) {
                sTInterface =
                    new char[strlen(my_struct->STInterface)+1];
                strcpy (sTInterface,my_struct->STInterface);
            };

            ...

            if (my_struct->STSupertypeNames != NULL) {
                sTSupertypeNames =
                    new STTypeNamesNode(my_struct->STSupertypeNames);
            };

            ...
        }
    }
```

Noch einfacher gestaltet sich die Umwandlung der Daten in der Gegenrichtung. Da die Melody-Klasse eine Unterklasse der Übertragungsstruktur ist, und jedes Objekt einer abgeleiteten Klasse gleichzeitig auch als ein Objekt der Klasse selbst angesehen werden kann, ist diese Umwandlung durch die Klassenhierarchie in C++ abgedeckt: Es ist also nichts weiter notwendig.

Die Integration der CORBA-Schnittstelle

Da die Datenstrukturen, die vom CORBA IDL-Compiler erzeugt werden, vom Aufbau verschieden von den ursprünglichen in C++ sind, muß für CORBA ein anderer Weg beschritten werden. Doch bevor auf die tatsächliche Realisierung der Integration eingegangen werden soll, zunächst ein paar Worte zu möglichen Alternativen.

Die CORBA-Schnittstelle stellt eine neue Zusatzoption für Melody dar. Sie soll sich natürlich möglichst gut in den vorhandenen Aufbau einfügen, andererseits aber trotzdem noch als Zusatz erkennbar und möglichst noch herausnehmbar sein, damit Melody auch ohne CORBA immer noch voll lauffähig ist. Beide Ziele scheinen auf den ersten Blick widersprüchlich, lassen sich aber in der Tat sehr gut vereinen, wenn man nach genauerer Betrachtung feststellt, daß beide durchaus verschiedene Aspekte der Implementation betreffen.

Die saubere Trennung des neuen Programmcodes vom bestehenden läßt sich sehr einfach dadurch erreichen, daß die neuen Teile in Blöcke gestellt werden, die durch entsprechende Preprocessor-Direktiven herausgenommen werden können. Definiert man für die CORBA-Schnittstelle ein neues Preprocessor-Symbol *USE_CORBA* und schließt alle Änderungen in „*#ifdef*

`USE_CORBA ... #endif` Blöcke ein, so läßt sich sehr einfach die alte Version erzeugen, wenn dieses Symbol nicht definiert wird. Ein angenehmer Seiteneffekt dieser Maßnahme ist, daß man sehr leicht alle Änderungen für die CORBA-Schnittstelle wiederfindet.

Falls möglich, kann aber noch weiter gegangen und größere Teile in neue Dateien ausgelagert werden, die ausschließlich die CORBA-Schnittstelle benötigt. Dies stellt sozusagen den Idealfall der Trennung dar, da dann der bestehende Programmcode fast¹ überhaupt nicht geändert werden muß. Dadurch vermeidet man neue Abhängigkeiten der bestehenden Dateien von CORBA und die sehr umfangreichen Deklarationen der CORBA-Datenstrukturen müssen nicht eingebunden werden. Die Übersetzung alter Teile wird nicht verlangsamt und nur die neuen Teile hängen von CORBA ab.

Die bisherigen Ausführungen beschränkten sich nur auf die Ebene des Programmscodes, ohne den inhaltlichen, logischen Aufbau des Programms selbst zu berücksichtigen. Die Frage, wie die Methoden für CORBA integriert werden können, ist immer noch offen. Generell sind hier wieder zwei extreme Positionen denkbar: entweder die CORBA-Methoden werden, ähnlich wie schon die vorhandenen Methoden für DCE, voll in die bestehenden Melody-Objekte integriert oder aber die CORBA-Schnittstelle wird vollständig getrennt in neue Objekte verlagert. Letztere Alternative hat den Vorteil, daß mit der Trennung der Objekte auch der Programmcode vollständig ausgelagert und somit der oben beschriebene Idealfall erreicht werden kann.

Doch damit wird nur ein sehr geringer Grad der Integration erreicht. Die CORBA-Schnittstelle ist getrennt vom Rest und hat einen völlig anderen Aufbau als die vorhandene DCE-Schnittstelle. Mehr noch: durch die totale Trennung wird die CORBA-Schnittstelle stärker aus dem Bewußtsein des Programmierers verbannt, was sich sehr nachteilig auswirken kann. Schließlich ist es bei Änderungen an den zugrundeliegenden Datenstrukturen notwendig, daß sowohl die DCE- als auch die CORBA-Methoden entsprechend angepaßt werden. Bei DCE ist dies meist sehr einfach und bei Daten, die nicht über Zeiger benutzt werden, muß gar nichts getan werden. Bei CORBA aber ist dies, wegen des anderen Aufbaus der Datenstrukturen nicht möglich, und es müssen alle Felder einzeln kopiert werden. Wird ein neues Feld hinzugefügt, so muß die CORBA-Methode entsprechend erweitert werden. Ist der Programmcode für CORBA aber in ein anderes Modul verlagert worden, so kann dies leicht vergessen werden, was zu einem fehlerhaften Programm führt, obwohl es sich immer noch fehlerfrei übersetzen läßt.

Damit wird deutlich, daß die totale Trennung von CORBA vom bestehenden Programm kaum wünschenswert sein kann. Eine integrativere Lösung ist für zukünftige Änderungen sicherlich die bessere. Und da an Melody im Rahmen von Studien- und Diplomarbeiten eine Vielzahl von Programmierern beteiligt sind, scheint die Integration in die bestehenden Melody-Objekte die beste Alternative. Sie läßt sich im Prinzip ähnlich aufbauen, wie sie schon in DCE vorhanden ist: Die Konvertierung von CORBA geschieht über einen entsprechenden Konstruktor bei Erstellung des Objekts. Die Gegenrichtung dagegen bedarf einer zusätzlichen Methode, da nicht einfach, wie dies bei DCE geschieht, das Basisobjekt verwendet werden kann. Doch durch eine Bündelung im Programmcode dürfte dies wenig Probleme bereiten.

1. Kleinere Änderungen lassen sich natürlich nie vollkommen umgehen, wenn man nicht zur Laufzeit auf Symboltabellen arbeiten möchte.

Diese „Vollintegration“ wurde deshalb die Wahl dieser Studienarbeit. Die CORBA-Methoden stehen so mehr im Bewußtsein des Programmierers, der Aufbau lehnt sich an den Aufbau von DCE an und auch sonst – was die Teile des Server-Frontends und des TUA angeht – kann der Design symmetrisch erfolgen. Die CORBA-Schnittstelle fügt sich so gut in die vorhandenen Teile ein. Durch die oben beschriebene Verwendung des Preprocessors andererseits läßt es sich erreichen, daß CORBA, falls nicht erwünscht oder nicht vorhanden, gar nicht eingebunden wird und eine reine DCE-Version erstellt wird. Und die CORBA-Teile sind im Programmcode bereits durch die entsprechenden Direktiven leicht erkennbar.

Einziger Nachteil dieser Lösung ist, wie bereits erwähnt, die notwendige Einbindung der sehr umfangreichen CORBA-Deklarationen, die zur Verlängerung der Übersetzungszeiten beitragen. Besonders bedauerlich ist es, daß bereits für die Deklaration der verschiedenen Melody-Klassen in den Headerfiles diese Einbindung nötig ist, was sich vermutlich nicht ohne Umgehung der Typsicherheit oder Einführung von neuen Klassen vermeiden läßt. Zwar erlaubt C++ bei Zeigern, daß auf Strukturen verwiesen wird, deren genauer Aufbau nicht bekannt ist. Doch ist dies nicht möglich, wenn diese Strukturen innerhalb von Klassen definiert sind. Da der CORBA IDL-Compiler alle Definitionen in eine Klasse namens „*MELODY_trader*“ ablegt, können Zeiger auf die CORBA-Strukturen nicht direkt verwendet werden, solange nicht diese Klasse definiert wird. Es gibt jetzt zwei Möglichkeiten, wie dies umgangen werden könnte: Entweder wird eine „Pseudo-Klasse“ gleichen Namens angelegt, deren einziger Zweck die Deklaration der benötigten Strukturen ist. Doch die Verwendung dieser Pseudo-Klasse gestaltet sich schwierig. Sie darf nur für Programmmodule verwendet werden, die nicht die richtigen CORBA-Definitionen benötigen, da die Klasse sonst doppelt definiert wird, was zu einem Übersetzungsfehler führt. Somit müßte also jedes Programmmodul zu Beginn definieren, ob es die kompletten CORBA-Definitionen braucht. Dies und die Tatsache, daß es damit in Melody zwei konkurrierende Definitionen einer Klasse gibt, macht deutlich, daß das Ergebnis kaum einem sauberen Programmdesign entspricht.

Die andere Lösung ist die Verwendung von *void*-Zeigern und expliziten *Type-Casts*. Dadurch werden sämtliche doppelten Definitionen vermieden und die CORBA-Definitionen müssen wirklich nur in Modulen eingebunden werden, wenn sie auch wirklich benötigt werden. Der große Nachteil ist aber, daß sämtliche Typsicherheit aufgegeben wird. Da ein *void*-Zeiger auf alles zeigen kann und immer explizit umgewandelt werden muß, kann der Compiler keine Überprüfung machen, ob die tatsächlich verwendeten Argumente auch dem eigentlich gewünschten Typ entsprechen. Da diese statische Typüberprüfung aber ein sehr wichtiges und hilfreiches Mittel bei der Programmentwicklung und -pflege ist, kommt ihre Umgehung für so umfangreiche Eingriffe, wie sie für die CORBA-Schnittstelle notwendig sind, nicht in Frage.

Also bleibt nichts anderes übrig, als eben doch immer sämtliche CORBA-Definitionen einzubinden, auch wenn sie möglicherweise in verschiedenen Programmmodulen überhaupt nicht benötigt werden und sich die Übersetzungszeit dadurch verlängert. Sauberer Programmdesign und Typsicherheit sprechen aber sehr für diese Lösung.

Die CORBA-Konvertierungsmethoden

Die Schnittstelle für die Konvertierung der CORBA-Datenstrukturen besteht im wesentlichen aus zwei neuen Methoden für jedes Objekt, das für die Übertragung benötigt wird. Ähnlich wie für DCE auch, wird die Konvertierung von CORBA durch einen Objekt-Konstruktor bewerkstelligt, der als Parameter die empfangene CORBA-Struktur bekommt und alle Daten an die passende Stelle im Objekt kopiert sowie für enthaltene Objekte ebenfalls die Konvertierung anstößt. Da der Aufbau der Daten in CORBA von den Strukturen in C++ differiert, kann nicht wie bei DCE einfach der Speicher kopiert werden, sondern es müssen sämtliche Felder einzeln umgewandelt werden.

Für die Gegenrichtung nach CORBA wird eine weitere Methode angelegt, die aus den Objekt-daten die Datenstruktur von CORBA ausfüllt. Diese Methode hat keine Entsprechung in DCE, da ja bereits durch die Ableitung der Melody-Klasse von den Übertragungsstrukturen die Konvertierung implizit vorhanden ist. In CORBA ist dafür eine neue Methode notwendig, die in dieser Implementierung einheitlich für alle Objekte „*make_oo*“ genannt wurde. Sie bekommt als Parameter immer eine Referenz auf die passende CORBA-Struktur, in die der Objekttinhalt kopiert wird. Beide Methoden seien einmal am Beispiel für die Klasse *STInsert* erläutert. Zunächst einmal die Erweiterung der Klassendeklaration:²

```
C++: class STInsert : public STInsert_struct {
    ...

    #ifdef USE_CORBA
        // Constructor for CORBA structure
        STInsert(const MELODY_trader::STInsert_struct_oo&);

        // Copy function to fill CORBA structure
        void make_oo(MELODY_trader::STInsert_struct_oo&) const;
    #endif

    ...
};
```

Der neue Konstruktor liest aus der übergebenen Struktur die Daten und initialisiert damit ein neues Objekt von *STInsert*. Da er nur die Daten liest, kann der Parameter als *const* deklariert werden. *make_oo* ist die umgekehrte Methode. Sie kopiert die Daten aus dem aktuellen Objekt in die übergebene Struktur. Hier kann wieder, weil das Objekt nicht verändert wird, die Methode für ein *const*-Objekt deklariert werden.

Beide Methoden haben ähnliche Aufgaben, jeweils nur in verschiedener Richtung. Für Felder, die nicht über Zeiger referenziert werden, kopieren die Methoden den Inhalt unter Berücksichtigung notwendiger Typ-Konvertierungen direkt in das Ziel-Feld. Für die Felder, die in C++ einen Zeiger auf ein weiteres Objekt enthalten, für das vergleichbare Methoden vorhanden sind, wird die Konvertierung in gleicher Weise angestoßen. Der Rest der Felder wird ebenfalls

2. Die umschließenden Preprocessor-Direktiven sind hier exemplarisch mit angegeben, werden in den weiteren Beispielen zur besseren Übersichtlichkeit aber weggelassen.

einzelnen umgesetzt, wobei im Konstruktor jeweils ein neues Haldenobjekt angelegt werden muß, während das Speichermanagement in der Gegenrichtung bereits durch die Zugriffsmethoden der CORBA-Strukturen automatisch erledigt wird. Die beiden Methoden aus dem obigen Beispiel sollen im Folgenden genauer betrachtet werden:

```
C++:  STInsert::STInsert(const
      MELODY_trader::STInsert_struct_oo& in)
      {
        // Hier werden die einzelnen direkten Felder
        // konvertiert.
        strcpy(name, in.name);

        if (in.sTInterface._d()) {
          sTInterface = new
            char[strlen(in.sTInterface.sTInterface()) + 1];
          strcpy(sTInterface, in.sTInterface.sTInterface());
        } else
          sTInterface = NULL;

        // Jetzt noch die referenzierten Objekte.
        if (in.sTDescriptionList.length() == 0)
          sTDescriptionList = NULL;
        else
          sTDescriptionList = new
            STDescriptionNodeT(in.sTDescriptionList);

        // Hier kommen die restlichen Felder
        ...
      }
```

Der Konstruktor muß aus den empfangenen Daten ein neues Objekt der Klasse *STInsert* erstellen. Die Basisklasse dafür ist *STInsert_struct*. Die Klasse enthält insgesamt sechs Felder, von denen die ersten drei in diesem Beispiel beschrieben sind. Dabei handelt es sich um „*name*“, einen String mit vorgegebener, maximaler Länge, „*sTInterface*“, einen String beliebiger Länge, und „*sTDescriptionList*“ einen Zeiger auf eine Liste von *STDescriptionNodeT*. Die Konvertierung des ersten Feldes ist die einfachste: Da der Speicherplatz für den String bereits in der Struktur angelegt ist, kann einfach der Inhalt direkt kopiert werden.³

Das zweite Feld „*sTInterface*“ ist ein String beliebiger Länge, der mit dem Makro *VAR-STRING* aus dem vorigen Kapitel angelegt wurde. Er besteht in CORBA aus einer Union, die zwei Varianten hat: Bei Diskriminante *TRUE* ist der String im Feld *sTInterface* gesetzt, bei *FALSE* wird ein *NULL*-Zeiger repräsentiert und das *dummy*-Feld der Union ist auf einen beliebigen boolschen Wert gesetzt. Aus dieser besonderen Darstellung des *NULL*-Zeigers leitet sich bereits die erste Abfrage im obigen Beispiel ab. Wenn die Diskriminante, die über die Methode „*_d()*“ abgefragt werden kann, nicht *FALSE* ist, handelt es sich um diesen Fall und das Ziel-

3. Es wird davon ausgegangen, daß CORBA eine eventuell notwendige Umsetzung der Zeichensätze automatisch bei der Übertragung vornimmt.

Feld wird entsprechend gesetzt. Im anderen Fall muß ein neuer String auf der Halde angelegt werden, der ausreichend Platz bietet und in den der empfangene Text kopiert wird.

Für die Liste in „*STDescriptionList*“ muß in analoger Weise die Unterscheidung des *NULL*-Zeiger erfolgen. Dies ist genau dann der Fall, wenn die Liste die Länge 0 hat. Sonst muß ein neues Listenobjekt angelegt und entsprechend initialisiert werden. Im C++-Feld wird deshalb ein neues Objekt der Listenklasse angelegt. Als Parameter müssen die Informationen aus der CORBA-Struktur übergeben werden. Der so aufgerufene Konstruktor hat wieder die gleiche Aufgabe: Die einzelnen Felder der Struktur zu konvertieren, wobei möglicherweise wieder weitere „Unterobjekte“ angelegt werden müssen.

Jetzt noch die Konvertierung in der Gegenrichtung:

```
C++: void
    STInsert::make_oo(MELODY_trader::STInsert_struct_oo& out)
    const
    {
        out.name = (const char *) name;
        if (sTInterface == NULL)
            out.sTInterface.dummy(FALSE);
        else
            out.sTInterface.sTInterface(
                (const char *) sTInterface);

        ((STDescriptionNodeT *)
         sTDescriptionList)->make_oo(out.sTDescriptionList);

        // Hier kommt noch der Rest ...
    }
```

Die Methode *make_oo* der Klasse *STInsert* hat die Aufgabe, die Daten des aktuellen Objekts in die CORBA-Struktur zu kopieren, die als Referenzparameter übergeben wird. Wieder müssen die direkten Felder einzeln kopiert und bei weiteren, enthaltenen Objekten die Konvertierung durch Aufruf seiner *make_oo*-Methode angestoßen werden. Die Konvertierung des ersten Feldes gestaltet sich sehr einfach. Beim Ziel-Feld in CORBA handelt es sich um ein Objekt der Klasse *String_mgr*, das bereits das gesamte Speichermanagement übernimmt. Der Zuweisungsoperator „=“ ist durch *Overloading* ersetzt worden. Um eine Kopie vom Original anlegen zu lassen, muß unbedingt ein Objekt vom Typ „*const char **“ übergeben werden! Wird dies nicht getan, dann kopiert das Objekt nur den Zeiger und erwirbt damit die „Rechte“ am referenzierten String. Das bedeutet unter anderem, daß der belegte Speicher bei Freigabe der gesamten Struktur automatisch mit freigegeben wird. Da aber die Strings im C++-Objekt immer noch verwendet werden, muß dies verhindert werden, weshalb ein *const* Objekt übergeben wird.

Das gleiche muß beim zweiten Feld *sTInterface* beachtet werden. Hier kommt noch die besondere Behandlung für die *NULL*-Zeigerdarstellung hinzu. Wenn der String gesetzt ist, wird er in das gleichnamige Feld der Union kopiert. Der explizite *Cast* zu *const* erzwingt wieder die Kopie. Ist der String dagegen nicht gesetzt, wird die *Dummy*-Variante durch Setzen eines beliebigen booleschen Werts gewählt. In der beschriebenen Realisierung wurde immer *FALSE* gewählt. (Als Assoziationshilfe für die Diskriminante, die denselben Wert hat.)

Für die Liste im dritten Feld ruft die Methode wieder die entsprechende Konvertierungsmethode des beinhalteten Objekts auf. Da die Objektklasse *STInsert* aus der Struktur *STInsert_struct* abgeleitet wurde, die selbst wiederum die Struktur *DescriptionNode_struct* für die Liste benutzt, muß hier noch ein expliziter *Cast* in die Melody-Klasse *STDescriptionNodeT*⁴ erfolgen, die in Wirklichkeit von Melody dort gespeichert wird. An dieser Stelle muß beachtet werden, daß hier keine Unterscheidung des *NULL*-Zeigers stattfindet! Für Listen kann dies direkt in der Konvertierungsmethode der Liste geschehen. Die Referenzierung des *NULL*-Objekts macht keine Probleme, da die Methodenaufrufe bereits statisch zur Übersetzungszeit festliegen.

Die Konvertierungsmethoden von Listen

Die bisherigen Ausführungen zu den beiden Konvertierungsmethoden treffen auf alle verwendeten Strukturen außer den Listenstrukturen zu. Bei Listen sind die Verhältnisse etwas anders. Auf der C++-Seite sind sie als verkettete Listen realisiert, bei denen immer ein Knoten die eigentliche Elementstruktur direkt, also ohne Indirektion über einen Zeiger, verwendet. Dadurch ergibt sich eine Mischfunktion der Listenstruktur sowohl als Listenknoten wie auch als Inhaltsträger. Auf der CORBA-Seite sind aber Inhalt und Liste besser getrennt, da dort eine Liste ein Array von Inhaltselementen ist. Dies wurde ausgenutzt, um die Konvertierung der Liste als ganzes von der Konvertierung des Listeninhalts zu trennen. Deshalb enthalten Listenobjekte zu den zwei, von allen Objekten bereit gestellten Konvertierungsmethoden drei weitere, ausschließlich vom Listenobjekt selbst verwendete, private Methoden.

```
C++: private:
    // Helper methods for _el structs.
    void take_oo(const MELODY_trader::takeEl_el_struct_oo&);
    void make_oo(MELODY_trader::takeEl_el_struct_oo&) const;
    // Constructor for el CORBA structure
    takeEl(const MELODY_trader::takeEl_el_struct_oo&,
           takeEl*);

public:
    // Constructor for CORBA structure
    takeEl(const MELODY_trader::takeEl_struct_oo&);
    // Copy function to fill CORBA structure
    void make_oo(MELODY_trader::takeEl_struct_oo&) const;
```

Zunächst einmal existieren selbstverständlich die beiden, bereits bekannten *public* Methoden mit dem entsprechenden Parameter. Dadurch bleiben die Implementierungsdetails der Liste verborgen, und sie kann ähnlich wie andere Datenstrukturen umgesetzt werden. Der einzige, sichtbare Unterschied außerhalb des Listenobjekts selber, ist die Behandlung des *NULL*-Zeigers, der im Gegensatz zu den normalen Strukturen, die eine zusätzliche *Union* einführen, durch die Listenlänge berücksichtigt wird.

4. Sie hat als Basis-Klasse *DescriptionNode_struct*, abweichend von der sonst üblichen Benennung in Melody, da von ihr insgesamt vier verschiedenen Listen abgeleitet werden.

Für die Konvertierung von CORBA nach C++ werden insgesamt drei Methoden definiert: die beiden Konstruktoren und die Methode „*take_oo*“. Der *public*-Konstruktor hat die Aufgabe, die gesamte Liste, die als Parameter übergeben wird, zu konvertieren. Das angelegte Objekt selber bildet den Kopf der Liste. Der zweite Konstruktor bekommt als Parameter ein einzelnes Element der Liste sowie einen Zeiger auf den Rest der schon behandelten Listenelemente. Er muß die Umsetzung dieses Elements anstoßen und dann den *next_el*-Zeiger der Liste auf den Rest der Liste verweisen lassen. Die letzte Methode „*take_oo*“ wandelt den Inhalt eines Elements um. Durch die Aufteilung in diese drei Methoden besteht eine klare Trennung zwischen der Funktionalität für Listen als Ganzes, die immer völlig gleich abläuft, und der für den Inhalt eines Elements.

Die Konvertierung in der Gegenrichtung, also von C++ nach CORBA, wurde ebenfalls in zwei Methoden aufgespalten. Wie bereits an den Deklarationen zu erkennen ist, ist wieder eine Methode für die Liste an sich zuständig, während die zweite, *private* deklarierte Methode ausschließlich den Inhalt betrachtet.

Nachdem klar ist, welche Aufgaben die einzelnen Methoden haben, kann jetzt die Implementierung näher betrachtet werden. Zunächst wieder die Richtung von CORBA nach C++:

```
C++: takeEl::takeEl(const MELODY_trader::takeEl_struct_oo& in)
    {
        takeEl *next = NULL;

        for (int i = in.length() - 1; i >= 1; i--) {
            next = new takeEl(in[i], next);
        }

        take_oo(in[0]);
        next_el = next;
    }
```

Wie bereits erklärt, konvertiert dieser Konstruktor die gesamte Liste. Er baut aus dem Array, das im Parameter übergeben wird, eine entsprechende verkettete Liste auf. Das Objekt selbst enthält den Kopf der Liste. Dazu wird zunächst die Liste von hinten nach vorne aufgebaut und immer durch den Zeiger in *next* verkettet. Die gewählte Realisierung erhält die Reihenfolge der Elemente in gleicher Weise, so daß am Ende der Kopf der Liste das erste Element des Arrays enthält. Die Verwendung des Konstruktors mit explizitem *next*-Zeiger vermeidet zudem rekursive Aufrufe.

```
C++: takeEl::takeEl(
        const MELODY_trader::takeEl_el_struct_oo& in,
        takeEl* nextNode)
    {
        take_oo(in);
        next_el = nextNode;
    }
```

Der zweite Konstruktor hat die Aufgabe, ein einzelnes Element umzusetzen und den Rest der bereits bestehenden Liste anzuhängen. Da der Inhalt durch *take_oo* konvertiert wird, genügen also ein Aufruf und eine Zuweisung. Die Implementierung der beiden Konstruktoren zeigt den

Grund für die Trennung von Inhalt und Liste ganz deutlich: Beide Konstruktoren müssen den Inhalt eines Knotens umsetzen. Durch die Einführung von *take_oo* wird unnötige Duplizierung von Programmcode vermieden.

```
C++: void
    takeEl::take_oo(
        const MELODY_trader::takeEl_el_struct_oo& in)
    {
        elem.tType = (takeType) in.tType;
        elem.name = (const char *) in.name;
        elem.supname = (const char *) in.supname;
    }
```

Schließlich noch die Umsetzung des Inhalts eines Knotens selber. In *take_oo* gelten die gleichen Regeln, wie sie bereits früher für die normalen Datenstrukturen beschrieben wurden. Einzige Abweichung ist die Verwendung des „*elem*“-Feldes im Listenknoten als Ziel. Damit ist die Konvertierung der CORBA-Struktur nach C++ vollständig. In der aktuellen Realisierung sind alle Listen genau in dieser Weise aufgebaut. Insbesondere bei den beiden Konstruktoren wird deutlich, daß Listen noch weiter vereinheitlicht werden können und sich der Entwurf eines *Templates* für Listen vielleicht lohnen könnte.

Die Konvertierung in der Gegenrichtung ist, wie oben bereits erklärt, in zwei Methoden aufgeteilt, um wieder zwischen Inhalt eines Elements der Liste und der Liste als ganzes zu trennen. Die Methode *make_oo*, die als Parameter die Listenstruktur bekommt, organisiert die Gesamtkonvertierung und ruft dabei immer die zweite Methode für die einzelnen Elemente auf.

```
C++: void
    takeEl::make_oo(MELODY_trader::takeEl_struct_oo& out)
    const
    {
        const takeEl *next = this;

        for (int i = 0; next != NULL; i++)
            next = next->next();
        out.length(i);

        next = this;
        for (i = 0; next != NULL; i++) {
            next->make_oo(out[i]);
            next = next->next();
        }
    }
```

Sie ist in zwei ähnliche Teile aufgeteilt. Die erste Schleife geht einmal über die ganze Liste und bestimmt so ihre Länge. Der Aufruf der *length*-Methode setzt das Array in der CORBA-Struktur auf diese Länge. Es wäre zwar ebenfalls möglich, das Array immer sukzessive um ein Feld zu vergrößern. Dies hätte aber für das Speichermanagement Nachteile, da dann jedesmal ein neuer, größerer Speicherbereich auf der Halde angelegt und sämtliche bisherigen Listenelemente umkopiert werden müßten. Die CORBA-Realisierung mag vielleicht optimiert sein und auch Speicher schon auf Vorrat belegen. Trotzdem wird früher oder später ein Umkopieren not-

wendig werden. Da die CORBA-Datenstrukturen aber einzig und allein für die einmalige Übertragung der Daten verwendet werden, also nach ihrer Erzeugung nicht mehr verändert werden, kann dies leicht durch das vorherige Durchzählen vermieden werden. Eine Besonderheit dieser Methode ist noch, daß sie auch für den Fall einer leeren Liste das korrekte Ergebnis liefert. Bei Aufruf der Methode für ein *NULL*-Objekt ist der *this*-Zeiger *NULL*. Damit bricht die Schleife gleich zu Beginn ab und ergibt Länge 0.

Die zweite Schleife geht ein zweites Mal über die gesamte Liste und konvertiert jedes einzelne Element der Liste in ein Feld des Arrays von CORBA. Dabei wird wieder die Reihenfolge der Liste berücksichtigt. Die Konvertierung geschieht durch Aufruf der Methode für die Element-Struktur, die analog wie bereits beschrieben, die einzelnen Felder der Struktur umwandelt. Für *takeEl* sieht dies etwa so aus:

```
C++: void
      takeEl::make_oo(MELODY_trader::takeEl_el_struct_oo& out)
      const
      {
          out.tType = (takeType_oo) elem.tType;
          out.name = (const char *) elem.name;
          out.supname = (const char *) elem.supname;
      }
```

Die Umwandlung der einzelnen Datentypen

Für die verschiedenen Beispiele in diesem Kapitel ist am Rande bereits auf die konkrete Umwandlung der einzelnen Datentypen eingegangen worden. Jetzt sollen aber an dieser Stelle alle verwendeten Typen in einer Übersicht dargestellt werden. Es wird dabei nicht mehr näher auf die CORBA Spezifikation eingegangen, sondern ausschließlich erläutert, warum die Realisierung so vorgenommen wurde.

Auch an dieser Stelle werden wieder einige neue Makros definiert, um die Umsetzung der einzelnen Typen zu vereinheitlichen und an einer zentralen Stelle verwalten zu können. Dadurch wird insbesondere die Anpassung an spätere, leicht variierende Versionen von CORBA wesentlich erleichtert, da sie nur an einer Stelle erfolgen muß. Die Makrodefinitionen, die im Folgenden eingeführt werden, finden sich alle in der Header-Datei *TRD_tcomm_oo.h*.

Die Umsetzung der einfachen Datentypen

Die Umsetzung von einfachen Ganzzahlen stellt sich natürlich am einfachsten dar. Sie können ohne weitere Maßnahmen direkt kopiert werden. Durch eine geeignete Wahl der entsprechenden Datentypen in CORBA läßt sich ein Überlauf vermeiden, so das dies kein Problem darstellt.

Ähnliches gilt auch für die Aufzählungstypen. Sie werden sowohl in C++ als auch in CORBA intern als eine Ganzzahl dargestellt, die sich wieder problemlos kopieren läßt. Dabei muß natürlich sichergestellt sein, daß die Abbildung der definierten Symbole zu den entsprechenden Zahlen in gleicher Weise erfolgt. Dies ist aber glücklicherweise der Fall, da die CORBA Spezi-

fikation ebenfalls wie C++ auch die Symbole in der Reihenfolge ihrer Definition durchnummeriert. Damit können Aufzählungstypen direkt kopiert werden.

```
// von C++ nach CORBA
out.aType = (AType_oo) aType;
// von CORBA nach C++
aType = (AType) in.aType;
```

Die Umsetzung der Strings

Auf die Umsetzung der Strings ist bereits früher eingegangen worden, weshalb hier nur noch einmal eine kurze Zusammenfassung folgen soll. Für Strings werden in den CORBA-Struktur-Objekte der Klasse *String_mgr* angelegt, die bereits ihre Verwaltung und Handhabung in einfacher Weise ermöglichen. Er in der CORBA-Struktur läßt sich sehr einfach durch eine Zuweisung setzen, da der Zuweisungsoperator von der Klasse neu definiert wird. Bei der Zuweisung ist aber zu beachten, daß das CORBA-Objekt mit der Zuweisung nur den Zeiger des Strings in den internen Speicher kopiert, solange es sich nicht um ein *const* Objekt handelt. Da die Strings von Melody noch weiterhin benötigt werden, muß deshalb bei der Zuweisung ein *Cast* dies sicherstellen.

In der Gegenrichtung von CORBA nach C++ kann direkt ohne weitere Probleme der String in das Objekt kopiert werden. Bei String ohne maximale Länge muß natürlich zuvor noch der Speicherplatz auf der Halde belegt werden, da diese Strings ja als einfache Zeiger realisiert sind.

Um die Handhabung zu vereinfachen und zu vereinheitlichen, wurden für die notwendigen Operationen Makros definiert, die für die Umwandlung von Strings verwendet werden sollten. Dadurch wird nichts vergessen und falls in einer späteren CORBA-Version Anpassungen notwendig sein sollten, so lassen sich diese an einer zentralen Stelle durchführen. In Anlehnung an die Benennung der Konvertierungsmethoden wurden bei den folgenden Makros immer der Präfix „*MAKE_*“ für die Konvertierung nach CORBA und „*TAKE_*“ von CORBA verwendet.

```
#define MAKE_STRING(name) out.name = (const char *) name
#define MAKE_ELEMSTRING(name) \
    out.name = (const char *) elem.name

#define MAKE_VARSTRING(name) \
    if (name == (char *) NULL) \
        out.name.dummy(FALSE); \
    else \
        out.name.name((const char *) name)

#define MAKE_ELEMVARSTRING(name) \
    if (elem.name == (char *) NULL) \
        out.name.dummy(FALSE); \
    else \
        out.name.name((const char *) elem.name)

// Beispiele
MAKE_STRING(name);
```

```
MAKE_VARSTRING(single_value);

// und in Listen
MAKE_ELEMSTRING(supname);
MAKE_ELEMVARSTRING(sInterface);
```

Für die Umwandlung von C++ nach CORBA stehen vier Makros zur Verfügung: *MAKE_STRING(name)* erzeugt die notwendige Anweisung, um einen String mit bekannter, maximaler Länge zu kopieren. Diese Strings wurden in der Schnittstellenbeschreibung mit dem Makro *STRING* angelegt. *MAKE_VARSTRING(name)* ist entsprechend das Makro für die Strings ohne Längenbeschränkung, die mit *VARSTRING* definiert wurden. Dieses Makro berücksichtigt noch die besondere Darstellung des *NULL*-Zeigers und setzt entsprechend entweder den Dummy oder den String.

Die beiden zusätzlichen Makros *MAKE_ELEMSTRING* und *MAKE_ELEMVARSTRING* sind die Pendanten, die in Listenobjekten verwendet werden können. Der Inhalt eines Listenknotens wird ja immer im Feld *elem* der Knotens selbst gespeichert. Diese beiden Makros berücksichtigen diesen Umstand bei der Umwandlung.

```
#define TAKE_STRING(name) strcpy(name, in.name)
#define TAKE_ELEMSTRING(name) strcpy(elem.name, in.name)

#define TAKE_VARSTRING(name) \
    if (in.name._d()) { \
        name = new char [strlen(in.name.name()) + 1]; \
        strcpy(name, in.name.name()); \
    } else \
        name = (char *) NULL

#define TAKE_ELEMVARSTRING(name) \
    if (in.name._d()) { \
        elem.name = new char [strlen(in.name.name()) + 1]; \
        strcpy(elem.name, in.name.name()); \
    } else \
        elem.name = (char *) NULL

// Beispiele
TAKE_STRING(name);
TAKE_VARSTRING(single_value);

// und in Listen
TAKE_ELEMSTRING(supname);
TAKE_ELEMVARSTRING(sInterface);
```

Wie bereits angedeutet ist die Umwandlung in Gegenrichtung genauso einfach. Für Strings mit maximaler Länge reicht sogar ein einfaches Kopieren aus, da ja der Speicherplatz bereits fest in der Struktur angelegt ist. Eine entsprechende Zugriffsmethode der *String_mgr*-Klasse macht es möglich, daß direkt aus dem String gelesen werden kann. Die beiden Makros für die Strings beliebiger Länge müssen natürlich, bevor sie den String kopieren, überprüfen, ob der String

überhaupt gesetzt war. Nach Abfrage der Diskriminante wird dann entweder Speicherplatz auf der Halde belegt und der String dorthin kopiert oder der Zeiger im Objekt wird einfach auf *NULL* gesetzt. Wie bereits für die *MAKE*-Makros auch, sind die beiden Makros *TAKE_ELEMSTRING* bzw. *TAKE_ELEMVARSTRING* für den Einsatz in Listen gedacht.

Die Umwandlung von Listen

Auch für die Listen ist bereits weiter oben die Umsetzung angegeben worden. Da sie ebenfalls immer in gleicher Weise erfolgt, wurden auch sie Makros definiert, die alle notwendigen Anweisungen automatisch erzeugen. Wie auch bei den Strings muß unterschieden werden zwischen Listen, die in normalen Datenstrukturen referenziert werden und Listen, die selbst innerhalb von anderen Listen auftauchen.

Folgende zwei Makros sind für die Umwandlung von C++ nach CORBA definiert:

```
#define MAKE_LIST(name, type) \  
    ((type *) name)->make_oo(out.name);  
  
#define MAKE_ELEMLIST(name, type) \  
    ((type *) elem.name)->make_oo(out.name);  
  
// Beispiele  
MAKE_LIST(valueList, valueEl)  
MAKE_ELEMLIST(STSubtypeName, STTypeNamesNode);
```

Da in den Makros die entsprechenden *make_oo* Methoden der enthaltenen Listen aufgerufen werden müssen, muß zusätzlich zum Namen des umzuwandelnden Felds auch der Typ des Objekts mit übergeben werden. Mit Hilfe dieses Typs kann der Zeiger, der dort gespeichert ist, in den korrekten Typ umgewandelt werden. Dies ist notwendig, da die Melody-Klassen von den Übertragungsstrukturen abgeleitet werden, die selbst natürlich nur Referenzen auf andere Übertragungsstrukturen enthalten können. Da sie aber in den Melody-Klassen dann Zeiger auf die entsprechenden, abgeleiteten Objekte enthalten, muß an dieser Stelle ein *Cast* erfolgen.

In der Gegenrichtung muß die besondere Darstellung des *NULL*-Zeigers noch berücksichtigt werden. Dies ist genau dann der Fall, wenn die empfangene Liste die Länge null hat. Dies wird abgeprüft, bevor ein neues Objekt des entsprechenden Typs erzeugt wird.

```
#define TAKE_LIST(name, type) \  
    if (in.name.length() == 0) \  
        name = NULL; \  
    else \  
        name = new type(in.name)  
  
#define TAKE_ELEMLIST(name, type) \  
    if (in.name.length() == 0) \  
        elem.name = NULL; \  
    else \  
        elem.name = new type(in.name)
```

Die Umsetzung von Unions

Die Umsetzung der Unions in Melody erfolgt im wesentlichen ähnlich wie die Umsetzung anderer Objekte auch. Was an dieser Stelle noch hinzukommt, ist die Auswahl der korrekten Variante, um die Daten richtig zu lesen oder zu schreiben. Wenn aus dem C++-Objekt die CORBA-Struktur erstellt werden soll, so gibt es noch ein zusätzliches Problem. Um in dieser Struktur die passende Variante zu setzen, muß eine zur Variante gehörende Methode aufgerufen werden und dieser als Parameter die ausgefüllte CORBA-Struktur übergeben werden. Diese Aufrufmethodik widerspricht leider der bisherigen Strategie, daß alle *make_oo* Methoden als Referenzparameter das Ziel der Umwandlung bekommen. Deshalb muß zunächst diese Umwandlung in eine lokale Variable erfolgen diese in die Union gesetzt wird.

```
switch (elem.property.type) {
case STT_NODE: {
    STAttributeT_struct_oo stt;

    ((STAttributeT *) elem.property.u.stt)->make_oo(stt);

    out.property.stt(stt);
    break;
}
...

```

In diesem Fall erfolgt die Umsetzung zunächst in die lokale Variable *stt* hinein. Der Aufruf an das Objekt zur Konvertierung erfolgt dabei in gewohnter Weise. Wie bereits bei der Umsetzung der Listen erklärt wurde, muß auch an dieser Stelle ein *Cast* erfolgen. Nach dem Aufruf stehen in der Variable alle notwendigen Informationen. Nun können diese in die Union eingetragen werden, indem die entsprechende Methode aufgerufen wird. Da beim Setzen in die Union eine Kopie der ursprünglichen Datenstruktur angelegt wird, gibt es keine Probleme, wenn der Speicher der lokalen Variable am Ende des Blocks automatisch wieder freigegeben wird.

Dieser Umstand fällt in der Gegenrichtung weg. Dort kann je nach Diskriminante der Union der entsprechende Konstruktor des enthaltenen Objekts aufgerufen werden und direkt in die C++-Union eingetragen werden, da C++ ja keinerlei Überprüfung in dieser Richtung vornimmt. Die folgenden Zeilen sind also völlig ausreichend:

```
// Setze Diskriminate in C++
elem.property.type = (node_type) in.property._d();

// konvertiere den Inhalt der Union
switch (elem.property.type) {
case STT_NODE:
    elem.property.u.stt = new
        STAttributeT(in.property.stt());
    break;
...

```

Selbstverständlich darf nicht vergessen werden, die Diskriminante in C++ zu setzen, da diese ja explizit verwaltet wird, während dies in CORBA implizit abläuft. Da in der vorliegenden Version die Unions nur an zwei Stellen explizit verwendet werden und auch in der Schnittstellen-

beschreibung keine Makros für sie definiert sind, wird auch hier die Umwandlung direkt ohne die weitere Hilfe von Makros durchgeführt.

Die Umsetzung der OPTSTRUCPs

Als letztes Konstrukt, das noch umgewandelt werden muß, bleiben die „*OPTSTRUCPs*“ noch übrig. Dieses Makro wird in der Schnittstellenbeschreibung dafür verwendet, um bei Zeigern auf andere Strukturen auch den *NULL*-Zeiger geeignet modellieren zu können. In der Tat handelt es sich bei der CORBA-Struktur um eine Union, die wie eben beschrieben behandelt werden kann. Da jedoch für die „*OPTSTRUCPs*“ im Gegensatz zu den „reinen“ Unions ein zusätzliches Makro in der Schnittstelle definiert wurde, wird dies ebenso für die Umwandlung gemacht. Der Aufbau der Makros sollte nach den bisherigen Erläuterungen einleuchtend sein:

```
#define MAKE_OPT(name, type) \
    if (name == (type *) NULL) \
        out.name.dummy(FALSE); \
    else { \
        type ## _struc_oo    t; \
\
        ((type *) name)->make_oo(t); \
\
        out.name.name(t); \
    }

#define MAKE_ELEMOPT(name, type) \
    if (elem.name == (type *) NULL) \
        out.name.dummy(FALSE); \
    else { \
        type ## _struc_oo    t; \
\
        ((type *) elem.name)->make_oo(t); \
\
        out.name.name(t); \
    }
```

Ähnlich wie die Makros für die Listen benötigen auch diese Makros die Typbezeichnung um die Methoden des entsprechenden Objekts aufrufen zu können. Wenn kein *NULL*-Zeiger übergeben wurde, muß die Umwandlung zunächst in einer lokalen Variable abgelegt werden, bevor die Union gesetzt werden kann. Im anderen Fall kann die Dummy-Variante direkt gesetzt werden.

```
#define TAKE_OPT(name, type) \
    if (in.name._d()) \
        name = new type(in.name.name()); \
    else \
        name = (type *) NULL

#define TAKE_ELEMOPT(name, type) \
    if (in.name._d()) \
        elem.name = new type(in.name.name()); \
```

```
else \  
    elem.name = (type *) NULL
```

Die Gegenrichtung gestaltet sich noch einfacher, da es sich bei der eingeführten Union auf CORBA-Seite um ein Hilfskonstrukt handelt, das nicht in C++ abgebildet wird. Statt dessen kann je nach Diskriminante ein neues Objekt angelegt werden oder der *NULL*-Zeiger gesetzt werden. Wie bereits an anderen Stellen auch, werden die „ELEM“-Makros in Listen verwendet, während sonst die „normalen“ Makros benutzt werden.

```
// Beispiele nach CORBA  
MAKE_OPT(cBlock, CBlock);  
MAKE_ELEMOPT(rule, SRule);  
  
// und von CORBA  
TAKE_OPT(cBlock, CBlock);  
TAKE_ELEMOPT(rule, SRule);
```

Problemlösungen

Leider stellt sich heraus, daß der vom Orbix IDL-Compiler erzeugte Programmcode einen Fehler enthielt. Dadurch war es nicht möglich eine Union korrekt zu übertragen, in der eine Variante gesetzt war, die aus einem einfachen Datentyp wie einem *boolean* oder *long* bestand. Da aber gerade dies an einigen Stellen, nämlich für die Dummy-Variante zur *NULL*-Zeigerdarstellung, zum Einsatz kam, mußte ein Weg um dieses Problem herum gefunden werden, bis eine neuere Version des Orbix Compilers vorliegt, in der dies behoben ist. Eine genauere Untersuchung des Problems ergab, daß das Attribut *isSet* im Union-Objekt nicht gesetzt wird, wenn die Dummy-Variante benutzt wird. Für die *clientID_union* erzeugt der IDL-Compiler zum Beispiel folgenden Code: (in **fett** bereits die fehlende Anweisung, wie sie durch das *sed*-Skript eingefügt wird)

```
void dummy (CORBA::Boolean IT_member) {  
    if (isSet && (__d != 0))  
    {  
        this-> MELODY_trader::SSearchReturn_el_struct_o::  
            clientID_union::~~clientID_union();  
        memset(this, 0, sizeof(*this));  
    }  
  
    __d = 0;  
    _dummy_ = IT_member; isSet = 1;  
}
```

Da die Zeilen die betroffenen Zeilen in *TRD_tcomm.hh* durch die eindeutige Benennung der Variante mit *dummy* klar zu identifizieren waren, konnte mit Hilfe eines einfachen *sed*-Skripts Abhilfe geschafft werden. Folgende Anweisung reicht dazu aus:

```
sed -e 's/_dummy_ = IT_member;/_dummy_ = IT_member; isSet = 1;/g' \  
    $(VS_SNAME)/TRD_tcomm.hh.ORIG > $(VS_SNAME)/TRD_tcomm.hh
```

Das Skript ersetzt einfach den Text zwischen den ersten beiden Schrägstrichen durch den Text, der darauf folgt. Auf diese Weise wird das Attribut korrekt gesetzt und die Unions können normal verwendet werden.

Die im letzten Kapitel beschriebenen Konvertierungsmethoden bilden einen wesentlichen Aspekt der CORBA-Schnittstelle für Melody. Im nun folgenden letzten Schritt wird die Funktionalität an der Nahtstelle zwischen CORBA und Melody beschrieben: Dort, wo der Kontrollfluß im *Trader User Agent* Melody verläßt und im *Client-Stub* die Übertragung startet, und auf der Serverseite, wo vom *Skeleton* heraus der *Trader System Agent* aufgerufen wird. An diesen beiden Übergangsstellen müssen entsprechende Methoden dafür sorgen, daß die notwendigen Konvertierungen richtig ablaufen. Auf der Clientseite müssen die übergebenen Parameterobjekte nach CORBA übersetzt werden, bevor sie übertragen werden können. Auf der Serverseite werden aus diesen empfangenen Strukturen wieder die Melody-Parameterobjekte erstellt, die dem *TSA* übergeben werden. Die Ergebnisse des Trader-Aufrufes beschreiten den gleichen Weg in umgekehrter Richtung: Sie werden auf Serverseite nach CORBA übersetzt, übertragen und schließlich auf Clientseite in neu erstellten Melody-Resultatobjekten zurückgegeben.

Der Vorzug dieser Lösung liegt klar auf der Hand: Durch Trennung der CORBA-Funktionalität vom eigentlichen Trader sind keine Anpassungen am *TSA* notwendig. Der *Objekt-Adapter* übernimmt diese Anpassungen zwischen CORBA-Objektschnittstelle und Melody-Schnittstelle. Durch Verwendung eines anderen Objekt-Adapters können andere Übertragungsprotokolle in gleicher Weise eingebunden werden. Die DCE-Schnittstelle von Melody ist ebenfalls so implementiert. Da die Objekt-Adapter voneinander unabhängig sind und alle gemeinsam auf einen einzelnen *TSA* zugreifen können, ist es ohne weiteres möglich, auch mehrere, verschiedene Adapter gleichzeitig zu installieren. Damit kann auf einen einzelnen Trader sowohl über DCE als auch über CORBA zugegriffen werden.

Der Objekt-Adapter hat also eine wichtige Aufgabe: er versteckt vor dem Benutzer wie auch vor dem Trader selbst das verwendete Übertragungsprotokoll. Der Benutzer greift in gewohnter Weise auf den *TUA* zu, ohne selbst zu wissen, ob zur Übertragung DCE oder CORBA verwendet wird, oder ob der Aufruf möglicherweise sogar lokal erfolgt. Auf der anderen Seite wird der *TSA* immer über dieselbe Schnittstelle aufgerufen, ohne daß eine Unterscheidung erfolgt, auf welchem Wege der Aufruf ankam oder wie die Ergebnisse zurück zum Benutzer kommen.

In diesem Kapitel soll nun die Implementation des CORBA-Objekt-Adapter erläutert werden. Sie besteht im wesentlichen aus drei Teilen: auf Serverseite aus dem *Frontend* und den Objekt-Adapter-Klassen, sowie auf Clientseite aus dem *CORBA-TUAgent*. Diese Einteilung leitet sich voll von der vorhandenen Implementierung der DCE-Schnittstelle ab, die sich in gleicher Weise zusammensetzt.

Zunächst wird die Clientseite betrachtet und an einem Beispiel erläutert, wie die notwendigen Konvertierungen veranlaßt werden. Anschließend werden weitere Aspekte erläutert, wie zum Beispiel die Frage, wie die Verbindung zum ORB hergestellt wird. Die Beschreibung der Serverseite erfolgt in ähnlicher Weise. Erst wird wieder die Verwendung der Konvertierungsmethoden erklärt, bevor die „Verwaltungsaspekte“ innerhalb der Objekt-Adapter-Klassen erläutert werden.

Mit diesem Kapitel wird die Beschreibung der CORBA-Schnittstelle für Melody unter C++ abgeschlossen. Ausgehend von der Umsetzung der Schnittstellenbeschreibung für CORBA-IDL wurden die Konvertierungsmethoden erläutert, die im hier beschriebenen Objekt-Adapter zum Einsatz kommen. Der Zugriff auf Melody ist damit aber nicht auf C++ beschränkt. Ein Klient kann in einer beliebigen Programmiersprache entwickelt sein, solange für diese Sprache ein CORBA-Mapping spezifiziert ist und er auf einen ORB zugreifen kann. Einschränkend muß allerdings festgestellt werden, daß der Zugriff in einer anderen Sprache etwas beschwerlicher ist, solange dort kein *TUA* implementiert wird, der die Erstellung der Parameterobjekte erleichtert. Stattdessen muß der Klient in der Lage sein, alle Übertragungsstrukturen selbst korrekt auszufüllen.

Der CORBA-TUA

Der TUA ist die Schnittstelle zwischen dem Klienten und Melody. Je nach Realisierung der Kommunikation existieren verschiedene Implementationen. Für die CORBA-Schnittstelle von Melody wurde ein TUA entwickelt, der dem Klienten den Zugriff über CORBA gestattet. Seine Aufgabe ist klar umgrenzt: Er bietet dem Klienten das gewohnte Trader-Interface an und setzt alle Methodenaufrufe in entsprechende CORBA-Aufrufe um. Dabei muß er alle Parameter in CORBA-Strukturen konvertieren und in umgekehrter Richtung aus den empfangenen Daten wieder Melody-Ergebnisobjekte erstellen. Zu dieser Aufgabe kommt noch der Verbindungsaufbau über CORBA hinzu. Beide Aspekte sollen in diesem Abschnitt erläutert werden.

Die Klasse TUAgent_oo

Die Schnittstelle des Traders in Melody ist in der Klasse *Trader_Interface* definiert. Sie enthält Definitionen aller notwendigen Methoden mit ihren Parametern. Diese Klasse wird allerdings nicht direkt verwendet, sondern sie stellt die Basis-Klasse für die beiden wesentlichen Teile dar: dem *TUA* und dem *TSA*. Der *TSA* ist auf Serverseite die Schnittstelle zum Trader, der *TUA* ist auf Clientseite die Schnittstelle vom Klienten. Durch die gemeinsame Basis-Klasse ist sichergestellt, daß sowohl *TUA* als auch *TSA* dieselben Methoden anbieten. Der Klient greift also auf den *TUA* genauso zu, wie wenn er den *TSA* direkt lokal ansprechen könnte.

Für die DCE-Schnittstelle wurde eine TUA-Klasse namens *TUAgent* implementiert, die den Zugriff auf Melody über DCE ermöglicht. In Analogie zur bisherigen Benennung der CORBA-Strukturen und Methoden wurde die Klasse des CORBA-TUA *TUAgent_oo* genannt. Sie leitet sich ebenfalls von der Klasse *Trader_Interface* ab. Sie enthält damit folgende Methoden:

- *TUAgent_oo*, initialisiert den TUA und setzt den Server-Namen.
- *prepareConnection*, baut die Verbindung mit dem Server auf. Diese Methode wird von allen anderen Methoden bereits selbst verwendet, so daß sie kaum vom Klienten aufgerufen werden sollte.
- *insertType*, Einfügen eines Diensttyps
- *deleteType*, Löschen eines Diensttyps
- *modifyType*, Ändern eines Diensttyps
- *listTypes*, Abfragen von Diensttypen. Es gibt zwei Methoden: eine gibt eine Liste aller passenden Typen zurück, eine alle Details eines einzelnen Typs.
- *insertContext*, Einfügen eines Kontexts
- *deleteContext*, Löschen eines Kontexts
- *listContexts*, Abfragen von Kontexten. Wieder in zwei Versionen: Einmal eine „Übersichtsliste“ und einmal einen einzelnen Kontext.
- *insertService*, Einfügen eines Dienstes
- *deleteService*, Löschen eines Dienstes
- *modifyService*, Ändern eines Dienstes
- *listService*, Abfragen von Diensten
- *searchService*, Suchen von Diensten
- *selectService*, Suchen und Auswählen eines Dienstes
- *announceTrader*, Anmelden eines Traders zur Kooperation
- *coopMessage*, Austauschen von Nachrichten zur Kooperation
- *getHost*, eine Hilfsmethode, die den Namen des Rechners zurückgibt, auf dem der Melody-Trader läuft.
- *isAvailable*, überprüft die Verfügbarkeit des Traders

Alle Methoden überprüfen vor ihrem Zugriff auf Melody, ob die Verbindung bereits aufgebaut wurde. Wenn dies nicht der Fall ist, wird dies zuerst mit Hilfe von *prepareConnection* getan. Diese Aufzählung der Methoden soll nur eine Übersicht sein. Für die genaue Verwendung und Bedeutung der Parameter sei auf die Melody-Dokumentation bzw. verschiedene Test- und Beispielprogramme von Melody verwiesen.

Konvertierung der Aufrufe

Anhand des Beispiels der Methode *insertType* soll hier die Umsetzung eines Aufrufs des TUA erläutert werden. Er enthält bereits alle wesentlichen Aspekte wie zum Beispiel die Konvertierung der Parameter in beiden Richtungen.

```
TSResult
TUAgent_oo::insertType(const STInsert *newType,
                       ErrorMessage* &returnMes)
{
    STInsert_struct_oo      nt;
    ErrorMessage_struct_oo_var errmsg;
    TSResult_oo             result;

    if(prepareConnection(returnMes) != comOK)
        return(tsFatalError);

    newType->make_oo(nt);

    try {
        result = trader->INSERTTYPE(nt, errmsg);
    }
    catch(const CORBA::SystemException& se) {
        returnMes = new ErrorMessage;
        returnMes->append(
            "TUAgent_oo Error: Can't call TAgent for INSERTTYPE"
        );
        connected = 0;
        cerr << "Call to INSERTTYPE failed" << endl;
        cerr << "Unexpected exception:" << endl << &se;
        return(tsFatalError);
    }

    returnMes = new ErrorMessage(errmsg);
    return (TSResult) result;
};
```

Als erstes wird in jeder Methode immer überprüft, ob die Verbindung zum Server bereits aufgebaut ist. *prepareConnection* übernimmt diese Aufgabe. Wenn eine Verbindung besteht, dann kann der Aufruf abgesetzt werden. Dazu müssen natürlich zunächst alle Parameter konvertiert werden. *insertType* hat als einzigen Parameter die Typspezifikation des neuen Diensttyps. Der zweite Parameter wird erst später für die Rückgabe von Fehlermeldungen benötigt. Die Typspezifikation wird durch Aufruf der *make_oo*-Methode des übergebenen Parameterobjekts in CORBA umgesetzt. Das Ergebnis wird in die übergebene Struktur geschrieben. Nach dieser Konvertierung kann der Aufruf der Methode im *Client-Stub* von CORBA erfolgen.

In der Schnittstellenbeschreibung wurden alle Methoden des Traders in entsprechende Methoden abgebildet, die denselben Namen in Großbuchstaben haben. Die Methode *INSERTTYPE* hat wie *insertType* auch, die gleichen Parameter in der gleichen Reihenfolge.¹ Im zweiten Parameter wird später die Fehlermeldung vom Trader zurückgegeben. Es handelt sich hier also um einen *OUT*-Parameter aus dem IDL-File. Um diesen Parameter korrekt zu verwenden, kann

1. Genaugenommen existiert noch eine weiterer Parameter. In diesem Parameter können noch weitere Informationen für CORBA oder den Server übergeben werden, die aber hier nicht benutzt werden.

nicht direkt eine Variable des Typs *ErrorMessage_struct_oo* übergeben werden. CORBA benötigt an dieser Stelle einen Platzhalter für eine Objektreferenz, die zurückgegeben wird. Der Name einer Referenz leitet sich immer aus dem ursprünglichen Namen durch anhängen von *_var* ab, so daß an dieser Stelle eine Variable des Typs *ErrorMessage_struct_oo_var* benutzt wird.

Die Methoden des Stubs finden sich in einem Objekt der Klasse *MELODY_trader*, das im Attribut *trader* in einer Objektreferenz des TUA gespeichert wird. Dieses Objekt wird bei der Bindung an den Server erzeugt und dort von der TUA-Methode *prepareConnection* abgelegt. (siehe unten)

Sollte während des Aufrufes von CORBA ein Fehler auftreten, so wird dies durch eine *Exception* vom Typ „*SystemException*“ signalisiert. In diesem Fall wird eine Fehlermeldung erzeugt und der Aufruf abgebrochen. Bei erfolgreicher Ausführung steht in *errmsg* die Fehlermeldung von Melody. Diese Fehlermeldung muß in das entsprechende Melody-Objekt konvertiert werden. Diese geschieht durch Verwendung des entsprechenden Objekt-Konstruktors. Im letzten Schritt wird der Rückgabewert in den richtigen Typ konvertiert und an den Klienten übergeben. Für diese Umwandlung genügt ein einfacher *Cast*, weil CORBA und C++ für die Numerierung von *enums* die gleiche Konvention verwenden.

Der Verbindungsaufbau – prepareConnection

Der Verbindungsaufbau erfolgt, wie bereits erklärt, in der Methode *prepareConnection*, die vor jeder anderen Benutzung von CORBA aufgerufen wird. Ihre Aufgabe beschränkt sich darauf, die Bindung des TUA an den Melody-Server über CORBA zu bewerkstelligen. Bei erfolgreicher Verbindung wird dies entsprechend zurückgegeben. In diesem Fall wird im Attribut *trader* die Objektreferenz abgelegt, die für die weitere Verwendung benötigt wird.

Da *prepareConnection* immer vor jeder Ausführung einer anderen CORBA-Methode aufgerufen wird, muß der Verbindungszustand in einem weiteren Attribute *connected* gespeichert werden. Besteht bereits eine Verbindung, dann meldet die Methode direkt Erfolg. Falls ein Fehler bei der Verbindung auftritt, wird eine Fehlermeldung im Parameter *returnMes* zurückgeben. Insgesamt sieht die Methode folgendermaßen aus:

```
COMResult
TUAgent_oo::prepareConnection(ErrorMessage* &returnMes)
{
    returnMes = NULL;

    if(connected == 1) return(comOK) ;

    try {
        trader = MELODY_trader::_bind(ENTRY_NAME, "cembalo");
    }
    catch(const CORBA::SystemException& se) {
        returnMes = new ErrorMessage;
        returnMes->append(
            "TUAgent_oo Error: Bind to %s failed", ENTRY_NAME);
        cerr << "Bind to object failed" << endl;
    }
}
```

```
        cerr << "Unexpected exception:" << endl << &se;
        return(comFatalError);
    }

    connected = 1;
    return(comOK);
};
```

Das Frontend

Auf der Serverseite besteht der Objekt-Adapter im wesentlichen aus 2 Teilen. Die erste der beiden ist das *Frontend*. Der Name leitet sich von der Benennung der DCE-Schnittstelle ab, wo eine ähnliche Funktionalität in *TRD_frontend.C* vorhanden ist. Das Frontend ist die Übergabestelle zwischen CORBA und Melody. Wenn ein Zugriff auf den Trader über CORBA erfolgt, dann wird im Frontend eine Methode aufgerufen, die alle übertragenen Daten entsprechend der Schnittstellenbeschreibung übergibt. Das Frontend übernimmt nun die notwendigen Anpassungen, damit der Aufruf an die richtige Methode des *TSA* erfolgen kann. Er wandelt also die hereinkommenden Datenstrukturen in Melody-Objekte um, ruft dann den Trader auf, nimmt die Ergebnisse von ihm wieder entgegen und wandelt sie in CORBA zurück.

Die Aufgabe läßt sich stark mit der Funktionalität des CORBA-TUAs vergleichen. Er ist das Gegenstück auf der Serverseite und bedient sich damit der selben Techniken, nur in umgekehrter Reihenfolge. Wie beim TUA auch, so erfolgt auch hier die Erläuterung direkt an einem Beispiel.

MELODY_traderBOAImpl ist eine Klasse in den vom IDL-Compiler generierten Header-Dateien. Sie stellt die Basisklasse für alle Implementationen des Melody-Interfaces dar und sie enthält die Deklarationen aller Zugriffsmethoden, wie sie in der Schnittstellenbeschreibung vorgegeben wurde. Diese Informationen sind damit bereits sehr hilfreich, um die Signaturen der zu implementierenden Methoden kennenzulernen. Die getrennte Definition der Klasse könnte zum Beispiel dazu verwendet werden, um innerhalb eines einzigen Programms verschiedene Objektimplementierungen anzubieten.

Von dieser automatisch erstellten Klasse wird in der Datei *TRD_frontend_oo.h* die Klasse der tatsächlichen Objektimplementierung abgeleitet. Sie wurde – in Anlehnung an die Beispiele von Orbix – *MELODY_trader_i* genannt. Aufgabe war es nun, alle Methoden dieser Klasse zu implementieren. Anhand des Beispiel der Methode für das Einfügen neuer Dienstypen sollen einige Teile erläutert werden.

```
MELODY_trader::TSResult_oo
MELODY_trader_i::INSERTTYPE (
    const MELODY_trader::STInsert_struct_oo& newType_oo,
    MELODY_trader::ErrMsg_struct_oo*& errmsg_oo,
    CORBA::Environment &IT_env
)
    throw (CORBA::SystemException)
{
```

```
TSResult_oo          result_oo;
TSResult             result;
STInsert             sTI(newType_oo);
ErrorMessage         *errmsg = NULL;

result = tSAgent->insertType(&sTI, errmsg);

// Convert error message for CORBA
errmsg_oo = new ErrorMessage_struct_oo;
if (errmsg) {
    errmsg->make_oo(*errmsg_oo);
    delete errmsg;
}

result_oo = (TSResult_oo) result;
return(result_oo);
}
```

Die Methode bekommt von CORBA drei Parameter. Die ersten beiden sind die Argumente, wie sie in der Schnittstellenbeschreibung vorgegeben wurden: Zunächst die Beschreibung des einzufügenden Dienstyps und an zweiter Stelle der Rückgabeparameter für die Fehlermeldung. Im dritten Parameter wird die CORBA-Umgebung übergeben. In ihr können weitere Informationen zwischen Client, ORB und Server ausgetauscht werden. Sie wurden aber in der vorliegenden Realisierung nicht benötigt.

Ähnlich wie schon im TUA ist es wieder der erste Schritt, aus den hereingekommenen Parametern die Melody-Objekte zu erstellen. In diesem Beispiel wird bereits mit Anlegen des Objekts in *sTI* die Konvertierung vorgenommen. Der zweite Parameter der Methode ist ein *OUT*-Parameter, das heißt er wird ausschließlich zur Rückgabe von Daten verwendet. Deshalb ist für ihn zunächst keine Konvertierung notwendig.

Da jetzt bereits alle Datenstrukturen, die für den Aufruf übergeben werden müssen, konvertiert sind, kann sich sofort der Aufruf der *TSA*-Methode anschließen. Der Trader gibt nach erfolgter Bearbeitung in *errmsg* Fehlermeldungen zurück, die jetzt weiter an den Klienten übertragen werden müssen. Dazu müssen sie zunächst einmal wieder in die entsprechende CORBA-Struktur umgewandelt werden. CORBA erwartet, daß der Speicher für den *OUT*-Parameter allokiert wird. Es gibt ihn der *make_oo*-Methode der *ErrorMessage*-Klasse wird die Konvertierung veranlaßt. Damit sind die Fehlermeldungen alle in die CORBA-Struktur kopiert worden und das Melody-Objekt kann gelöscht werden, da es nicht mehr benötigt wird.

Als letzten Schritt in dieser Methode muß noch der Rückgabewert in den richtigen Typ konvertiert werden. Es genügt hier wieder ein einfacher Cast, da die Symbole in Aufzählungstypen sowohl in C++ als auch in CORBA in gleicher Weise durchnummeriert werden.

Wie zu sehen ist, wird durch das Frontend die Trennung der CORBA-Schnittstelle vom eigentlichen Trader erreicht. Alle notwendigen Umwandlungen, die aufgrund von CORBA notwendig sind, erfolgen an dieser Stelle. Der Trader merkt davon nichts und wird an seiner gewohnten Schnittstelle aufgerufen.

Die Objekt-Adapter-Klassen

Damit fehlt jetzt nur noch ein letzter Baustein für die volle Einsatzfähigkeit der CORBA-Schnittstelle: Die Komponente, die die Objektimplementierung aktiviert und diese beim ORB anmeldet. Auch dieser Teil wurde wieder in Anlehnung an die bereits vorhandene DCE-Schnittstelle implementiert. Dort fand sie sich in den Klassen *TRD_DCE_OA*, die grundlegende Methoden definierte, und *TRD_DCE_OA_TSA*, in der die konkrete Einbindung in DCE erfolgte. Um aber eine klare, eindeutige Aufgabenteilung zu haben, waren einige Änderungen notwendig, die mit verschiedenen Umstellungen verknüpft waren. Unter anderem mußten Aufgaben, wie die Initialisierung des *Trader System Agents* natürlich herausgenommen werden, da dies die Aufgabe einer allgemeinen Initialisierungsfunktion ist und nichts mit der DCE-Schnittstelle an sich zu tun hat. Sie wurde deshalb an dieser Stelle bereinigt.

Bei der Umstellung dieses Teils wurden die Aufgaben vom allgemeinem Objekt-Adapter-Teil und vom für die Serverseite spezifischen Teil getrennt. Dabei wurde auch berücksichtigt, daß später vielleicht ein Analogon für die Clientseite entwickelt wird, das die Aufgaben der *TUAgent*-Klassen übernimmt. Um die Einheitlichkeit aller Objekt-Adapter-Klassen zu gewährleisten, wurden deshalb zunächst einmal abstrakte Basis-Klassen definiert, die alle benötigten Methoden deklarieren.

Der allgemeine Objekt-Adapter TRD_OA

Ausgangspunkt aller Objekt-Adapter-Klassen ist die neue Klasse *TRD_OA*. In ihr finden sich die Deklarationen der Methoden, die sowohl auf Server- als auch auf Clientseite benötigt werden. Zur Zeit finden sich hier zwei Methoden, die sich mit der Benennung der Schnittstelle beschäftigen, sowie eine weitere Methode, deren Anwendung ausschließlich für die Ausgabe von Versionsmeldungen gedacht ist.

```
class TRD_OA {
public:
    TRD_OA() {;}

    virtual const char *OAtype()=0;

    virtual const char *short_name()=0;
    virtual const char *address()=0;
};
```

Die ersten beiden Methoden sind *address* und *short_name*. Ihre Aufgabe ist die eindeutige Identifizierung des Servers für die entsprechende Schnittstelle. Dabei wird von *address* eine eindeutige Adresse erwartet, mit Hilfe derer der Objekt-Adapter in der Lage ist, den entsprechenden Server wiederzufinden. Benötigt wird sie zum Beispiel beim Einsatz eines kooperativen Traders, dem diese übergeben wird. *short_name* soll einen Kurznamen liefern, mit dem sich der Server beim benutzten System, also zum Beispiel DCE oder CORBA, anmeldet. Die letzte Methode mit dem Namen *OAtype* soll eine Identifizierung des Objekt-Adapters liefern, die für Kontrollausdrucke oder ähnliches verwandt werden kann. Texte wie „MELODY CORBA2 1.2“ sind denkbar, die den Typ des Objekt-Adapters und seine Version beinhalten. Diese Informationen können insbesondere sehr hilfreich sein, wenn die entsprechenden Teile

über dynamische Libraries geladen werden und möglicherweise zwischenzeitlich ersetzt wurden.

Der serverseitige Objekt-Adapter TRD_OA_TSA

Von dieser Objekt-Adapter-Klasse abgeleitet wird die Basis-Klasse für die Serverseite. Sie wurde in Anlehnung an die vorhandenen Klassen *TRD_OA_TSA* genannt. Sie erweitert den allgemeinen Objekt-Adapter um Methoden, die notwendig sind für die Initialisierung, die Aktivierung und die Beendigung der entsprechenden Schnittstelle. Sie sieht im Detail so aus:

```
class TRD_OA_TSA : public virtual TRD_OA {
public:
    TRD_OA_TSA() {;};

    virtual void parse_args (const int argc,
                            const char *argv[],
                            int consumed[])=0;

    virtual int init ()=0;
    virtual int terminatewait (pthread_t)=0;
    virtual int wait ()=0;
    virtual int shutdown ()=0;
};
```

Die erste Methode *parse_args* erlaubt die Konfigurierung des Objekts-Adapters beim Start des Traders über die Argumente auf der Befehlszeile. Um bereits benutzte Argumente markieren zu können, wird zusätzlich das Feld *consumed* übergeben, das für jedes Argument einen Eintrag vorsieht. Ist dieser Eintrag 0, so wurde es bereits von einem anderen Objekt-Adapter oder vom allgemeinen Teil benutzt und sollte deshalb nicht mehr interpretiert werden. Im anderen Fall kann der Inhalt untersucht werden. Handelt es sich um ein Argument für diesen Adapter, so setzt er den Eintrag auf einen Wert ungleich 0.

In der Methode *init* hat Funktionalität Platz, die zur Vorbereitung der Schnittstelle benötigt wird. In der DCE-Schnittstelle erfolgt hier zum Beispiel die Registrierung des Interfaces. Liefert diese Methode als Wert nicht 0, so wird von einem Fehlerzustand ausgegangen, der die Arbeit dieses Objekt-Adapters verhindert. Er wird deshalb nicht aktiviert, sondern es erfolgt im nächsten Schritt ein Aufruf an *shutdown*.

Die Aktivierung der Schnittstelle im System erfolgt nach der Initialisierung durch den Aufruf der *wait* Methode. Ihre Aufgabe ist es, daß die Schnittstelle auf Aufträge wartet, sie entgegennimmt, bearbeitet und die Ergebnisse wieder zurücküberträgt. Wenn diese Methode zurückkehrt, ist die Arbeit des Objekt-Adapters beendet und es folgt der Aufruf an die *shutdown* Methode.

Um den Trader auch in besonderen Fällen beenden zu können, muß der Objekt-Adapter durch die Methode *terminatewait* die Möglichkeit bieten, eine aktivierte Schnittstelle wieder zu deaktivieren. Wenn sie aufgerufen wird, so muß sie dem unter der als Parameter übergebenen Threadnummer laufenden *wait* signalisieren, daß die Bearbeitung beendet werden soll, und

danach wieder zurückkehren. Sobald es möglich ist, muß daraufhin *wait* die Arbeit abbrechen und ebenfalls zurückkehren.

Sobald dies der Fall ist, wird die letzte der Methoden *shutdown* aufgerufen, um notwendige „Aufräumarbeiten“ zu erledigen. Der DCE-Objekt-Adapter löscht dort zum Beispiel – in Umkehrung zur Aufgabe von *init* – die Registrierungen des Servers in DCE. Mit der Rückkehr von dieser Methode ist die Arbeit des Objekt-Adapters beendet.

Die CORBA-Objekt-Adapter-Klassen

Aus den beiden gerade beschriebenen allgemeinen Objekt-Adapter-Klassen werden nun zwei entsprechende Klassen für die CORBA-Schnittstelle abgeleitet. Analog zur vorhandenen Benennung der DCE-Schnittstelle wurden die Klassen *TRD_CORBA_OA* und *TRD_CORBA_OA_TSA* genannt. Im Folgenden soll kurz auf die Realisierung der einzelnen Methoden eingegangen werden.

Für die Realisierung von *short_name* wurde der Name verwendet, unter dem sich ein Server beim ORB anmeldet und der in gleicher Weise auch zum Auffinden vom Klienten benutzt werden kann. Dieser Name identifiziert den Server. Bei der Anmeldung registriert der ORB die neue Implementierung mit diesem Servernamen. Wenn nun ein Klient über diesen Namen den Trader verwenden will, so sucht der ORB nach einem Server, an den er den Aufruf weiterleiten kann.

Die exakte Zuordnung zu einer Implementierung geschieht über die Adresse, die von *address* geliefert wird. Sie muß deshalb weitere Informationen enthalten, die einen Server von anderen unterscheiden. Zu diesem Zweck bietet der ORB eine Methode an, die es ermöglicht, aus einer Objektreferenz ein eindeutigen String zu erzeugen. Dieser Aufruf wird im Konstruktor von *TRD_CORBA_OA_TSA* verwendet, um diesen String dann zwischenzuspeichern.

```
TRD_CORBA_OA_TSA ( ) : TRD_CORBA_OA ( ) {  
    my_full_name = CORBA::Orbix.object_to_string(&myTrader);  
};
```

Die Methode *OAtype* gibt den festen Text „*MELODY Orbix2/Corba2 1.2*“ zurück, wobei die Versionsnummer die SCCS-Version der benutzten Quelldatei ist. Sie erhöht sich damit automatisch, wenn Änderungen am Objekt-Adapter vorgenommen werden.

Der serverseitige Objektadapter erlaubt als Befehlszeilenargument „-C“, mit dem der Servername geändert werden kann. Er wird direkt, ohne Leerzeichen angeschlossen. Wird der Servername nicht mit dieser Option gesetzt, so wird der Inhalt der Environment-Variable *TRADER_DOM* benutzt. Ist diese Variable ebenfalls nicht gesetzt, so verwendet der Objekt-Adapter standardmäßig den Namen „*MELODY*“.

Die *init* Methode des CORBA-Objekt-Adapters hat keine Aufgabe. Sämtliche Registrierung beim ORB oder im Interface Repository geschieht implizit an anderen Stellen, so daß *init* nur ein Aufruf an die Methode *banner* enthält, die eine Kopfzeile auf dem Bildschirm ausgibt.

```
int TRD_CORBA_OA_TSA :: wait ()
{
    try {
        CORBA::Orbix.impl_is_ready(my_short_name,
                                   CORBA::Orbix.INFINITE_TIMEOUT);
    }
    catch(const CORBA::SystemException& se) {
        cerr << "Unexpected Exception: " << endl << &se;
    }
    return (0);
}
```

In der *wait* Methode muß dem ORB nur noch mitgeteilt werden, daß die Objekt-Implementierung bereit ist und unter dem Servername registriert wird, der wie oben beschrieben bestimmt wurde. Durch den zweiten Parameter beim Aufruf an *impl_is_ready* wird angegeben, daß die Implementation nicht nach einiger Zeit der Inaktivität beendet werden soll. Da der Trader üblicherweise ständig bereit sein soll, muß dies so geschehen. Die Methode kehrt in diesem Fall erst dann wieder zurück, wenn sie durch entsprechenden Aufruf der Methode *deactivate_impl* des ORBs deaktiviert wird, wie dies in *terminatewait* geschieht.

```
int TRD_CORBA_OA_TSA :: terminatewait (pthread_t thr)
{
    CORBA::Orbix.deactivate_impl(my_short_name);
    return(0);
}
```

Die letzte, noch fehlende Methode des serverseitige Objekt-Adapters ist *shutdown*. Auch diese Methode hat, wie *init* auch, bei CORBA keine Aufgabe, da die Abmeldung des Servers bereits durch den Aufruf von *deactivate_impl* geschehen ist.

Damit ist die Beschreibung der Integration der CORBA-Schnittstelle in Melody vollständig. Ausgehend von der Umsetzung der Schnittstellenbeschreibung für CORBA-IDL folgte die Erläuterung der Konvertierungsmethoden, die eine Umwandlung der Melody-Objekte in CORBA-Strukturen und umgekehrt ermöglichten. In diesem Kapitel folgten die Maßnahmen, die nötig waren, um die CORBA-Schnittstelle in Melody zu verwenden, ohne daß dabei direkt die Schnittstelle zum Trader oder zum Klienten verändert werden mußte.

Mit Abschluß der CORBA-Schnittstelle für Melody kann jetzt jeder Klient, der in der Lage ist, auf CORBA zu zugreifen, auch Melody benutzen. Er hat entweder die Möglichkeit, sich durch einen IDL-Compiler alle *Stubroutinen* für seine Programmiersprache erzeugen zu lassen, und so sehr bequem Aufrufe abzusetzen. Oder aber er benutzt das *Dynamic Invocation Interface* von CORBA und setzt alle Aufrufe mit den Parameter zur Laufzeit zusammen. Voraussetzung ist aber in beiden Fällen immer, daß ein *CORBA-Mapping* für seine Programmiersprache definiert wurde.

Die Firma Orbix bietet mit *OrbixWeb* einen IDL-Compiler für Java an. Da zur Zeit noch kein Java-Mapping standardisiert ist, mußte ein eigenes, vorläufiges definiert werden. Dieser IDL-Compiler liest genauso wie sein Pendant für C++ die Schnittstellenbeschreibung und erzeugt daraus alle notwendigen Klassen, um über den ORB auf CORBA-Objekte zuzugreifen. *OrbixWeb* ist allerdings in der ersten Version 1.0 nur in der Lage, die Anbindung auf Clientseite anzubieten. Es ist noch nicht möglich, unter Java auch einen Server zu implementieren. Da für diese Studienarbeit nur ein Java-Client erstellt werden mußte, bedeutet dies keine Einschränkung.

In diesem Kapitel soll das erstellte Java Interface beschrieben werden. Dazu wird zunächst wieder von der in Kapitel 7 umgesetzten Schnittstellenbeschreibung ausgegangen. Aus dieser werden mit dem OrbixWeb IDL-Compiler die benötigten Datenstrukturen¹ und Stubroutinen erzeugt. Es soll hier kurz auf die Aufrufparameter eingegangen werden. Leider stellte sich heraus, daß die erzeugten Programmteile fehlerhaft waren und sich nicht direkt vom Java Compiler übersetzen ließen. Mehrere kleine *sed*-Skripts beheben diese Fehler bis eine spätere Compiler-Version diese Probleme nicht mehr hat. Die Skripts werden in diesem Kapitel kurz erläutert.

1. Auch in diesem Kapitel werden die vom IDL-Compiler erzeugten Klassen, in Anlehnung an die vorherigen Kapitel, als „CORBA-Datenstrukturen“ bezeichnet, auch wenn es sich dabei natürlich ebenfalls um Java Klassen handelt.

Aufbauend auf die erzeugten Datenstrukturen wird der Java TUA entwickelt, der parallel zum im letzten Kapitel entwickelten CORBA TUA aufgebaut wurde. Um Anfragen an den Trader einfacher stellen zu können, wurden zusätzlich noch die Parameterobjekte aus C++ in Java nachgebildet, so daß die Verwendung des Java TUAs in gleicher Weise erfolgen kann wie bereits aus C++ bekannt ist. Diese Beschreibung bildet den Abschluß des Kapitels. Damit können beliebige Java Applikationen über den Java TUA auf Melody zugreifen.

Die Übersetzung der Schnittstellenbeschreibung

Der OrbixWeb IDL-Compiler erzeugt aus der Schnittstellenbeschreibung alle notwendigen Java-Klassen für die Clientseite. Wie bereits der C++ IDL-Compiler von Orbix, so hatte auch dieser Compiler die gleichen Schwierigkeiten mit der Bearbeitung der Makrodefinitionen. Deshalb wurde wieder der Standard-Preprocessor verwendet, um die Makroerweiterung vor der eigentlichen Übersetzung durchzuführen. Der Aufruf erfolgt im entsprechenden *Makefile* automatisch. Er sieht fast genauso aus wie der bereits auf Seite 42 beschriebene Aufruf. Zur zusätzlichen Unterscheidung zwischen C++ und Java-IDL-Compiler wurde das Symbol „`__ORBIX_JAVA__`“ definiert.

Nach diesem vorgeschobenen Schritt kann nun die Übersetzung gestartet werden. Auch hier wurden Argumente wieder in ähnlicher Weise wie für den C++-Fall gewählt. Die Argumente für die Ablage im Interface Repository sind völlig gleich. Darüber hinaus wurde noch zwei Java-spezifische Argumente hinzugefügt. Mit der Option „`-jQ`“ werden in den erzeugten Klassen Methoden namens *equals* erzeugt, die zwei Objekte auf inhaltliche Gleichheit – auch rekursiv in enthaltenen Strukturen – überprüft. Diese Methode ist notwendig, da der Gleichheitsoperator „`==`“ ausschließlich überprüft, ob es sich um dasselbe Objekt handelt, nicht aber, ob es sich um zwei Objekte gleichen Inhalts handelt. Gibt es zum Beispiel eine Kopie eines Objektes, so liefert „`==`“ Ungleichheit, weil es sich um zwei verschiedene Objekte handelt, während *equals* nach inhaltlicher Überprüfung die Gleichheit bestätigt.

Die zweite zusätzliche Option ist „`-jP MELODY`“. Dadurch werden alle erzeugte Klassen innerhalb eines neuen *Packages* mit dem Namen „*MELODY*“ abgelegt. Da eine große Zahl an Klassen erzeugt werden, ist diese Strukturierungshilfe äußerst sinnvoll. Damit sieht der vollständige Aufruf wie folgt aus:

```
/opt/Orbix_2.0/OrbixWeb1.0/bin/idl -jQ -jP MELODY
-d melody_if -R TRD_tcomm.idl
```

Der OrbixWeb Compiler erzeugt alle Java-Klassen standardmäßig im Unterverzeichnis *java_output* des aktuellen Verzeichnisses. All diese Klassen müssen jetzt vom Java-Compiler in Bytecode übersetzt werden. Um das *Makefile* nicht unnötig zu vergrößern und sich nicht von den erzeugten Klassennamen abhängig zu machen, wurde für ihre Übersetzung ein kleiner Trick verwendet, der den Beispielen von OrbixWeb entnommen ist: Statt alle Dateien einzeln aufzuzählen, werden mit Hilfe des Befehls des UNIX-Kommandos *find* alle „*.java*“ Dateien im Unterverzeichnis *java_output* gesucht und in einem einzigen Compileraufruf übersetzt.

Dies hat den großen Vorteil, daß die Übersetzung insgesamt schneller abläuft, weil alle für die Übersetzung benötigten Klassen nur ein einziges Mal vom Compiler geladen werden müssen.

Und die Abhängigkeiten zwischen den einzelnen CORBA-Klassen müssen nicht untersucht werden, was sonst notwendig ist, da bei der Übersetzung einer Klasse immer alle benötigten Klassen bereits übersetzt vorliegen müssen.

Dieses Vorgehen für die Übersetzung der CORBA-Klassen stellt allerdings auch keinen Nachteil dar, da generell gilt, daß der IDL-Compiler immer sämtliche Klassen neu erzeugt und damit auch sämtliche Klassen vom Java-Compiler neu übersetzt werden müssen. Das Vorgehen stellt also vielmehr eine Optimierung der Übersetzung dar. Die verwendete Anweisung ruft zunächst *find* auf, um alle Java-Dateien zu finden, speichert diese Namen dann in der Shell-Variablen *FILES* ab und ruft mit dieser Variablen den Java-Compiler auf.

```
FILES=`find java_output -name \*.java -print` ; \  
javac -g -d ../classes $FILES
```

Leider stellte sich heraus, daß die vom IDL-Compiler erzeugten Klassen einige Probleme bei ihrer Übersetzung machten. Der Java-Compiler meldete einige syntaktische Fehler! Alle Fehler betrafen dabei die auftretenden Unions. Der erste Fehler wurde in den Unions erzeugt, die zur Unterscheidung des *NULL*-Zeigerfalls eingeführt wurden. In der Schnittstellenbeschreibung wurde dort als Diskriminante eine *boolean* verwendet, was durchaus der CORBA-Spezifikation entspricht. Der IDL-Compiler erzeugte nun in den entsprechenden Klassen verschiedene *switch*-Anweisungen, die gerade über diese Diskriminante gehen. Unglücklicherweise erlaubt aber Java nicht, daß in einer *switch* Anweisung eine *boolean* verwendet wird. Dies führte zu Syntaxfehlern.

Da dies eindeutig ein Fehler des IDL-Compilers ist, wurde der erzeugte Programmtext mit Hilfe eines *sed*-Skriptes korrigiert, bis in einer späteren Version dies einmal behoben sein wird. Die Anwendung des Skriptes wurde sehr dadurch erleichtert, daß alle Klassen, die korrigiert werden müssen, aufgrund der Makrodefinitionen im IDL-File auf „*_union*“ enden und gleichzeitig dies die einzigen mit dieser Endung sind. Dadurch konnte die Suche der zu korrigierenden Datei wieder – in ähnlicher Weise wie gerade beschrieben – mit Hilfe eines *find* Aufrufes stattfinden. Die Anwendung eines Skriptes wurde weiterhin dadurch erleichtert, da die erzeugten Klassen vom Aufbau her fast identisch waren.

Das Skript findet sich im Verzeichnis des Java TUA unter dem Namen *bool.sed*. Eine genaue Erläuterung würde an dieser Stelle zu weit führen. Die Aufgabe ist durch die Fehlermeldungen des Java-Compilers klar umrissen. Alle *switch*-Anweisungen müssen durch entsprechende *if*-Anweisungen ersetzt werden. Dazu wurde der Aufbau der betroffenen Dateien untersucht. Um die vorgenommenen Änderungen des Skriptes nachzuvollziehen, sei es dem geneigten Leser empfohlen, sich die Ausgangsdatei sowie die veränderten Dateien näher anzuschauen. Die Ausgangsdatei ist dabei immer unter dem ursprünglichen Namen mit der zusätzlichen Endung „*.ORIG*“ zu finden. Die einzelnen Anweisungen des Skriptes sollen hier kurz für diejenigen erklärt werden, die sich nicht mit der Syntax von *sed* Skripten auskennen.

```
sed : 1. Löschen der Zeilen 18 bis 25  
      18,25d  
      2. Einfügen von „return d;“ statt der gelöschten Zeilen  
      26i\  
      return d;  
      3. Löschen der Zeilen 46 bis 53  
      46,53d
```

4. Einfügen von „return !d;“ stattdessen
54i\
return !d;
5. Globales Ersetzen von „switch“ in „switch (discriminator)“ durch „if“
s/switch (discriminator)/if (discriminator)/g
6. Das gleiche nochmals, falls nach dem „switch“ das Leerzeichen fehlt
s/switch(discriminator)/if (discriminator)/g
7. Globales Löschen aller „case true:“
s/case true://g
8. Globales Ersetzen aller „case false:“ durch „ } else { „
s/case false:/ } else {/g
9. Globales Entfernen aller „break;“ Anweisungen
s/break;//g
10. Löschen der Zeilen, bei denen in der ersten „default:“ steht und in der letzten „NO;“ zu finden ist. Dies betrifft nicht die bereits in 1. und 3. gelöschten Teile!
/default: /, /NO);/d

Damit sind aber noch nicht alle Fehler behoben! Ein weiteres Problem trat in den beiden von Melody definierten Unions auf. Auch dort meldete der Java Compiler Fehler. Es ist nicht ganz geklärt, ob es sich dabei um ein Problem des erzeugten Programmcodes handelt oder aber um einen Fehler des Java Compilers selbst. Das Problem leitet sich daraus ab, daß OrbixWeb sowohl ein Package als auch eine Klasse unter dem Namen des Interfaces erzeugt. Der Java-Compiler hatte nun Schwierigkeiten zu unterscheiden, ob die in der für die Union benutzten Konstanten in einer Klasse des Packages oder aber in der Interface-Klasse selbst zu finden war. Dieses Problem trat nicht auf, wenn die erzeugten Klassen nicht durch die „-jP“ Option in ein getrenntes Package gelegt wurden. Es tritt also erst bei mehrfach geschachtelten Packages auf.

Durch Einfügen einer expliziten *import* Anweisung und Referenzierung der Konstanten über ihren direkten Klassennamen gelang es, diesen Fehler zu beheben. Da in den beiden Unions zwei verschiedene Klassen verwendet werden, mußten auch zwei verschiedene Skripte entwickelt werden. Sie finden sich im Verzeichnis des Java TUA unter den Namen *nodetype.sed* und *valtempl.sed*. Die beiden Skripte enthalten nur zwei Anweisungen. Diese sollen hier nur kurz anhand einer der beiden erklärt werden. Zum Nachvollziehen sei wieder auf die entsprechenden Dateien verwiesen.

sed: 1. Einfügen der *import* Weisung nach Zeile 2
2a\
import MELODY.MELODY_trader.node_type_oo;
2. Entfernen der Packagebezeichnung vor der importierten Klasse ab Zeile 3 bis zum Ende der Datei
3, \$s/MELODY.MELODY_trader.node_type_oo/node_type_oo/

Nach dem durch die dargestellten Skripte der generierte Java-Code korrigiert wurde, kann er jetzt erfolgreich übersetzt werden. Um die übersetzten Klassen auch nutzen zu können, muß darauf geachtet werden, die UNIX-Environmentvariable *CLASSPATH* entsprechend so zu setzen, daß sie vom Java Compiler und Interpreter auch gefunden werden.

Der JAVA Trader User Agent

Mit der erfolgreichen Übersetzung der Schnittstellenbeschreibung ist es schon möglich, Melody zu verwenden. Doch um Klienten eine dem C++-TUA ähnliche Zugriffsmöglichkeit anzubieten, wurde eine weitere Klasse implementiert, die diese Umsetzung vornimmt. Ihr Aufbau ist im Prinzip gleich wie in C++, wobei in der vorliegenden Realisierung keine expliziten Konvertierungen stattfinden. Allerdings existieren einige Klassen, die den Zugriff auf die Parameterobjekte in einer ähnlichen Weise erlauben, wie er bereits von Melody bekannt ist. Diese Klassen werden allerdings erst später im Kapitel „Die Java Zugriffsklassen“ auf Seite 82 nach dieser Beschreibung des Java TUA erläutert.

Der prinzipielle Aufbau der Methoden des Java TUAs sei am schon bekannten Beispiel der Methode *insertType* erklärt. Sie hat die Aufgabe, einen neuen Dienstyp in den Dienstypraum von Melody einzufügen. Sie bekommt als IN-Parameter die Beschreibung des neuen Dienstyps. Melody gibt als Ergebnis des Aufrufes eine Statusmeldung zurück. Bei Fehlern kommen im zweiten Parameter der Methode Fehlermeldungen zurück. Die Methode selbst ist sehr einfach:

```
Java: public int insertType(STInsert_struct_oo newType,
                           ErrorMessage_struct_oo returnMes)
        throws SystemException
    {
        if (!connected)
            prepareConnection();

        return trader.INSERTTYPE(newType, returnMes);
    }
```

Alles was die Methode macht, ist die Parameter entgegenzunehmen, dann ebenso wie der C++-TUA auch, (vgl. Seite 68) die Verbindung aufzubauen, falls sie noch nicht besteht, den Aufruf abzusetzen und das Ergebnis entsprechend zurückzugeben. Wie bereits erklärt, führt der Java TUA im Gegensatz zum C++-TUA keine expliziten Konvertierungen durch. Bei Verwendung der später in diesem Kapitel erläuterten Zugriffsklassen erfolgt diese Umsetzung implizit beim Aufruf durch die Vererbungshierarchie der Klassen, ähnlich wie dies in C++ bei der Realisierung der DCE-Schnittstelle ausgenutzt wurde.

Zweiter deutlicher Unterschied zwischen C++ und Java Methode ist die Handhabung der *Exceptions*. Anders wie in C++ werden die Exceptions von CORBA nicht abgefangen und durch eine eigene Fehlermeldung ersetzt. Statt dessen werden sie an die aufrufende Methode weitergegeben. Dies erscheint sinnvoller, da das Exception Handling in Java bereits seit Anfang an fest integriert ist und zu den Grundlagen von Java gehört. Das C++-Exception Handling ist dagegen erst später hinzugekommen. Es wird in Melody auch nirgends verwendet. Deshalb wurden die Exceptions, die von CORBA unter C++ auftreten können, abgefangen und in Fehlermeldungen umgewandelt.

Die Methode *prepareConnection* hat wie unter C++ auch die Aufgabe, die Verbindung mit dem Server über den ORB herzustellen. Wenn sie bereits besteht, wird nichts weiter unternommen. Ansonsten wird durch Aufruf der Methode *_bind* in der vom IDL-Compiler erzeugten Klasse *MELODY_trader* ein passender Server gesucht. War dieser Schritt erfolgreich, so wird die

zurückgegebene Objektreferenz im Attribut *trader* des Java TUA abgespeichert. Alle weiteren Zugriffe auf den Server erfolgen dann über diese Referenz.

Zusätzlich wird in dieser Methode noch die Ausgabe von Diagnose-Ausgaben von Orbix unterdrückt. Dies geschieht durch Aufruf der Methode *setDiagnostics*. Der Parameter 0 bedeutet das keine Meldungen gemacht werden soll. Alternativ kann hier auch 1 übergeben werden, um einfache Meldungen einzuschalten. Bei Übergabe von 2 werden ausführliche Meldungen erzeugt. Da sie aber ausschließlich für den Programmierer interessant sind, wurden sie ausgeschaltet.

```
Java: private void prepareConnection()
        throws SystemException
    {
        if (connected)
            return;

        _CORBA.Orbix.setDiagnostics(0);

        trader = MELODY_trader._bind(ENTRY_NAME);
        connected = true;
        return;
    }
```

Die Java Zugriffsklassen

Der Java TUA ermöglicht es, daß Aufrufe an Melody in gleicher Weise wie unter C++ auch geschehen können. Die Parameter für diese Aufrufe müssen allerdings „von Hand“ gefüllt werden. Dazu muß der Programmierer des Klienten wissen, welche Bedeutung die einzelnen Felder der Datenstrukturen haben und sie korrekt ausfüllen. Unter C++ gibt es an dieser Stelle verschiedene Zugriffsmethoden, die von den Parameterobjekten angeboten werden und auf diese Weise wesentlich einfacher verwendet werden können. Aus diesem Grund wurden diese Zugriffsmethoden unter Java nachgebildet. Sie erhalten alle, so weit dies möglich ist, den gleichen Namen und die gleichen Parameter wie unter C++ auch. Auch intern erfüllen die Methoden exakt die gleichen Aufgaben wie die analogen C++-Methoden auch.

Diese Zugriffsklassen sind wie die ursprünglichen C++-Klassen benannt worden. Ihre Namen leiten sich damit von den CORBA-Datenstrukturen durch Streichen der Endung „_struc_oo“ ab. Jede dieser Klassen hat als Basis-Klasse die entsprechende CORBA-Struktur. Die Methoden operieren direkt auf diesen Strukturen und füllen sie korrekt aus, so daß sie jederzeit für die Übertragung genutzt werden können. Da jedes Objekt einer abgeleiteten Klasse gleichzeitig auch ein Objekt seiner Basis-Klasse ist, können sie problemlos bei Aufrufen an den Java TUA verwendet werden. Wie vielleicht deutlich wird, handelt es sich dabei um die gleichen Mechanismen, die unter C++ für die DCE-Schnittstelle verwendet wurden.

Durch Verwendung dieser Zugriffsklassen ist eine Portierung von Melody-Anfragen von C++ nach Java sehr einfach: Es muß nur die Syntax der Aufrufe entsprechend umgesetzt werden. Da C++ und Java sowieso sehr ähnlich sind, ist diese Umsetzung sehr einfach. Allerdings müssen noch zwei Einschränkungen gemacht werden: In der vorliegenden Version des Java Interfaces

werden keine Management-Attribute von Melody unterstützt. Solange das Management-System nicht unter Java verwendbar ist, können diese Attribute nicht benutzt werden. Deshalb wurden die entsprechenden Klassen nicht implementiert.

Die zweite Einschränkung ist, daß die Ergebnisobjekte von Melody nicht nachgebildet wurden. Die Ergebnisse von Anfragen werden nicht wieder in solche Objekte zurückkonvertiert, wie dies unter C++ geschieht. Der Klient muß stattdessen direkt auf den CORBA-Datenstrukturen arbeiten und die Ergebnisse von dort auslesen. Dies scheint gerechtfertigt, da auch diese Objekte unter C++ nur sehr wenige Zugriffsmethoden anbieten und ihre Attribute meist direkt gelesen werden. Die Nachbildung dieser Objekte würde also hauptsächlich zusätzlichen Wartungsaufwand bedeuten, der zur Zeit nicht sinnvoll scheint.

Die Portierung der Zugriffsklassen

Die Realisierung der Zugriffsmethoden auf die Parameterobjekte unter Java ist gleichbedeutend mit der Portierung der Klassen von C++ nach Java: Es müssen die Eigenschaften der Objekte in gleicher Weise unter Java nachgebildet werden. Erste wichtige Aspekte dieser Nachbildung sollen hier erst einmal am Beispiel des Konstruktors von *STInsert* erläutert werden.

```
C++: class STInsert : public STInsert_struct {
    ...
    STInsert(const char ServiceType[STLENGTH],
            const char *Interface=NULL)
    {
        strcpy( (char*)name, ServiceType);
        if(Interface != NULL) {
            sTInterface = new char[strlen(Interface)+1];
            strcpy( (char *)sTInterface,Interface);
        }
        else
            sTInterface = NULL;
        sTDescriptionList = NULL;
        sTSupertypeNames = NULL;
        sTRuleList = NULL;
        sTTakeList = NULL;
    };
    ...
Java: class STInsert extends STInsert_struct_oo {
    ...
    public STInsert(String ServiceType, String Interface)
        throws BAD_PARAM
    {
        name = ServiceType;
        sTInterface = new sTInterface_union();
        if (Interface == null) {
            sTInterface.dummy(false, false);
        } else {
            sTInterface.sTInterface(Interface, true);
        }
    }
}
```

```
    STDescriptionList = new DescriptionNode();
    STSupertypeNames = new STTypeNamesNode();
    STRuleList = new SRuleNodeT();
    STTakeList = new takeEl();
}
...
}
```

Die Umsetzung der Parameter erfolgt in direkter Weise. Die Strings von C++ werden auf Strings unter Java abgebildet. Dabei fällt die Unterscheidung von Strings mit fester maximaler Länge weg. Sie kommt bei der besonderen Behandlung von NULL-Strings wieder herein. Bei den Parameterdeklarationen ist zu beachten, daß Java keine Default-Argumente kennt, wie dies hier für *Interface* ausgenutzt wird. Sollen Default-Argumente nachgebildet werden, so müssen Methoden mit sämtlichen möglichen Aufrufsignaturen implementiert werden und die Default-Werte dort gesetzt werden. An dieser Stelle kann also unter Java ein Konstruktor hinzugefügt werden, der nur den Dienstyp als Argument erwartet und dann den oben beschriebenen Konstruktor mit *null* als zweites Argument aufruft:

```
Java: public STInsert(String ServiceType, String Interface)
        throws BAD_PARAM
    {
        this(ServiceType, null);
    }
```

Bei Zuweisungen unter Java sollte immer daran gedacht werden, daß abgesehen von den Basistypen, immer nur Objektreferenzen kopiert werden. Es ist also nicht möglich, eine Kopie von einem Objekt anzulegen, indem es einfach anderen Variable zugewiesen wird! Deshalb müssen normalerweise neue Objekte angelegt werden und entsprechend initialisiert werden. Bei *String* kann allerdings eine besondere Eigenschaft ausgenutzt werden. Objekte von diesem Typ sind konstant, das heißt auch, wenn bei einer Zuweisung nur die Objektreferenz und nicht das Objekt selbst kopiert wird, kann der Text selbst nicht verändert werden! Deshalb muß kein neuer String angelegt werden, wenn er kopiert werden soll. Die Zuweisung von *ServiceType* an *name* reicht also aus.

Für die Strings mit beliebiger Länge wurden in CORBA Unions eingeführt, um auch den Fall berücksichtigen, daß ein NULL-Zeiger übergeben werden soll. Im Konstruktor von *STInsert* muß deshalb diese Union angelegt werden, da sonst an dieser Stelle nur ein *null*-Objekt steht. Unter C++ muß dies nicht geschehen, da die Union direkt in die Klasse eingebunden ist, und somit mit Anlage des gesamten Objektes bereits existiert. Nachdem die neue Union angelegt wurde, wird sie korrekt initialisiert. Wenn *null* übergeben wurde, wird die *dummy*-Diskriminante aktiviert, ansonsten wird der String gesetzt.

Bei den letzten vier Zuweisungen handelt es sich um Initialisierungen von Listen. Unter C++ sind sie als verkettete Listen repräsentiert. Da die Listen leer angelegt werden sollen, genügt die Zuweisung von *NULL*. Doch die Listen wurden in CORBA als Arrays realisiert. Um die Liste zu initialisieren, wird ein Konstruktor definiert, der eine leere Liste erzeugt. Diese Konstrukturen werden von *STInsert* verwendet.

Soweit zu diesem Beispiel. Auf den folgenden Seiten werden jetzt alle Abbildungen, die für die Portierung verwendet wurden, genauer erläutert.

Die Verwendung von Enums

Aufzählungstypen aus der Schnittstellenbeschreibung werden vom IDL-Compiler in Klassen gleichen Namens übersetzt, die alle definierten Konstanten beinhalten. Um auf sie zugreifen zu können, ist es deshalb notwendig, den Klassennamen zusätzlich mit anzugeben. Da die Klassen im Package *MELODY.MELODY_trader* definiert werden, muß dem Namen immer noch der Packagename vorangestellt werden, so lange nicht die Klasse explizit importiert wird. Um zum Beispiel *alo_pure* aus der Enum *ALOType* zu verwenden, wurde folgende Abbildung gewählt:

```
C++: ...
      setType(alo_pure);
      ...

Java: ...
      import MELODY.MELODY_trader.ALOType_oo;
      ...
      setType(ALOType_oo.alo_pure);
      ...
```

Die Konstanten selbst sind in den Klassen als normale Ganzzahlen vom Typ *int* definiert. Sollen sie übergeben oder zurückgeben werden, so müssen die entsprechenden Methoden dies berücksichtigen.

Die Verwendung von Unions

Unions werden bereits in der Schnittstellenbeschreibung etwas anders angelegt wie in C++ (Siehe „Die Unions“ auf Seite 39.). In C++ wurden die Unions innerhalb einer zusätzlichen Struktur angelegt, die unter CORBA nicht existiert. Außerdem muß für den CORBA-Strukturen die Diskriminante nicht explizit gesetzt werden. Sie wird implizit durch Setzen des Inhalts entsprechend angepaßt. Andererseits muß beim Abfragen der Union immer darauf geachtet werden, daß nur die gesetzte Variante gelesen werden darf. Wird dies nicht getan, wird von CORBA eine Exception erzeugt!

```
CORBA:// CORBA Schnittstellenbeschreibung
      union u switch (Value_Template_oo) {
          case YES:      ALT_struct_oo alt;
          case NO:       ALO_struct_oo alo;
      } value_el;

C++: // Definition der Union für C++
      struct {
          Value_Template value_template;
          union {
              ALT_struct alt;
              ALO_struct alo;
          } u;
      } value_el;
```

```
C++: // Zugriff auf die Union unter C++
      ALOType getType()
      {
        return value_el.u.alo.type;
      };

      void setType(ALOType type) {
        value_el.value_template = NO;
        value_el.u.alo.type = type;
      };

Java: // Zugriff auf die Union unter Java
      public int getType()
          throws BAD_OPERATION
      {
        return value_el.alo().type;
      }

      public void setType(int new_type)
      {
        try {
          value_union val = new value_union();
          val.dummy(false, false);

          value_el.alo(new ALO_struct_oo(new_type, val),
                      Value_Template_oo.NO);
        } catch (BAD_PARAM ex) {
          System.out.println("exception BAD_PARAM: "+
                             ex.getMessage());
        }
      }
    }
```

Die Methode *getType* wird direkt, ohne weitere Änderungen in Java umgesetzt. Es erfolgt dabei ebenfalls keine Abfrage auf die Diskriminante. Sollte diese falsch gesetzt sein, so meldet die Zugriffsmethode der Union unter Java eine *BAD_OPERATION* Exception. Der Inhalt der Union wird immer über eine Methode abgefragt, die den Namen der Variante in der Schnittstellenbeschreibung – in diesem Fall also *alo()*, hat.

Die Methode *setType* setzt unter C++ die Diskriminante auf den Fall *NO* und schreibt darauf hin in das *type* Feld der Variante *alo* einen neuen Wert. Unter Java erfolgt das Setzen der Diskriminante implizit durch Einfügen eines neuen Inhalts in die Union. Gesetzt wird die Union durch Verwendung einer Methode, die wiederum den Namen ihrer Variante trägt, nun aber ein Objekt des entsprechenden Typs sowie den Wert der neuen Diskriminante erwartet. Dieses Objekt muß dabei mit den vorangestellten Anweisungen erst erzeugt werden, bevor dann die Zuweisung geschehen kann. Die dabei möglicherweise entstehende *BAD_PARAM* Exception wird abgefangen, da sie nicht auftreten kann, weil die Union ja korrekt gesetzt wird.

```
Java: if (value_el.alo().value.is_value())
      return new ALOModify(value_el.alo().value.value());
      else
```

```
return new ALOModify(null);
```

Die Abfrage der Diskriminante kann auf zwei verschiedene Weisen geschehen. Die erste Möglichkeit ist oben dargestellte. Zu jeder Variante der Union existiert eine Methode, mit der überprüft werden kann, ob sie gesetzt ist. Der Name der Methode leitet sich vom Namen der Variante durch Voranstellen von „*is_*“ ab. Sie liefert *true*, falls die Variante gesetzt ist. Alternativ kann die Diskriminante direkt durch Aufruf der Methode *discriminator* abgefragt werden.

Da die *VARSTRINGs* und die *OPTSTRUCPs* in der Schnittstellenbeschreibung für CORBA ebenfalls durch Unions ersetzt werden, lassen sich diese Anmerkungen in gleicher Weise auf sie übertragen. Wenn zum Beispiel ein *null*-String übergeben wird, muß die korrekte Variante gesetzt werden, wie dies im folgenden Beispiel geschehen ist:

```
Java: if (newval == null)
    value.dummy(false, false);
    else
    value.value(newval, true);
```

Rückgabe von Strings in Referenzparametern

Java übergibt immer Werteparameter und kennt keine Referenzparameter. Damit ist es nicht direkt möglich, übergebene Parameter von einer Methode ändern zu lassen. Statt dessen ist es notwendig, für diesen Parameter ein Objekt zu definieren, das verändert werden kann. In Melody werden an einige Stellen Methoden angeboten, die einen String über einen Parameter zurückgeben. Um dies nachbilden zu können, muß in Java ein Objekt der Klasse *StringBuffer* übergeben werden, das dann innerhalb der Methode verändert werden kann. Dort kann problemlos der Text gesetzt werden.

```
Java: public void getValue(StringBuffer val)
    {
        val.setLength(0);
        val.append("new value");
    }
```

Für die Beschreibung und die genaue Verwendung von *StringBuffer* sei hier auf die Dokumentation der Java API [3] verwiesen.

Die Umsetzung von String-Vergleichen

Wie bereits früher erwähnt, verwendet Java in allen Variablen nur Objektreferenzen, es sei denn, es handelt sich um einfache Basis-Typen wie *char* oder *int*. Dies muß bei String-Vergleichen berücksichtigt werden! Ein einfacher, direkter Vergleich zweier Strings mit „*==*“ erfüllt diese Aufgabe nicht richtig. Dieser Vergleich überprüft nur, ob es sich in beiden Fällen um das gleiche *String*-Objekt handelt. Es kann aber sein, daß es zwei verschiedene *String*-Objekte gibt, die den gleichen Inhalt haben. Um zwei Strings auf inhaltliche Gleichheit zu überprüfen, existiert in der *String*-Klasse eine Methode mit dem Namen *equals*, die verwendet werden kann.

```
Java: // Überprüfe auf Leerstring
    if (altp.value.equals(""))
        setValue((String) null);
```

```
else
    setValue(altp.value);
```

Abbildung von Listen

Listen werden in den CORBA-Datenstrukturen, anders als in C++, als Arrays angelegt. Die Zugriffsmethoden mußten deshalb abweichend von der Vorlage in C++ realisiert werden. Zunächst einmal ist wichtig, zu beachten, daß eine leere Liste in C++ völlig anders dargestellt wird als in den CORBA-Strukturen. In C++ genügt ein einfacher NULL-Zeiger. Unter CORBA muß die Liste angelegt sein, jedoch mit Länge 0. Dies war bereits im Einführungsbeispiel zu den Zugriffsklassen zu sehen. Der entsprechende Java-Konstruktor initialisiert die Liste direkt als leer.

```
Java: // Constructor
public DescriptionNode()
{
    // Initialize an empty list
    super(0);
}
```

Um an diese Liste nun ein neues Element anzuhängen, muß ein anderer Weg beschritten werden, als dies in C++ getan wird. In C++ wird durch einen zweiten Konstruktor ein neues Listenelement angelegt und angehängt. Dies kann dort so geschehen, weil die Liste durch eine verkettete Liste dargestellt wird. Jeder Knoten der Liste ist damit gleichzeitig Element der Liste und Anker für eine ganze Liste! Die CORBA-Datenstrukturen erfordern hier aber eine deutliche Trennung. Aus diesem Grund wurde dieser zweite C++-Konstruktor ersetzt durch eine neue Methode *add*, die abgesehen vom Zeiger auf den nächsten Knoten die selben Parameter erhält.

Gleichzeitig wurden für die Listenelemente neue Zugriffsmethoden definiert, die in dieser Weise kein Gegenstück in C++ habe. Die Funktionalität findet sich dort bereits in den Listen-Objekten. Der in Java verfolgte Ansatz entspricht wieder dem Gedanken, die Liste als Ganzes von den Elementen an sich zu trennen.

```
C++: SRuleNodeT(SRule *newrule,
               OpType op,
               SRuleNodeT *node = NULL) {
    elem.Op = op;
    elem.rule = newrule;
    next_el = node;
};

Java: // Add a element to the list
public void add(SRule newrule, int op)
    throws BAD_PARAM
{
    int i;

    if (length < 0)
        i = 1;
```

```
else
    i = length + 1;

ensureCapacity(i);

buffer[i - 1] = new SRuleNodeT_el(op, newrule);
length = i;
}
```

Unter Java hat eine Liste zwei Attribute: *buffer* ist das Feld, in dem die einzelnen Elemente abgelegt werden. *length* ist die Zahl der in dem Feld gespeicherten Elemente. *length* wird bei Anlage der Liste auf (-1) initialisiert. Die Methode *add* muß zunächst in einem ersten Schritt sicherstellen, daß das Feld genügend Platz aufweist. Dazu wird die Methode *ensureCapacity* verwendet, die als Parameter die neue Größe erwartet. Nach dem Aufruf kann das neue Element im Feld angelegt werden. Es wird durch Aufruf eines neu geschaffenen Konstruktors der Elementklasse angelegt. Man beachte dabei, daß das Feld in der CORBA-Struktur als Feld von Element der Klasse *SRuleNodeT_el_struct_oo* angelegt wurde. Tatsächlich wird aber hier, wie an allen anderen Stellen der Zugriffsklassen auch, die abgeleitete Klasse benutzt. Dies muß bei der Verwendung der Objekte beachtet werden. Soll eine Methode der Zugriffsklasse verwendet werden, so muß zuvor ein expliziter Cast erfolgen.

Der Zugriff auf ein einzelnes Element geschieht über die normale Array-Adressierung. Folgendes Beispiel sucht in der Liste *sTDescriptionList* nach dem Element mit dem Namen *myName* und ruft von diesem Element die Methode *set_ALT* auf. Zur Verwendung der Casts sei auf den folgenden Abschnitt verwiesen.

```
C++: found = 0;
sDNP1 = (STDescriptionNodeT*)sTDescriptionList;
while(sDNP1 != NULL) {
    if(strcmp(myname, sDNP1->get_name()) == 0) {
        found = 1;
        sDNP1->set_ALT (newalt, op);
    }
    sDNP1 = sDNP1->next();
}
```

Java: `boolean found = false;`

```
for (int i = 0; i < sTDescriptionList.length; i++) {
    if (myName.equals(((DescriptionNode_el)
        (sTDescriptionList.buffer[i])).get_name())) {
        found = true;
        ((DescriptionNode_el)
        (sTDescriptionList.buffer[i])).set_ALT(newalt, op);
    }
}
```

Verwendung von Zugriffsmethoden innerhalb von Zugriffsmethoden

Möchte eine Zugriffsmethode auf eine Zugriffsmethode eines anderen Objektes zugreifen, so gibt es zunächst einmal ein Problem: In den CORBA-Datenstrukturen sind natürlich die Zugriffsklassen nicht bekannt. Alle Felder enthalten nur Verweise auf andere Strukturen von CORBA. Durch die Zugriffsmethoden wurden aber an dieser Stelle tatsächlich Objekte der abgeleiteten Zugriffsklassen gespeichert.

Um hier nun auf die Methoden zugreifen zu können, muß ein expliziter Cast auf die wirklich verwendete Klasse benutzt werden. Das Java Laufzeitsystem überprüft später die Zulässigkeit des Casts. Stellt sich heraus, daß es sich überhaupt nicht um ein Objekt der angegebenen Klasse handelt, wird eine Exception erzeugt. Solange zur Erzeugung der Parameterobjekte ausschließlich die Zugriffsklassen verwendet werden, stellt dies kein Problem dar. Werden allerdings Teile der Datenstrukturen von Hand gesetzt und mit anderen Aufrufen gemischt, so können Inkonsistenzen auftreten, die eben gerade dazu führen. Dies sollte immer beachtet werden!

```
Java: public void set_ALT(ALT newalt, int op)
        throws BAD_OPERATION, BAD_PARAM
    {
        ((STAttributeT) property.stt()).setALT(newalt, op);
    }
```

Beispielumsetzung für das Java Interface

Zum Abschluß dieses Kapitels soll noch an einem Beispiel die Umsetzung einer kurzen Anwendung gezeigt werden. Wie gut zu sehen ist, wird durch die Einführung der Zugriffsklassen erreicht, daß der Aufbau der Parameterobjekte, in diesem Fall von STInsert, in paralleler Weise erfolgt. Eine Umsetzung von C++ nach Java beschränkt sich fast ausschließlich auf die Anpassung an die Syntax von Java.

Im Beispiel wird ein neuer Dienstyp mit dem Namen „*SecondService*“ eingefügt. Er ist ein Subtyp vom Typ „*MELODY*“. Daraufhin werden dem Dienstyp vier Regeln hinzugefügt. *mod_rule* fügt eine Regeln hinzu, wenn sie noch nicht vorhanden ist. Sonst wird eine bestehende Regel durch die neue ersetzt. Schließlich wird noch ein neues Attribut mit dem Namen „*CDS*“ hinzugefügt. Danach wird dafür ein Default-Wert gesetzt.

```
C++: TUAgent_oo *tUAgent = new TUAgent_oo();

STInsert *sti
    = new STInsert("SecondService",
                  "This is my second service");
sti->add_supertype("MELODY");
sti->mod_rule("Speed",
            "~min[QueueLength+RunningJobs+HostLoad]");
sti->mod_rule("Cost", "~min[Cost]");
sti->mod_rule("DEFAULT", "~min[Cost]");
sti->mod_rule("working", "AdmState == 1 && OpState != 2");
```

```
sti->add_attribute("CDS", Trader_String, NULL,
                  Optional, staAtt, qNo);
ALTModify *altm = new ALTModify(NULL);
sti->rep_alt("CDS", altm);

ErrorMessage *errmes = NULL;
int ret_value = tUAgent->insertType(sti, errmes);
```

Für Java sind nur wenige Änderungen nötig. Der Name der TUAgent-Klasse lautet anders. Es existieren keine Zeiger unter Java. In den Deklarationen werden sie einfach weggelassen. Die Dereferenzierung erfolgt nur noch über den Punkt (.). Schließlich müssen für die Symbole der Aufzählungstypen noch die Klassen angegeben werden.

```
Java: TUAgent tUAgent = new TUAgent();
```

```
STInsert sti
    = new STInsert("SecondService",
                  "This is my second service");
sti.add_supertype("MELODY");
sti.mod_rule("Speed",
             "~min[QueueLength+RunningJobs+HostLoad]");
sti.mod_rule("Cost", "~min[Cost]");
sti.mod_rule("DEFAULT", "~min[Cost]");
sti.mod_rule("working", "AdmState == 1 && OpState != 2");

sti.add_attribute("CDS", AType_oo.Trader_String, null,
                  AForce_oo.Optional, Aclass_oo.staAtt,
                  QualityGuarantee_oo.qNo);
ALTModify altm = new ALTModify(null);
sti.rep_alt("CDS", altm);

ErrorMessage_struct_oo errmes;
int ret_value = tUAgent.insertType(sti, errmes);
```

Ein Teil der Aufgabe dieser Studienarbeit war es, Melody im Agentensystem Mole verfügbar zu machen. Es sollte für Agenten ermöglicht werden, daß sie Anfragen an Melody stellen und sich Dienste vermitteln lassen. Im Prinzip ist dieses Ziel mit der Fertigstellung des Java Interfaces des letzten Kapitels bereits erreicht. Agenten, die ja selbst in Java geschrieben sind, können über den Java TUA und die CORBA-Schnittstelle auf Melody zugreifen. Dazu muß aber ein Agent eine Voraussetzung erfüllen: Er muß sich an einem Ort befinden, an dem OrbixWeb und der Java TUA vorhanden sind. Die Klassen der beiden Teile müssen vom Java Interpreter zu finden sein, sonst ist der Versuch zum Scheitern verurteilt und das Laufzeitsystem meldet eine Exception.

Agenten in Mole sind mobil, können also ihren Aufenthaltsort verändern, und so auch an Orte kommen, die eben diese Voraussetzung nicht erfüllen. Wenn sich ein Agent genau an einem solchen Ort befindet und doch einen Auftrag für Melody hat, so bleibt ihm nichts anderes übrig, als von neuem zu migrieren. Bei der Kommunikation zwischen Agenten besteht dieses Problem nicht, weil sie über Nachrichten geschieht, die auch zwischen verschiedenen Orten verschickt werden können. Aber solange es keinen Agenten für Melody gibt, ist dies nicht möglich.

Allerdings muß zu diesem Punkt gesagt werden, daß er für den hier vorgestellten Ansatz nur von geringerer Bedeutung ist. Für den Nachrichtenaustausch werden nämlich die CORBA-Datenstrukturen selber verwendet. Der Klient stellt also in gewohnter Weise die Daten für eine Nachricht zusammen, versendet sie und bekommt später die Antwort zurück. Damit müssen sich alle benötigten Klassen auch am „entfernten“ Sendeort befinden oder der Agent bringt sie bei der Migration mit. Da sie vom Umfang her sehr groß sind, ist fragwürdig, inwiefern das sinnvoll ist. Es ist noch zu untersuchen, wie relevant dieser Aspekt für die Anwendung ist und ob nicht für den Datenaustausch andere Nachrichten verwendet werden sollten.

Ein Melody Agent hat aber noch einen weiteren Vorteil für die Bearbeitung. Da Klienten und der Melody Agent immer in zwei verschiedenen Threads laufen, erfolgt sie in jedem Fall asynchron. Der Klient kann also anderen Arbeiten nachgehen (und auch weitere Aufträge schick-

ken), während Melody die Anfrage bearbeitet. Würde der Zugriff dagegen direkt über den Java TUA erfolgen, so wäre sie synchron – der Agent wartet auf die Rückkehr des TUA-Aufrufs –, solange der Agent nicht durch eigene Threads die Asynchronität selbst implementiert.

Die in diesem Kapitel vorgeschlagene Implementation eines Melody Agenten stellt eine noch recht einfache Lösung dar, um Agenten über Nachrichtenaustausch Zugriff auf Melody zu verschaffen. In dieser Lösung werden grundsätzliche Fragen über die Einbindung eines Trading-Dienstes nicht berücksichtigt. Sie soll nur die generelle Machbarkeit der Aufgabe darlegen. Weiterführende Arbeiten müssen untersuchen, wie Melody in Mole am besten angewendet werden kann. Dazu gehören auch Fragen, wie Agenten ihre Dienste anbieten können.

Der Melody Agent

Bevor auf die für den Austausch mit dem Melody Agenten bestimmten Nachrichten eingegangen wird, einige Erläuterungen zur Implementation des Agenten selbst. Wie bereits erklärt, akzeptiert und versendet der Agent als Nachrichteninhalte die CORBA-Datenstrukturen. Seine Aufgabe stellt sich dabei als reine Schnittstellenfunktion dar. Ohne besondere Behandlung werden die empfangenen Nachrichten in Aufrufe des Java TUA abgebildet und die Ergebnisse des TUAs werden auf gleiche Weise zurückgeschickt. Darüber hinaus muß er noch eventuell notwendige Initialisierungen zu Beginn erledigen. In diesem Abschnitt sollen die drei wesentlichen Teile des Agenten erläutert werden: die Methoden *start*, *stop* und *receiveMessage*.

Die *start* Methode eines Agenten wird vom Agentensystem aufgerufen, wenn er seine Arbeit beginnen kann. Sie muß damit den Agenten so weit vorbereiten, daß er auf danach ankommende Nachrichten korrekt reagieren kann. Zunächst einmal benötigt der Agent natürlich den Java TUA, der in einem ersten Schritt erzeugt wird. Dazu wird einfach ein neues Objekt der Klasse *TUAgent* erzeugt und im Agenten abgespeichert, da dieses Objekt die Schnittstelle zu Melody darstellt. Im nächsten Schritt wird überprüft, ob der erzeugte TUA eine Verbindung zum Melody Server aufbauen kann. Da der TUA in jeder seiner Methoden automatisch die Verbindung herstellt, wenn sie noch nicht besteht, genügt also ein normaler Aufruf des TUA aus. Der vorliegende Melody Agent ruft dazu die Methode *getHost* auf, über die der Server aufgefordert wird, den Namen seines Rechners zurückzusenden. Da diese Methode keine direkte Traderfunktionalität benutzt, ist sie besonders dafür geeignet, den Verbindungsaufbau zu überprüfen.

```
Java: public void start() {
    String hostName;

    // Erstelle einen neue TUA.
    tUAgent = new TUAgent();

    try {
        hostName = tUAgent.getHost();
    } catch (SystemException tae) {
        System.out.println("exception: " + tae.toString());
        return;
    }
};
```

```
...  
}
```

Sollte der Verbindungsaufbau nicht gelingen, so wird von der Methode *getHost* eine Exception erzeugt, die der Agent abfängt. In diesem Fall wird eine Fehlermeldung ausgegeben und der ganze Vorgang abgebrochen, da der Agent seine Arbeit ohne Verbindung zu Melody nicht aufnehmen kann. War der Aufruf dagegen erfolgreich, ist er bereit für ankommende Nachrichten. Doch zuvor wird er noch im Dienstverzeichnis des Ortes, wo er sich befindet, registriert. Ein Ort in Mole bietet den Agenten an, ihren Dienst unter einem Namen registrieren zu lassen. Dieses Verzeichnis ist natürlich sehr einfach, bietet aber eine zusätzliche Abbildung von einem allgemeinen Dienstnamen auf einen bestimmten Agenten. Da es sich beim Melody Agenten um einen *System Agenten* handelt, der im Gegensatz zu den *Benutzer Agenten*, nicht migrieren kann, ist die Registrierung durchaus gerechtfertigt.

```
Java: static final String servicename = "MELODY";  
...  
void start{  
...  
    actuallocation.registerMeForService(myname,  
    servicename);  
}
```

Wenn ein Agent sich beim Start registriert, so muß er dies natürlich am Ende wieder rückgängig machen. Das geschieht durch Aufruf der Methode *unregisterMeForService* des Ortes, an dem er sich befindet. Dieser Aufruf wurde deshalb in die *stop* Methode des Melody Agenten eingefügt. Weitere Aufgaben dieser Methode wären eigentlich noch der Verbindungsabbau sowie die Freigabe des erzeugten TUAs. Doch beides ist nicht möglich. Für den Verbindungsabbau gibt es leider keine Methoden in CORBA. Und genauso wenig ist es unter Java möglich, irgendwelche Objekte zu löschen. Nicht mehr benötigte Objekte werden vom Java *Garbage Collector* automatisch entfernt. Damit bleibt für *stop* nur der eine Aufruf übrig:

```
Java: public void stop() {  
    actuallocation.unregisterMeForService(myname,  
    servicename);  
}
```

Damit zum letzten und auch wichtigsten Teil des Melody Agenten, dem Nachrichtenempfang. Der Agent empfängt von Klienten Nachrichten, in deren Inhalt die notwendigen Parameter für die TUA-Aufrufe übergeben werden. Sie werden einfach weitergegeben. Die empfangenen Nachrichten müssen dazu selbstverständlich sämtliche Informationen enthalten, um eine eindeutige Abbildung auf einen Aufruf zu erlauben. Vereinfachend kommt hinzu, daß in den Nachrichten an den Melody Agenten nur die IN-Parameter der Methode enthalten sein müssen. Die restlichen OUT-Parameter werden in die Rückantwort zurückgesandt.

Es sind verschiedene Ansätze denkbar. Der zunächst einfachste Weg erscheint die Definition von neuen Nachrichtenklassen für jeden Aufruf zu sein. Jede Nachricht könnte ganz individuell auf die notwendigen Parameter eingestellt werden. Die unterschiedlichen Nachrichtentypen könnten durch Verwendung des Operators *instanceof* überprüft werden. Dieser Operator überprüft, ob ein bestimmtes Objekt von einer bestimmten Klasse ist. Er liefert dabei auch *true*,

wenn es sich bei der Klasse des Objekts um eine abgeleitete Klasse der angegebenen handelt. Das folgende Beispiel soll dies demonstrieren.

```
Java: Message m;
      if (m instanceof MELODYMessage) {
          // m gehört zu MELODYMessage oder einer abgeleiteten
          Klasse
          MELODYMessage mm = (MELODYMessage) m;
      } else {
          // m gehört nicht zu MELODYMessage.
          // m kann deshalb nicht zu MELODYMessage umgewandelt
          werden.
      }
```

Die Einführung von individuellen Nachrichtenklassen hat aber einen Nachteil: Sie führt zu einer wahren „Explosion“ an neuen Klassen, denn fast 20 neue Klassen müßten eingeführt werden. Das kaum wünschenswert sein kann. Deshalb wurde ein etwas anderer Weg beschritten. Ein genauer Blick in die TUA Methoden zeigt nämlich, daß sich die meisten bereits eindeutig durch ihre Parameter unterscheiden. Nur die „Listenabfragen“ benötigen noch etwas mehr Information. Im nächsten Abschnitt werden die tatsächlich realisierten Nachrichtenklassen ausführlich beschrieben.

Wenn eine eindeutige Abbildung der ankommenden Nachrichten auf die verschiedenen TUA Aufrufe existiert, muß diese nur noch im Agenten umgesetzt werden. Es muß eine für die vom Aufruf zurückkommende Fehlermeldung ein Objekt angelegt werden. Die hereinkommenden Objekte werden in den richtigen Typ umgewandelt, bevor sie der Methode übergeben werden. Die Resultate müssen wieder in eine Antwortnachricht geschrieben und diese zurück an den Auftraggeber geschickt werden. Am Ende des folgenden Abschnitts über die Melody Nachrichten findet sich ein Ausschnitt aus *receiveMessage*.

Die Melody Nachrichten

Um den Nachrichtenaustausch mit dem Melody Agenten zu vereinfachen, wurden mehrere neue Nachrichtenklassen definiert, durch die zusammen mit den Klassen der Inhaltsobjekte eine eindeutige Abbildung zu den TUA Methoden erfolgen kann. In diesem Abschnitt sollen diese neuen Klassen erklärt werden, so daß verständlich wird, wie diese Abbildung vorgenommen wird.

Die Klasse MELODYMessage

Diese Klasse ist die Basisklasse aller Nachrichten, die vom Melody Agenten akzeptiert oder selbst versendet werden. Empfängt der Agent eine Nachricht, die nicht von dieser Klasse abgeleitet wurde, so wird diese ignoriert! Basis-Klasse dieser Klasse ist die normale Mole-Nachrichtenklasse *Message*. *MELODYMessage* selbst hat keine weiteren Attribute und bietet neben dem einfachen, leeren Konstruktor noch zwei weitere an. Die Attribute und die Methoden der Nachricht sollen hier einmal in einer kurzen Übersicht dargestellt werden.

- *AgentName sender*. Der Name des sendenden Agenten.
- *LocationName senderlocation*. Der Ort, von dem der Sender die Nachricht schickt.
- *AgentName receiver*. Der Name des empfangenden Agenten.
- *LocationName receiverlocation*. Der Ort des Empfängers.
- *int errorsemantics*. Die Fehlersemantik der Nachricht. Wird hier 0 gesetzt, so wird nichts weiter unternommen, wenn die Nachricht nicht ausgeliefert werden kann. Bei einem Wert von 1 wird eine Fehlermeldung zum Sender zurückgesendet.
Durch einen Wert größer als 100 wird die Nachricht verzögert ausgeliefert. Die Größe der Verzögerung in Sekunden errechnet sich dabei aus dem Wert von *errorsemantics* abzüglich 100.
- *long messageid*. Die Id der Nachricht.
- *Object content*. Der Inhalt der Nachricht: Der IN-Parameter.

Die angebotenen Methoden orientieren sich an den Methoden der Basis-Klasse *Message* von Mole. Methoden für *MELODYMessage*:

- *MELODYMessage()*. Der einfache Konstruktor, falls die Attribute alle direkt geschrieben werden sollen.
- *MELODYMessage(AgentName, LocationName, AgentName, LocationName, int, long, Object)*. Der volle Konstruktor, mit dem alle Attribute der Nachricht auf einmal gefüllt werden können. Die ersten beiden Parameter betreffen den Sender, die nächsten zwei den Empfänger. Darauf folgen Fehlersemantik, Id der Nachricht und Inhaltsobjekt.
- *MELODYMessage(AgentName, LocationName, AgentName, LocationName, int, Object)*. Dieser Konstruktor hat den gleichen Aufbau wie der vorangegangene, abgesehen davon, daß die Id der Nachricht weggefallen ist. Sie wird intern initialisiert.

Dieser Nachrichtentyp kann immer dann direkt verwendet werden, wenn die Klasse des Inhaltsobjektes bereits die TUA Methode genau bestimmt. Dies ist zum Beispiel beim Aufruf von *insertType* der Fall. Dort muß nur der erste Parameter der Klasse *STInsert_struct_oo* übertragen werden. Da diese Klasse ausschließlich von *insertType* verwendet wird, ist damit die eindeutige Zuordnung gegeben. Tabelle 1 enthält eine Übersicht über alle für diese Nachrichtentypen zulässigen Inhaltsobjekte sowie ihre Abbildung auf die TUA Methoden.

TABELLE 1. Inhalt von MELODYMessage

Inhalt	Methode
null	getHost
STInsert_struct_oo	insertType
STDelete_struct_oo	deleteType
STModify_struct_oo	modifyType
CInsert_struct_oo	insertContext
CDelete_struct_oo	deleteContext
SInsert_struct_oo	insertService

TABELLE 1. Inhalt von MELODYMessage

Inhalt	Methode
SModify_struc_oo	modifyService
SDelete_struc_oo	deleteService
SList_struc_oo	listService

Die Klasse MELODYListRequest

Mit dieser zweiten Nachrichtenklasse werden alle noch fehlenden TUA Methoden abgedeckt. Es handelt sich dabei um *listTypes*, *listContexts*, *searchService* und *selectService*. Bei allen handelt es sich um Abfragen, die bei gleichen IN-Parameter verschiedene Ergebnisse liefern können. Um dies zu berücksichtigen, wurde von *MELODYMessage* die Klasse *MELODYListRequest* abgeleitet, die um das Attribute *all* erweitert wurde.

- *boolean all*. Dieses Flag gibt an, ob in der Anfrage nur nach einem einzigen Dienstyp, Kontext oder Dienst gesucht wird, oder ob eine Auflistung aller passenden Elemente gewünscht ist.

MELODYListRequest bietet die gleichen Methoden wie *MELODYMessage* auch. Allerdings werden die Konstruktoren, abgesehen einmal vom einfachen Konstruktor, um das neue Attribut erweitert. Für die genaue Bedeutung sei deshalb auf die Übersicht des letzten Abschnitts verwiesen.

- *MELODYListRequest()*. Der einfache Konstruktor.
- *MELODYListRequest(AgentName, LocationName, AgentName, LocationName, int, long, Object, boolean)*. Der volle Konstruktor mit Id der Nachricht. Der letzte Parameter enthält den Wert für das neue Attribute *all*.
- *MELODYListRequest(AgentName, LocationName, AgentName, LocationName, int, Object, boolean)*. Der volle Konstruktor ohne Id der Nachricht.

Die genaue Abbildung der Nachrichten kann der folgenden Tabelle entnommen werden. Zur Unterscheidung der doppelten *listTypes* und *listContexts* Methoden wurde zusätzlich noch die Klasse der zurückgegebenen Liste mit angegeben.

TABELLE 2. Inhalt von MELODYListRequest

Inhalt	all	Methode	Liste
STList_struc_oo	true	listTypes	STListAllReturn_struc_oo
STList_struc_oo	false	listTypes	STListRetrurn_struc_oo
CList_struc_oo	true	listContexts	CListAllReturn_struc_oo
CList_struc_oo	false	listContexts	CListReturn_struc_oo
SWish_struc_oo	true	searchService	SSearchReturn_struc_oo
SWish_struc_oo	false	selectService	SSearchReturn_struc_oo

Die Klasse MELODYPing

Diese Klasse ist die erste der Antwortklassen des Melody Agenten. Sie wird als Antwort auf eine Nachricht ohne Inhaltsobjekt verschickt. Diese Nachricht wird auf die Methode *getHost* abgebildet, die den Namen des Rechners zurückgibt, auf dem der Server läuft. Der Konstruktor dieser Nachricht wurde deshalb speziell auf diese Aufgabe zugeschnitten. Da sie nur als Antwort auf eine andere Nachricht verschickt wird, wurde auf die explizite Angabe von Sender und Empfänger verzichtet. Diese Informationen werden stattdessen aus der Ursprungsnachricht gelesen. Um die Zuordnung der Anfrage zur Antwort sicherzustellen, enthält diese Klasse deshalb als zusätzliches Attribut noch den Wert der Id der Ursprungsnachricht in *requestid*. Dieses Attribut wird allerdings von den Konstruktoren bereits automatisch gesetzt, es sei denn, es wird der einfache Konstruktor verwendet.

- *long requestid*. Die Id der Nachricht, die mit dieser Nachricht beantwortet wird.

Folgende Konstruktoren werden definiert:

- *MELODYPing()*. Der einfache Konstruktor.
- *MELODYPing(MELODYMessage, long, String)*. Dieser Konstruktor erzeugt eine Antwort auf die im ersten Parameter übergebene Nachricht. Mit dem zweiten Parameter wird die Id dieser Nachricht vorgegeben. Der letzte Parameter enthält den Rechnername von *getHost*. Dieser String wird im Inhalt abgelegt.
- *MELODYPing(MELODYMessage, String)*. Wie oben, bloß ohne Angabe der Id.

Diese Nachricht wird als Antwort auf Nachrichten ohne Inhaltsobjekte verschickt. Dann enthält der Inhalt den Rechnernamen. Außerdem wird eine Antwort dieses Typs verschickt, wenn die empfangene Nachricht nicht auf eine TUA Methode abgebildet werden konnte. In diesem Fall wird allerdings das *null* Objekt im Inhalt übertragen.

Die Klasse MELODYReply

Diese Klasse wird für alle Methodenaufrufe verwendet, die keine zusätzlichen Informationen außer den Fehlermeldungen und dem Statuswert zurückgeben. Sie ist wieder speziell auf diese Antworten zugeschnitten und bietet deshalb Konstruktoren, die die zu beantwortende Nachricht, den Statuswert und die Fehlermeldungen als Argument erwarten. Folgende Attribute werden zusätzlich zur Basis-Klasse *MELODYMessage* definiert:

- *long requestid*. Die Id der Nachricht, die mit dieser Nachricht beantwortet wird.
- *int returnCode*. Der Statuswert des Methodenaufrufs.

Folgende Konstruktoren existieren:

- *MELODYReply()*. Der einfache Konstruktor.
- *MELODYReply(MELODYMessage, long, int, Object)*. Der Konstruktor mit Vorgabe der Id der Nachricht im zweiten Parameter. Der erste Parameter ist die zu beantwortende Nachricht. Die letzten beiden Parameter sind der Statuswert und die Fehlermeldungen. Die Fehlermeldungen werden im *content* Attribut abgelegt.
- *MELODYReply(MELODYMessage, int, Object)*. Der gleiche Konstruktor, jedoch ohne Angabe einer Id.

Die Nachricht wird immer auf Methodenaufrufe verschickt, die außer den Fehlermeldungen und Statuswert keine weiteren Informationen zurückgeben. Die Fehlermeldungen werden dabei im Inhalt der Nachricht abgelegt. Im Einzelnen sind dies folgende Methoden: *insertType*, *deleteType*, *modifyType*, *insertContext*, *deleteContext*, *insertService*, *deleteService*, *modifyService*.

Die Klasse MELODYListReply

Für die Antworten der jetzt noch nicht abgedeckten Methoden des TUAs dient diese Klasse. Sie wird für alle Methoden verwendet, die neben den Fehlermeldungen noch weitere Informationen, wie zum Beispiel eine Liste von Dienstypen, zurückgeben. Wie die Klassen zuvor auch, ist sie in besonderer Weise auf ihre Aufgabe zugeschnitten. Ihre Basis-Klasse ist *MELODYReply*. Sie definiert aber noch ein weiteres zusätzliches Attribut, in dem die Liste zurückgegeben wird.

- *Object list*. Die vom Methodenaufruf zurückgegebene Liste.

Es werden wieder die üblichen drei Konstruktoren angeboten, wobei als zusätzliches Argument die Liste übergeben werden kann.

- *MELODYListReply()*. Der einfache Konstruktor.
- *MELODYListReply(MELODYMessage, long, int, Object, Object)*. Mit diesem Konstruktor wird das gesamte Objekt initialisiert. Die ersten drei Argumente sind wie in *MELODYReply* bereits auch die zu beantwortende Nachricht, die Id dieser neuen Nachricht sowie der Statuswert des Methodenaufrufs. Der vierte Parameter ist die vom Methodenaufruf zurückgegebene Liste, die im Attribut *list* gespeichert wird. Der letzte Parameter sind wieder die Fehlermeldungen.
- *MELODYListReply(MELODYMessage, int, Object, Object)*. Gleicher Konstruktor wie oben, ohne explizite Angabe der Nachrichten-Id.

Die Fehlermeldungen werden wieder im Inhalt der Nachricht abgespeichert. Auf folgende Methodenaufrufe kann diese Nachricht folgen: *listTypes*, *listContexts*, *listService*, *searchService*, *selectService*.

Die Klasse MELODYError

Im Fall, daß ein Aufruf an den Server nicht erfolgen kann, wird von CORBA eine Exception erzeugt. Dieser Fehlerfall wird durch *MELODYError* abgedeckt. Zu genaueren Beschreibung des Fehlers wird zusätzlich als Inhalt der Nachricht die Exception selbst mitgeschickt. Ihr Aufbau ist ähnlich wie *MELODYPing*. Sie definiert zusätzlich zur Basis-Klasse *MELODYMessage* das Attribut *requestid*, welches eine Zuordnung der Nachricht zur Auftragsnachricht ermöglicht.

- *long requestid*. Die Id der Nachricht, die diesen Fehler verursachte.

Wie üblich werden drei Konstruktoren definiert:

- *MELODYError()*. Der einfache Konstruktor.

- *MELODYError(MELODYMessage, long, Object)*. Dieser Konstruktor erzeugt eine Antwort auf die Nachricht, die im ersten Parameter übergeben wird. Die Id der Nachricht wird explizit auf den Wert des zweiten Parameters gesetzt. Die Exception wird an dritter Stelle übergeben.
- *MELODYError(MELODYMessage, Object)*. Der gleiche Konstruktor ohne Angabe der Id.

Die Exception wird im Inhaltsattribut *content* der Nachricht abgelegt.

Folgende Übersicht zeigt noch einmal das Zusammenspiel aller Nachrichtenklassen mit den entsprechenden Methodenaufrufen:

TABELLE 3. Übersicht über die Nachrichtenklassen und ihre Abbildung

Methode	Auftrag ^a	Inhalt ^b	all	Antwort ^c	Inhalt ^d	list ^e
getHost	null	–	–	MP	String ^f	–
insertType	MM	STInsert	–	MR	ErrorMessage	–
deleteType	MM	STDelete	–	MR	ErrorMessage	–
modifyType	MM	STModify	–	MR	ErrorMessage	–
listTypes	MLR	STList	true	MLR	ErrorMessage	STListAllReturn
listTypes	MLR	STList	false	MLR	ErrorMessage	STListReturn
insertContext	MM	CInsert	–	MR	ErrorMessage	–
deleteContext	MM	CDelete	–	MR	ErrorMessage	–
listContexts	MLR	CList	true	MLR	ErrorMessage	CListAllReturn
listContexts	MLR	CList	false	MLR	ErrorMessage	CListReturn
insertService	MM	SInsert	–	MR	ErrorMessage	–
deleteService	MM	SDelete	–	MR	ErrorMessage	–
modifyService	MM	SModify	–	MR	ErrorMessage	–
listService	MM	SList	–	MLR	ErrorMessage	SList
searchService	MLR	SWish	true	MLR	ErrorMessage	SSearchReturn
selectService	MLR	SWish	false	MLR	ErrorMessage	SSearchReturn
<i>sonst</i>	?	?	?	MP	null	–
<i>Fehler</i>	?	?	?	ME	Exception	–

a. MM = *MELODYMessage*. MLR = *MELODYListRequest*

b. Hier sind die Zugriffsmethoden genannt. Die CORBA-Datenstrukturen haben zusätzlich die Endung „_struc_oo“.

c. MP = *MELODYPing*. MR = *MELODYReply*. MLR = *MELODYListReply*. ME = *MELODYError*.

d. Zur kürzeren Darstellung ist hier *ErrorMessage* statt *ErrorMessage_struc_oo* geschrieben!

e. Den Namen muß noch die Endung „_struc_oo“ angehängt werden.

f. Name des Rechners des Melody-Servers.

Verwendung der Nachrichtenklassen im Melody Agenten

In diesem letzten Abschnitt wird noch an einem Ausschnitt aus der *receiveMessage* Methode des Melody Agenten die Verwendung der eben definierten Nachrichtenklassen erläutert. Es soll nur anhand zweier verschiedener TUA-Aufrufe wesentliche Aspekte der Implementierung erläutert werden. Den Anfang macht wieder der Aufruf des TUAs zum Einfügen eines neuen Diensttyps *insertType*.

```
Java: void receiveMessage(Message mes) {
    // initialisiere die Antwort auf null.
    Message ret = null;

    if (!(mes instanceof MELODYMessage)) {
        // Keine Melody Nachricht. Abbruch.
        return;
    }

    // Mache eine MELODYMessage aus der Nachricht
    MELODYMessage m = (MELODYMessage) mes;
    ...
    // Umsetzung der Aufrufe
    if (m.content instanceof STInsert_struct_oo) {
        // insertType...
        ret_value = tUAgent.insertType(
            (STInsert_struct_oo) m.content, returnMes);

        // Erstelle die Antwortnachricht.
        ret = new MELODYReply(m, ret_value, returnMes);
    } else
    ...
    // schicke die Antwort zurück
    actuallocation.message(ret);
    ...
}
```

receiveMessage überprüft zunächst einmal, ob die empfangene Nachricht eine Melody-Nachricht ist. Ist dies nicht der Fall, so wird sofort abgebrochen. Um den Inhalt korrekt lesen zu können, wird die Nachricht mit einem Cast einer Variablen der richtigen Klassen zugewiesen. Die genaue Unterscheidung des richtigen Methodenaufrufs geschieht durch Abfrage der Klasse des Nachrichteninhalts. Wenn es sich um *STInsert_struct_oo* handelt, kann ein Aufruf an die TUA-Methode *insertType* gemacht werden. Der Speicher für die Fehlermeldung in *returnMes* wurde bereits vorher einmal belegt. Nachdem der Aufruf zurückkehrt, muß die Antwortnachricht angelegt werden. Für die Antwort auf *insertType* reicht *MELODYReply* aus. Der Statuswert in *ret_value* und die Fehlermeldungen in *returnMes* werden als Parameter dem Konstruktor übergeben. Schließlich wird am Ende der Bearbeitung die Antwort an den ursprünglichen Sender zurückgesandt.

Bei Empfang einer Nachricht für die Abfrage von Diensttypen wird die Umsetzung des Aufrufes etwas komplizierter. Es handelt sich bei ihr um eine Nachricht der Klasse *MELODYListRequest*. Der Inhalt der Nachricht ist von der Klasse *STList_struct_oo*. Je nach dem, ob das

Attribut *all* in der Nachricht gesetzt wurde oder nicht, unterscheidet sich der Aufruf und das Ergebnis. Diese Unterscheidung muß berücksichtigt werden.

```
Java: ...
    boolean all = false;

    // Bei ListRequest Nachrichten muss das all
    // Attribut abgefragt werden.
    if (m instanceof MELODYListRequest) {
        all = ((MELODYListRequest) m).all;
    }
    ...
    } else if (m.content instanceof STList_struct_oo) {
        // Aufruf fuer listTypes

        if (all) {
            // Abfrage aller passenden Diensttypen
            // Die Ergebnisse werden von der Methode
            // in STListAllReturn_struct_oo zurückge-
            // geben.
            STListAllReturn_struct_oo returnList
                = new STListAllReturn_struct_oo();

            // Aufruf der Methode
            ret_value = tUAgent.listTypes(
                (STList_struct_oo) m.content,
                returnList, returnMes);

            // Erstellen der Antwortnachricht.
            ret = new MELODYListReply(
                m, ret_value, returnList, returnMes);
        } else {
            // Abfrage eines Datentyps.
            // Die Ergebnisklasse ist STListReturn.
            STListReturn_struct_oo returnList
                = new STListReturn_struct_oo();

            // Auf der Methode
            ret_value = tUAgent.listTypes(
                (STList_struct_oo) m.content,
                returnList, returnMes);

            // Erstellen der Antwort
            ret = new MELODYListReply(
                m, ret_value, returnList, returnMes);
        }
    }
    ...
```

Die Nachricht enthält zusätzlich zu *MELODYMessage* noch das zusätzliche Attribut *all*. Dieses Attribut wird zunächst in einer gleichnamigen lokalen Variablen abgespeichert. Dadurch muß später nicht mehr zwischen den beiden Nachrichtenklassen unterschieden werden. Ein Aufruf für *listTypes* wird durch den Inhalt von der Klasse *STList_struct_oo* angezeigt. Doch es gibt zwei verschiedene Methoden von *insertType*: Eine gibt die komplette Liste der passenden Dienstypen zurück, die andere gibt nur einen zurück. Zu dieser Unterscheidung wird das *all* Attribut abgefragt. Daraufhin kann ein Ergebnisobjekt der passenden Klasse angelegt werden und der Aufruf der Methode erfolgt. Die Ergebnisliste wird in einer Nachricht der Klasse *MELODYListReply* zurückgegeben.

Durch dieses Vorgehen läßt sich der Agent sehr einfach verwenden. Ein Klient erstellt das Parameterobjekt mit den Zugriffsklassen. Daraufhin sendet er die richtige Nachricht mit dem Parameter als Inhalt an den Agenten. Nachdem die Bearbeitung abgeschlossen ist, sendet der Melody Agent die Ergebnisse in einer neuen Nachricht an den Klienten zurück. Da der Nachrichtenaustausch auch über die Grenzen eines Ortes hinweg stattfindet, muß sich ein Auftraggeber nicht am Ort des Melody Agenten aufhalten. Zudem kann die Verwendung durch die Parallelarbeit von Klient und Melody Agent effizienter sein.

Der im letzten Kapitel beschriebene Melody Agent wurde hauptsächlich entwickelt, um die generelle Machbarkeit der Einbindung zu zeigen. Das Vorgehen mit der Verwendung der CORBA-Datenstrukturen zur Übertragung ist nicht unbedingt als optimal zu bezeichnen, da diese Datenstrukturen sehr umfangreich sind. Besonders für Agenten, die selbst relativ klein sind und einfache Aufgaben haben, kann dies sehr nachteilig sein. Es ist also zu untersuchen, ob der Melody Agent nicht zusätzlich noch eine weitere Schnittstelle anbietet, bei der ein anderer Weg beschritten wird. Besonders einfach wäre es natürlich, wenn der Zugriff durch Nachrichten geschehen könnte, die nur Texte enthalten. Dazu wäre es notwendig, eine Anfragesprache zu definieren, die eine Abbildung in die richtigen Datenstrukturen erlaubt.

Auch generell ist noch zu untersuchen, wie Agenten Trading-Dienste nutzen sollten. Der vorliegende Melody Agent erlaubt es zwar prinzipiell Agenten, ihre Dienste von Melody verwalten zu lassen, doch fehlen noch Konzepte, wie dies konkret erfolgen sollte. Bei einer Vermittlung von Melody muß der Dienstbringer genau spezifiziert werden. Dazu sind vermutlich der Agentenname und sein Aufenthaltsort notwendig. Für Systemagenten, die ortgebunden sind, ist diese Angabe kein Problem. Möchte man aber auch normale Benutzeragenten vermitteln lassen, so kommt die Schwierigkeit hinzu, diesen Agenten auch nach der Migration noch zu finden.

Wie bereits bei der Beschreibung der Java Schnittstelle erklärt wurde, wird im Moment keine Unterstützung des Managementsystems von Melody angeboten. Der Zugriff auf dynamische Attribute von Diensten ist nicht möglich, da die unterliegenden Mechanismen für Anwendungen unter Java noch nicht definiert sind. Es wäre aber natürlich sehr gut, wenn ein solcher Zugriff ermöglicht würde.

Genauso wünschenswert wird wohl auch die Kombination von CORBA und Mole in umgekehrter Richtung sein: Über CORBA könnte eine Schnittstelle von Agenten nach außen angeboten werden, so daß es CORBA-Klienten möglich ist, auf Mole-Agenten zuzugreifen. Diese Dienstleistungen von Agenten könnten dann durch Melody vermittelt werden. In der dieser Arbeit zugrundeliegenden Version von OrbixWeb ist dies aber noch nicht möglich. OrbixWeb

bietet zur Zeit nur Funktionen für die Seite des Klienten an. Server unter Java können noch nicht eingebunden werden.

Auch sonst fehlen noch weitere vereinfachende Ansätze für Agenten, wenn sie CORBA nutzen wollen. Im Moment müssen sie sich direkt der OrbixWeb Software bedienen und so mit CORBA-Objekten direkt über den ORB kommunizieren. Schnittstellenagenten für CORBA-Dienste könnten hier Abhilfe schaffen. Bereits ein einfacher Agent, der das Dynamic Invocation Interface anbietet, könnte eine große Erleichterung sein. Möglicherweise aber wird es sogar möglich sein, spezifische Schnittstellenagenten zu erzeugen, die ähnlich den erzeugten Stub-Methoden den Zugriff erleichtern. Im Idealfall erfolgt diese Erzeugung zur Laufzeit des Agentensystems durch eine Abfrage des Interface Repositories.

Noch ein paar abschließende Worte zur CORBA-Schnittstelle von Melody. Mit dieser neu geschaffenen Schnittstelle eröffnet sich ein neues Spektrum von Anwendungen, die den Trader nutzen können. Der Zugriff auf Melody wird nicht weiter eingeschränkt. Ein Klient muß nur in der Lage sein, mit einem ORB zu kommunizieren und die Datenstrukturen von Melody entsprechend seines IDL-Mappings auszufüllen. Doch wie dies bereits bei Java der Fall war, ist dieser Weg sehr unbequem und erfordert Wissen über den grundsätzlichen Aufbau dieser Strukturen. Aber es dürfte in jeder Programmiersprache möglich sein, eine Schnittstelle ähnlich dem existierenden TUA zu implementieren, die den Aufbau von Anfragen über Zugriffsmethoden – wie hier für Java geschehen – zu ermöglichen. Der Vorteil solcher Lösungen ist immer, daß damit die Verwendung immer in der gleichen Weise wie unter C++ geschehen kann. Der Nachteil dabei ist natürlich, daß eventuell besondere Möglichkeiten der Programmiersprache oder des Mappings nicht genutzt werden können.

Es bleibt also festzustellen, daß die drei großen Bereiche dieser Arbeit, Melody, Mole und CORBA sowie ihr Zusammenspiel noch eine große Zahl interessanter Fragen bereithält, die in der Zukunft auftauchen werden.

-
- [1] Burger Cora, Kovacs Ernő, *Projektbeschreibung MELODY* Technischer Bericht 8/95, Universität Stuttgart, Fakultät für Informatik (<http://www.informatik.uni-stuttgart.de/ipvr/vs/Publications/Publications.html#1995-kovacs-01>)
 - [2] Campione Mary, Walrath Kathy, *The Java Tutorial* Addison-Wesley, 1996 (<http://java.sun.com/books/Series/Tutorial/index.html>)
 - [3] Gosling James, Yelling Frank, The Java Team, *Java API Documentation Version 1.0.2* 1996 (<http://java.sun.com/products/JDK/CurrentRelease/api/packages.html>)
 - [4] Hohl Fritz, *Konzeption und Implementierung eines einfachen Mobile Agenten Systems* Universität Stuttgart, Fakultät für Informatik, Diplomarbeit Nr. 1267, 1995 (http://www.informatik.uni-stuttgart.de/cgi-bin/ncstrl_rep_view.pl?/inf/ftp/pub/library/ncstrl.ustuttgart_fi/DIP-1267/DIP-1267.bib)
 - [5] Hohl Fritz, *Mole Alpha 1.0 Documentation* Universität Stuttgart, Fakultät für Informatik, IPVR, Abteilung Verteilte Systeme, 1996 (<ftp://suntrec.informatik.uni-stuttgart.de/pub/MOLE/documentation.html>)
 - [6] IONA Technologies Ltd., *Orbix for Java White Paper* February 1996(<http://www-usa.iona.com/www/Orbix/Java/index.html>)
 - [7] IONA Technologies Ltd., *Orbix2 Reference Manual*
 - [8] IONA Technologies Ltd., *The OrbixWeb API* May 1996 (<http://www-usa.iona.com/www/Orbix/JavaAPI/intro.html>)
 - [9] IONA Technologies Ltd., *The OrbixWeb Programming Guide* 1996 (<http://www-usa.iona.com/www/Orbix/OrbixWeb/doc/pguide/pguide.toc.html>)

-
-
- [10] Kernighan Brian W., Ritchie Dennis M., *The C Programming Language, Second Edition* Prentice Hall Software Series, 1988
 - [11] Object Management Group, *The Common Object Request Broker: Architecture and Specification* Revision 2.0 July 1995 (auch über <http://www.omg.org/corba2/corb2prf.htm>)
 - [12] Stroustrup Bjarne, *The C++ Programming Language, Second Edition* Addison-Wesley 1994
 - [13] Sun Microsystems, *The Java Language Specification, Version 1.0 Beta Draft* (<ftp://ftp.javasoft.com/docs/javaspec.ps.zip>)

Ich versichere, daß ich diese Arbeit selbständig verfaßt habe und nur die angegebenen Hilfsmittel verwendet habe.

Gerald Vogt
