



Diplomarbeit 1734

Visualisierung von Datenbank-Abfragen in
Java

Ralf Engelmann

Ausarbeitung

Arbeit Nr: 1734	Art: DA	Vorgehensmodellklasse: Modell zur Softwareentwicklung	Bearbeiter: Engelmann	Betreuer: SI/Ja	Beginn: 01.05.1999	Ende: 31.10.1999
Dokument: Ausarbeitung		Version: 1.0	Autor: Engelmann	Datum: 08.11.99	Status: akzeptiert	
1.0 akzeptiert			Engelmann	SI 11.11.99 10:33	07.11.99	

Kurzfassung

In vielen Industriefirmen sind Produktionsanlagen mit Datenbanken verbunden, in welchen die anfallenden Prozessdaten gespeichert werden. Diese Prozessdaten werden nach verschiedenen Kriterien mit Hilfe von Datenbank-Abfragen ausgewertet und die Ergebnisse werden grafisch aufbereitet.

Die Aufgabe des dieser Arbeit zu Grunde liegenden Projektes war es, solche Prozessdaten durch ein zu entwickelndes Anwendungssystem, basierend auf Java-Technologie in einem WWW-Browser zu präsentieren.

CORBA (Common Object Request Broker Architecture) bietet ein standardisiertes Verfahren zur Realisierung verteilter Anwendungen. Die zu entwickelnde Anwendung wurde als verteilte Anwendung auf der Basis einer Three-tier-Anwendungsarchitektur entworfen und mit Hilfe der CORBA-Technologie realisiert. Der Einsatz der CORBA-Technologie in diesem Projekt erlaubt den flexiblen und intelligenten Zugriff auf die Datenbank und die gleichzeitige Verarbeitung der gelesenen Daten. Die Präsentation der aufbereiteten Daten erfolgt in einem Java-Applet, welches in einem WWW-Browser ausgeführt wird und durch die erwähnte CORBA-Technologie die Daten erhält.

Dieses Projekt wurde in Zusammenarbeit mit der Industriefirma adstec Automation, Daten- und Systemtechnik GmbH (Oberaichen) durchgeführt.

Abstract

Manufacturing plants in many companies are connected with databases storing data produced by the plants processes. The data coming from the plants processes is examined by various criteria using database queries and the results are prepared for use in graphics.

In this project an application system had to be developed. This system had to present the data generated from those processes in an WWW-Browser by using Java.

CORBA (Common Object Request Broker Architecture) offers standardized methods to develop distributed applications. The application developed in this project has been designed as a distributed application based on a three-tier-architecture-model using CORBA technology. The employment of CORBA in this project allows the development of flexible and intelligent methods to access the database and to prepare the data for the presentation simultaneously. The presentation of the prepared data is made available in an Java-Applet running in a WWW-Browser. The prepared data is transmitted to the Applet using CORBA technology.

This work has been made in cooperation with adstec Automation, Daten und Systemtechnik GmbH (Oberaichen).

0 Inhaltsverzeichnis

0	INHALTSVERZEICHNIS	3
1	EINFÜHRUNG	4
1.1	Aufgabenstellung	4
1.2	Ausgangsbedingungen	4
1.3	Anforderungen an das System.....	6
2	GRUNDLAGEN DER VERWENDETEN TECHNOLOGIEN.....	7
2.1	Relationale Datenbanksysteme	7
2.1.1	Konzept relationaler Datenbanksysteme	7
2.1.2	Struktur und Funktionen der Abfragesprache SQL.....	10
2.1.3	Datenbankschnittstellen.....	12
2.2	Java - Datenbankfunktionen.....	13
2.2.1	Java Klassen und Funktionen	14
2.2.2	Treiber und Schnittstellen.....	15
2.3	CORBA	16
2.3.1	Architektur verteilter Anwendungen	16
2.3.2	CORBA-Architektur.....	19
2.3.3	Erstellung einer CORBA-Anwendung in der Praxis.....	26
2.3.4	Alternativen zu CORBA.....	32
2.3.5	CORBA-Produkte.....	33
3	REALISIERUNG	36
3.1	Grundkonzepte der Anwendung.....	36
3.2	Applet oder Anwendung ?.....	39
3.2.1	Unterschiede im Sicherheitskonzept	41
3.2.2	Laden von Bilddateien.....	42
3.2.3	Initialisierung des ORB's.....	43
3.3	Visualisierungsklassen	44
3.4	Datenbanknavigation.....	48
3.4.1	Menükomponente	48
3.4.2	Weitere Navigationselemente.....	52
3.5	Anwendung der CORBA-Technologie	55
3.5.1	Anwendung von JavaIDL.....	55
3.5.2	Anwendung von Jaguar	60
4	ERFAHRUNGSBERICHT	63
	ANHANG	66
	Literaturverzeichnis	66
	Abbildungsverzeichnis	67
	Verzeichnis Codebeispiele und vervollständigte Codetexte	68
	Index	70

1 Einführung

In vielen Industriefirmen sind Produktionsanlagen mit SQL-Datenbanken verbunden, in welchen die anfallenden Prozessdaten gespeichert werden. Diese Prozessdaten werden nach verschiedenen Kriterien mit Hilfe von Datenbank-Abfragen ausgewertet und die Ergebnisse werden grafisch aufbereitet.

1.1 Aufgabenstellung

In Zusammenarbeit mit der Industriefirma ads-tec Automation-, Daten und Systemtechnik GmbH soll im Rahmen dieser Diplomarbeit ein auf Java basierendes Anwendungsprogramm entwickelt werden, das die Abfragen auf SQL-Datenbanken und die Visualisierung der Ergebnisse unabhängig vom verwendeten Betriebssystem gestattet.

Damit der Einsatz der Anwendung möglichst benutzerfreundlich gestaltet werden kann, soll die Anwendung als Java-Applet ausgeführt werden, das in HTML-Seiten eingebettet und auf einem Web-Server bereitgestellt wird.

Durch die Bereitstellung der Anwendung auf einem Web-Server sollen Modifikationen und Wartungen an der Anwendung vereinfacht werden.

1.2 Ausgangsbedingungen

Die Prozessdatenbank ist das Archivierungsmedium eines Systems zur Erfassung der Prozessdaten einer Zylinderkopfhärteanlage.

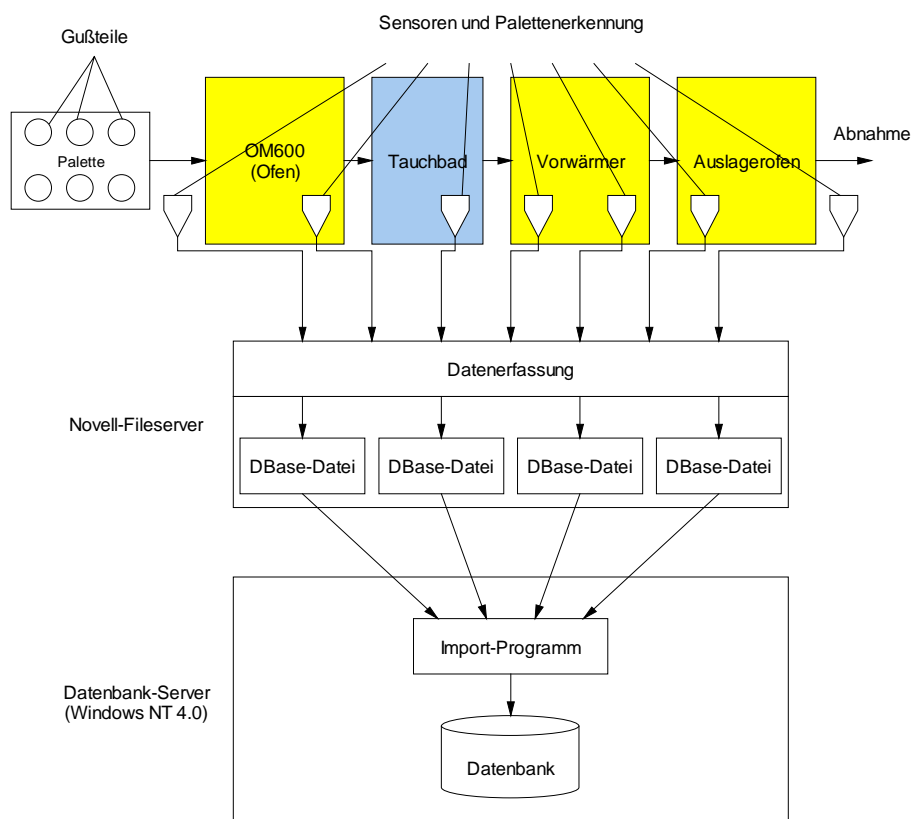


Abbildung 1.1: Zylinderkopfhärteanlage OM600 und Prinzip der Datenerfassung

Die Anlage besteht aus mehreren Öfen mit nachgeschalteten wassergefüllten Abschreckbecken, in der die Zylinderköpfe als Gussteile auf so genannten Paletten prozessiert werden. An verschiedenen Stellen der Anlage befinden sich Sensoren und Palettenerfassungseinrichtungen, die ihre Daten an ein Datenerfassungssystem weitergeben. Das Datenerfassungssystem speichert die Daten in DBase-Datenbankdateien auf einem Fileserver. Mit einem Importprogramm werden die Daten in die Prozessdatenbank importiert.

Der Kunde von ads-tec, der diese Zylinderkopfhärteanlage betreibt, hatte eine Fremdfirma beauftragt, ein Programm zur Auswertung der Datenbankdaten zu implementieren. Die Anwendung wurde mit dem Microsoft Visual Basic Programmierpaket erzeugt, und erfüllte die Erwartungen des Kunden nur ungenügend. Insbesondere die Ausführungsgeschwindigkeit, die Stabilität und die Benutzerführung waren Hauptkritikpunkte des Kunden.

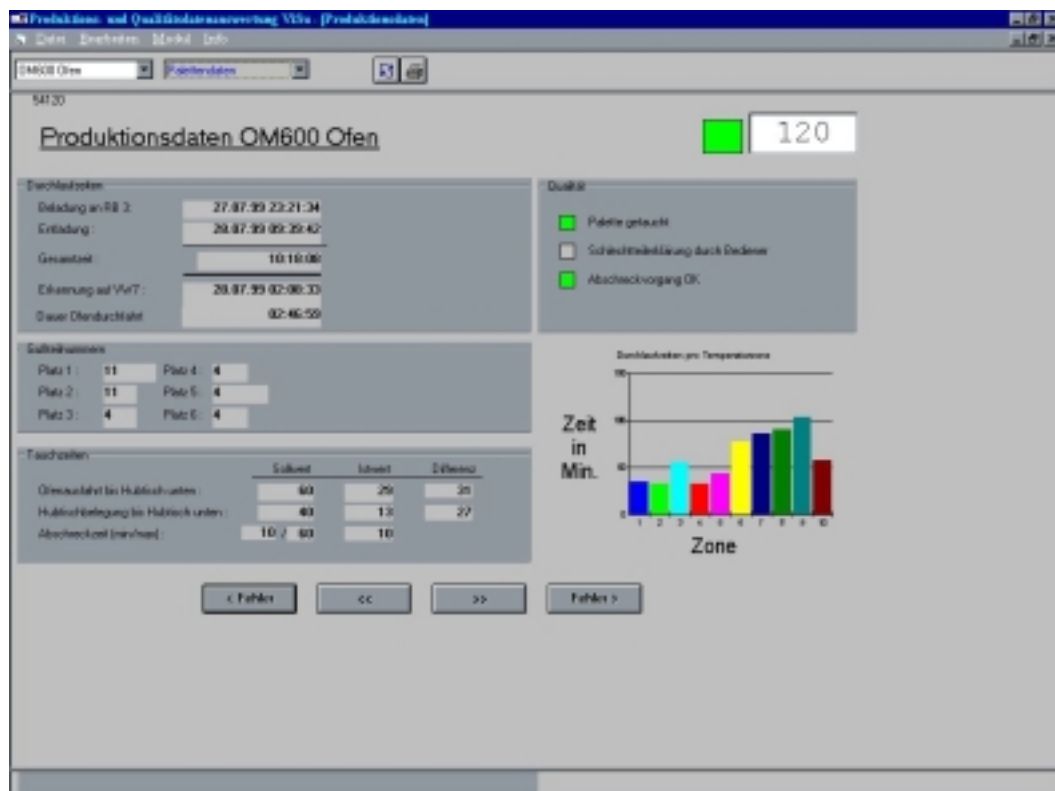


Abbildung 1.2: Programm zur Auswertung der Datenbank

Die Prüfung der Visual-Basic-Anwendung ergab, dass sich auf Grund der sehr großen Resultatmengen bestimmter Datenbankabfragen erhebliche Performanceprobleme einstellten. Die geringe Performance der Visual-Basic-Anwendung war zurückzuführen auf die sehr hohen Datenmengen, die zwischen Datenbankserver und Anwendung übertragen wurden und die auf Grund der hohen Datenmengen zeitintensive Verarbeitung der Daten im Programm. Messungen ergaben, dass nach Aufruf des Programms, Einstellung der Anzeige und der Anzeige der Daten bis zu 15Mbyte Daten übertragen wurden. Auf einer ISDN-Verbindung dauerte es bis zu 30 Minuten bis die Anzeige aufgebaut war, in dem LAN des Kunden immerhin noch ca. 10min.

Es musste daher für die zukünftige Entwicklung eine Lösung gefunden werden, die für die Anzeige notwendigen zu übertragenden Daten zu reduzieren. Die Reduzierung der zu übertragenden Daten auf ein Mindestmaß sollten zu schnelleren Antwortzeiten des Auswertungsprogramms und auf Grund der geringeren Größe, der im Speicher abgelegten Datenstrukturen auch zu einer verbesserten Stabilität führen. Für die Benutzerführung musste eine weitgehend intuitiv zu bedienende Oberfläche geschaffen werden.

1.3 Anforderungen an das System

Das Datenbank-Server-System und die darauf installierten Prozessdatenbanken waren von der Aufgabenstellung her vorgegeben:

- Servermaschine auf Intel Pentium II Prozessorbasis
- Betriebssystem Windows NT 4.0
- Datenbanksystem Microsoft SQL-Server Version 6.5

Das Lastenheft formulierte hierzu weitere Anforderungen an das zu entwickelnde System:

- Das neue System sollte mindestens die Funktionalität der zu ersetzenden Datenbank-Anwendung abdecken können. (/LF10/)
- Das neue System sollte auf dem bestehenden Datenbanksystem (inklusive Datenstruktur) aufsetzen. (/LF20/)
- Die Auswertung und Abfrage der Prozessdaten soll von mehreren Benutzern gleichzeitig durchgeführt werden können. (/LF30/)
- Die Anwendung sollte durch einen WWW-Browser genutzt werden können. (/LF40/)

Als Entwicklungsumgebung für Java sollte Sybase PowerJ Enterprise zum Einsatz kommen. Mit der Entwicklungsumgebung mitgeliefert wurde Sybase Jaguar CTS, ein auf CORBA basierendes Produkt zur Entwicklung verteilter Anwendungen. Der Einsatz dieses Produktes und der zu Grunde liegenden Technologie sollte im Rahmen der Arbeit bei der Erstellung des Systems geprüft werden.

2 Grundlagen der verwendeten Technologien

2.1 Relationale Datenbanksysteme

Datenbanken sind ein Mittel um Daten strukturiert zu speichern und auf Grund der Strukturierung auswertbar zu machen.

Die einfachste Form einer Datenbank ist eine Textdatei. Die Struktur eines Textes wird durch die Grammatik beschrieben. So lassen sich Texte in Sätze, Nebensätze und Wörter unterteilen. Die Wörter eines Satzes lassen sich bestimmten Klassen, wie Substantiv, Adjektiv, Verb, usw. zuordnen. Die Auswertung einer Grammatik ist jedoch für ein EDV-System ein sehr großes Problem. Eine Grammatik ist von der zu Grunde liegenden Zielsprache bestimmt, und weist in mehr oder weniger häufigen Fällen Ausnahmen auf.

Für die Verarbeitung von Daten in EDV-Systemen wird daher häufig ein viel einfacheres Strukturierungsmodell gewählt: die Tabelle. Tabellen lassen sich mathematisch durch Matrizen beschreiben und somit durch iterative Algorithmen in einem EDV-System auswerten.

2.1.1 Konzept relationaler Datenbanksysteme

Das Abspeichern von Daten in einer einzigen Tabelle kann jedoch sehr ineffizient sein. Der Grund besteht darin, dass die Strukturierung von Daten in einer Tabelle zu Redundanz der Daten führt. Unter Redundanz versteht man das mehrfache Abspeichern gleicher Daten in einer Tabelle.

Name	Straße	Ort	Buchtitel	Autor	Leihbeginn	Leihende
Franz	Sturmstraße	Stuttgart	Micky Maus	Disney	1.10.99	1.11.99
Hans	Allee	Fellbach	Transistoren	v. Münch	3.9.99	3.10.99
Franz	Sturmstraße	Stuttgart	Gute Küche	Bocuse	4.8.99	3.9.99
Tina	Schlossstraße	Vaihingen	Micky Maus	Disney	15.9.99	15.10.99
Helga	Poststraße	Fellbach	Faust	Goethe	7.8.99	8.9.99

Abbildung 2.1: Beispieldatenbank Bibliothek

Man erkennt an den farbig hinterlegten Bereichen die Redundanz der Tabelle.

Die in der obigen Abbildung dargestellte Datenbanktabelle enthält Redundanz. Man nennt diesen Zustand einer Datenbanktabelle 1. Normalform, da die Spalten bereits nur atomare Werte, d.h. einfache voneinander unabhängige Daten, enthalten. Redundanz in einer Datenbanktabelle führt zu einem erhöhten Speicherplatzbedarf und erhöht die Gefahr von Inkonsistenz der gespeicherten Daten, wenn z.B nicht alle Einträge in dem redundanten System synchronisiert geändert werden.

Das relationale Datenbanksystemmodell wurde entwickelt, um diese Redundanz zu minimieren. Das Grundprinzip des relationalen Modells besteht darin die Datenbank aus

mehreren miteinander verknüpften Tabellen aufzubauen. Diesen Vorgang bezeichnet man als Normalisierung.

Angewendet auf obiges Beispiel ergeben sich folgende Tabellen:

Tabelle: Kunden

Kunden-Nr	Name	Straße	Ort
1	Hans	Allee	Fellbach
2	Franz	Sturmstraße	Stuttgart
3	Tina	Schlossstraße	Vaihingen
4	Helga	Poststraße	Fellbach

Tabelle: Bücher

Buch-Nr.	Buchtitel	Autor
1	Micky Maus	Disney
2	Transistoren	v. Münch
3	Gute Küche	Bocuse
4	Faust	Goethe

Tabelle: Leihvorgänge

Kunden-Nr.	Buch-Nr.	Leihbeginn	Leihende
2	1	1.10.99	1.11.99
1	2	3.9.99	3.10.99
2	3	4.8.99	3.9.99
3	1	15.9.99	15.10.99
4	4	7.8.99	8.9.99

Abbildung 2.2: Beispiel für eine redundanzreduzierte Datenbank (2. Normalform)

Wie das Beispiel zeigt wurde die Redundanz entfernt. Keine der Tabellen weist in irgendwelchen Teilbereichen (Teilbereich ≥ 2 Zellen) identische Werte auf.

In der Tabelle Leihvorgänge werden der Kunden und die entleihenden Bücher des Bestandes referenziert. In dieser Form der Datenbank (2. Normalform) ist eine eindeutige Zuordnung zwischen Kunden, Büchern und Leihvorgängen möglich. Der Vorteil in der Praxis besteht darin, dass nicht bei jedem neuen Vorgang sämtliche Daten der Kunden neu eingegeben werden müssen, sondern nur die Tabelle Leihvorgänge um die entsprechenden Einträge ergänzt werden muss. Der Übergang von der einfachen Tabelle zu einem System verknüpfter Tabellen bezeichnet man als ‚Normalisierung‘.

In den Tabellen Kunden und Bücher lässt sich jede Zeile (Datensatz) eindeutig über ein einziges Merkmal identifizieren. In der Tabelle Kunden ist dies die Spalte Kunden-Nummer; in der Tabelle Bücher ist dies die Buch-Nummer. Dieses eindeutige Identifizierungsmerkmal bezeichnet man als Primärschlüssel. Ist der Wert des Primärschlüssels bekannt, lassen sich die übrigen Werte eines Datensatzes so sehr schnell ermitteln. Dieser Weg ist in der Tabelle

Leihvorgänge nicht möglich. Diese enthält lediglich die so genannten Fremdschlüssel als Referenz auf die beiden anderen Tabellen. Die Fremdschlüssel in einer Tabelle einer relationalen Datenbank sind dadurch gekennzeichnet, dass sie nur Werte der Primärschlüssel der referenzierten Tabelle enthalten können. Eine eindeutige Identifizierung wäre im vorgestellten Beispiel nur möglich, wenn eine weitere Spalte als Primärschlüssel hinzugefügt werden würde.

Das Einfügen eines eindeutigen Identifizierungsmerkmals in Form einer zusätzlichen Spalte mit 'künstlichen', d.h. nicht im realen Datenmodell vorhandenen Werten, ist ein recht häufiger Vorgang beim Design solcher Datenbanken. Meist wird eine zusätzliche Spalte mit Integer-Werten hinzugefügt.

Die wesentliche Funktion eines relationalen Datenbanksystems besteht darin die verschiedenen Tabellen einer Datenbank anhand der Schlüsselwerte zu verknüpfen und auf Grund dieser Verknüpfungen (Relationen, Entitäten) für die Integrität der gespeicherten Daten zu sorgen.

Bei der Verknüpfung der verschiedenen Tabellen unterscheidet man mehrere Typen:

- 1:1 Verknüpfung

Zu jedem Datensatz einer Tabelle lässt sich genau ein Datensatz einer anderen Tabelle zuordnen. Bei näherer Betrachtung erscheint diese Zuordnung unnötig. Wenn einer der Datensätze einer Tabelle nur mit jeweils einem einzigen Datensatz einer anderen Tabelle verknüpft ist, tritt keine Redundanz auf. 1:1- Verknüpfung werden daher meistens nur dann durchgeführt, wenn die Aufteilung in mehrere Tabellen der Übersichtlichkeit dient. Ein weiterer Grund für die Wahl dieses Verknüpfungstyps bei Entwurf und Definition von Datenbanktabellen kann darin gesehen werden, dass eine Tabelle sonst zu viele leere Felder enthalten würde. Dieses Problem lässt sich allerdings durch den folgenden Beziehungstyp umgehen.

- 1:n Verknüpfung

Zu einem Datensatz einer Tabelle lassen sich mehrere Datensätze einer anderen Tabelle zuordnen. Die Zahl der zugeordneten Datensätze ist hierbei variabel ($0 \dots \infty$). Im Beispiel kann der Kunde ‚Franz‘ mehrere Bücher ausleihen bzw. das Buch ‚Micky Maus‘ existiert in mehreren Exemplaren, sodass es von verschiedenen Kunden gleichzeitig ausgeliehen werden kann. Zwischen der Tabelle Kunden und der Tabelle Leihvorgänge besteht eine 1:n Beziehung ebenso zwischen Bücher und Leihvorgänge.

- n:m Verknüpfung

Theoretisch wäre es denkbar mehrere Datensätze einer Tabelle einer unterschiedlichen Menge an Datensätzen einer anderen Tabelle zuzuordnen. Im obigen Beispiel besteht zwischen der Tabelle Kunden und der Tabelle Bücher eine n:m-Beziehung. Mehrere **n** Kunden können unterschiedlich viele **m** Bücher ausleihen. In realen Datenbanksystemen ist diese Verknüpfungsart jedoch nicht realisierbar und würde nur zu der oben beschriebenen Redundanz führen. In der Praxis wird daher eine Zwischentabelle (hier z.B. Leihvorgänge) erzeugt und die n:m-Beziehung in zwei 1:n Beziehungen umgewandelt.

Die folgende Abbildung des (geringfügig erweiterten) Beispiels zeigt die Zusammenhänge verschiedener verknüpfter Tabellen.

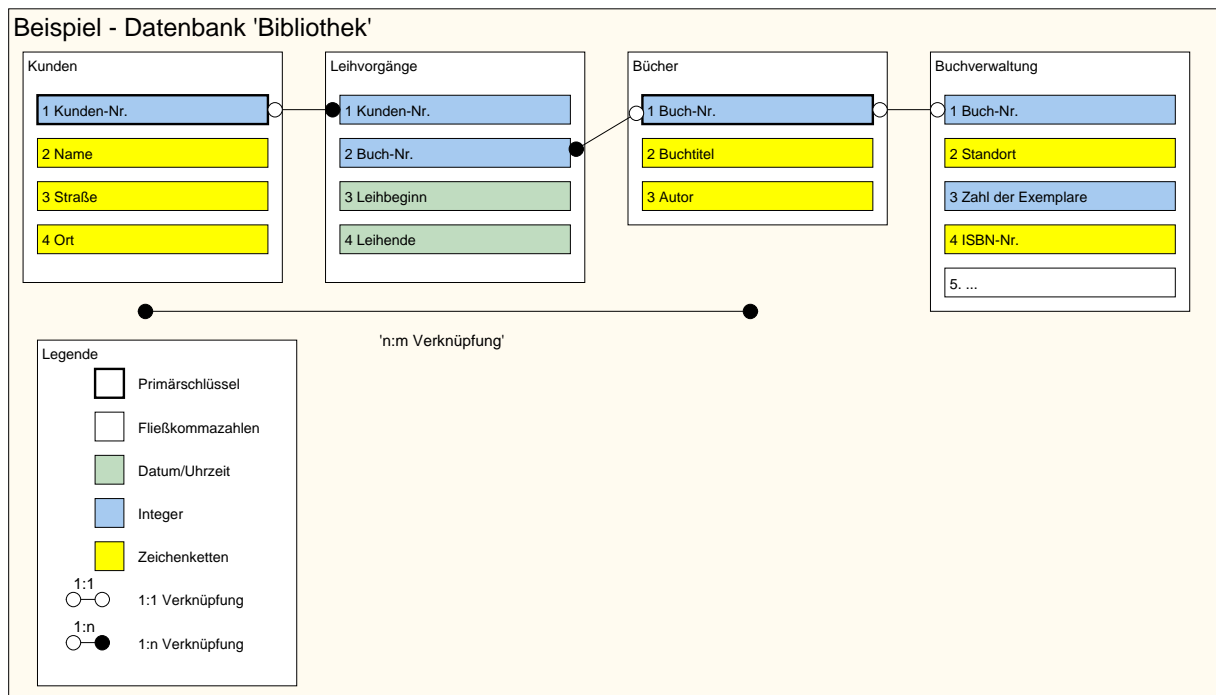


Abbildung 2.3: Struktur der Beispieldatenbank

Integritätsregeln sind notwendig, um die Konsistenz der Daten zwischen verknüpften Tabellen zu wahren. Diese so genannte referenzielle Integrität verhindert, dass ungültige Daten in die Tabellen eingegeben werden können.

Für die Verbuchung von Leihvorgängen wäre es zum Beispiel wenig nützlich, wenn Bücher an Kunden verliehen würden, die nicht in der Kundenliste vermerkt sind. Auch das Verleihen von Büchern, die noch nicht in der Büchertabelle vermerkt sind, wäre wenig sinnvoll. Neben diesen systeminhärenten Integritätsregeln können bei den meisten Datenbankprodukten noch zusätzliche Regeln (Constraints) definiert werden, die die Eingabe ungültiger Werte in die Tabellenspalten verhindern.

Neben der vom Benutzer angelegten Datenstruktur gibt es in der Regel in einer Datenbank noch zusätzliche Verwaltungsstrukturen: so genannte Metadaten. Die häufig in speziellen Systemtabellen niedergelegten Metadaten beschreiben die Aufbau und Struktur der Benutzerdaten. Häufig enthalten sie auch wichtige Systemdaten, wie Angaben zum Zeichensatz, die Dimensionen, Datentypen, und die Eingaberegeln für jeweiligen Tabellenspalten, Nutzer-Zugriffsrechte und vieles mehr.

Zur Abfrage und Manipulation der Daten und Metadaten einer Datenbank wurde für relationale Datenbanksysteme die Sprache SQL entwickelt.

2.1.2 Struktur und Funktionen der Abfragesprache SQL

SQL (Structured Query Language) gliedert sich in drei Teilbereiche: DDL Data Definition Language, DML Data Manipulation Language und DCL Data Control Language.

DDL-Befehle erlauben die Definition, Bearbeitung und Löschung von Strukturen, die die Daten enthalten. Zu diesen Strukturen gehören:

- Datenbanken (Database)
- Tabellen (Table)
- Sichten (View)
- Stored Procedures und Trigger

Stored Procedures und Trigger dienen der automatischen, ereignisgesteuerten Ausführung von SQL-Befehlen. Stored Procedures sind hierbei einfache Programme, die in selbst in der Datenbank gespeichert sind und die bedingungsgesteuerte Ausführung von Befehlen erlauben. Sie können nach Ausführung einen Rückgabewert an das aufrufende Programm zurückliefern. Trigger erlauben es Programme ereignisgesteuert auszuführen. Zu den durch Trigger ausgelösten Ereignissen gehören, das Hinzufügen und Ändern von Datensätzen, Anlegen und Ändern von Tabellen und anderen Objekten.

Typische Befehle sind CREATE, ALTER und DROP. Diese Befehle ändern die Struktur der Datenbank durch Änderung der oben bereits erwähnten Metadaten.

DML-Befehle bearbeiten die Daten innerhalb dieser Strukturen. Die vier fundamentalen Befehle sind: INSERT, UPDATE, DELETE und SELECT.

- INSERT dient zum Einfügen von Daten in eine Tabelle
- UPDATE erlaubt die Änderung existierender Daten
- DELETE löscht Daten
- SELECT erlaubt das Abfragen und Finden von Informationen in der Datenbank.

Dieser Befehl ist vermutlich der Wichtigste, Mächtigste – und Komplizierteste – aller SQL-Befehle. Er erlaubt Informationen wiederzufinden, die vorher gespeichert wurden. Mit diesem Befehl können die Daten verschiedener Tabellen der Datenbank gesammelt und ausgewertet werden. Dies erlaubt eine von der tatsächlichen Tabellenstruktur unabhängige Sicht auf die Datenbankdaten. Mit einem DDL-Befehl (CREATE VIEW *Sichtname*) können darüberhinaus häufig benutzte Abfragebefehle und Strategien als so genannte *Sichten* (vgl. oben) in der Datenbank gespeichert werden. Auf diese Art kann durch einen einfachen SELECT-Befehl (SELECT * FROM *Sichtname*) ein weitaus komplexerer SELECT-Befehl abgekürzt werden.

DCL Befehle schließlich erlauben die Verwaltung von Zugriffsrechten (Role) auf den Objekten (Tabellen, Views, ...) der Datenbank.

Ein relationales Datenbankmanagementsystem (RDBMS) erlaubt die Verwaltung und den Zugriff auf Datenbanken, die nach dem relationalen Modell konzipiert werden. Das RDBMS ist meist als Serverprozesssystem realisiert, das die Datenbanken in Form großer Dateien auf der Festplatte verwaltet. Häufig verwenden die RDBM-Systeme betriebssystemunabhängige, hardwarenahe Methoden, um auf diese Dateien zuzugreifen. Beim Microsoft SQLServer

werden diese Dateien z.B. als Medien bezeichnet, in denen die eigentlichen Datenbanken zunächst angelegt werden müssen.

Der Benutzerzugriff auf das Datenbanksystem erfolgt meist über ein Netzwerk mittels eines herstellerspezifischen Anwendungsprotokolles. Damit der Programmierer Befehle zum Serversystem übermitteln kann, liefert der Hersteller in der Regel eine Softwarebibliothek aus, deren Funktionen den Zugriff auf die RDBMS-Funktionen von einer Programmiersprache ausgehend ermöglichen.

Neben der Programmierung von Datenbankanwendungen mittels dieser herstellerspezifischen Schnittstellen gibt es auch einige herstellerunabhängige und sogar plattformunabhängige Schnittstellen (Java) für die Programmierung von Datenbankanwendungen.

2.1.3 Datenbankschnittstellen

Der klassische Weg mit einem relationalen Datenbanksystem zu kommunizieren, ist die Verwendung einer datenbankspezifischen Clientanwendung. Mit dieser Anwendung können über eine Kommandozeilen-Eingabe ein oder mehrere SQL-Befehle an das RDBMS geschickt werden. Zur Kommunikation mit dem Datenbank-Server über ein Netzwerk kommt ein spezielles herstellerspezifisches Anwendungsprotokoll zum Einsatz.

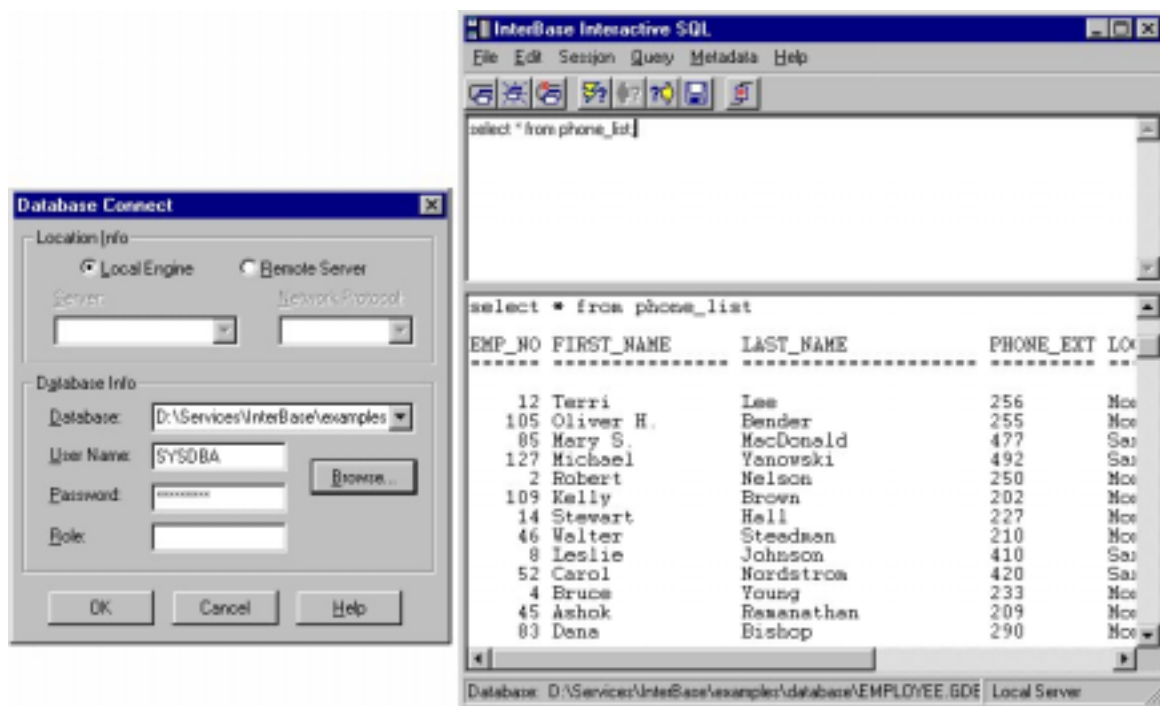


Abbildung 2.4: Client-Anwendung (Interbase-Server 5.5, Inprise)

Um die Erstellung von Fremdprodukten zu ermöglichen, liegen den meisten Datenbankprodukten Softwarebibliotheken für verschiedene Programmier- und Skriptsprachen bei, die die Entwicklung eigener Anwendungen und damit die Manipulation der Datenbankdaten über Formularfenster und ähnliche grafische Ein- und Ausgabemittel ermöglichen. Diese Anwendungen sind allerdings auf Grund der proprietären Methoden des Zugriffs auf das spezielle Datenbankprodukt beschränkt. Um dieses Dilemma zu beheben wurde die ODBC-Schnittstelle als standardisierte Schnittstelle entworfen.

ODBC ist keine datenbankspezifische Schnittstelle eines Herstellers sondern integraler Bestandteil des Betriebssystems. ODBC-Implementierungen (Open Database Connectivity) gibt es allerdings mit wenigen Ausnahmen nur für die Microsoft Betriebssystemplattformen. Ein so genannter ODBC-Treiber setzt in der Regel auf einer herstellerspezifischen Softwarebibliothek auf und stellt dem Programmierer ein einheitliches API (Application Programm Interface), eben die ODBC-API, zur Verfügung.

Ein wichtiges Merkmal der ODBC-Schnittstelle sind so genannte Datenquellen: Darunter versteht man Parametersätze, die wichtige Verbindungsinformationen zum Datenbankserver für eine Anwendung bereithalten.

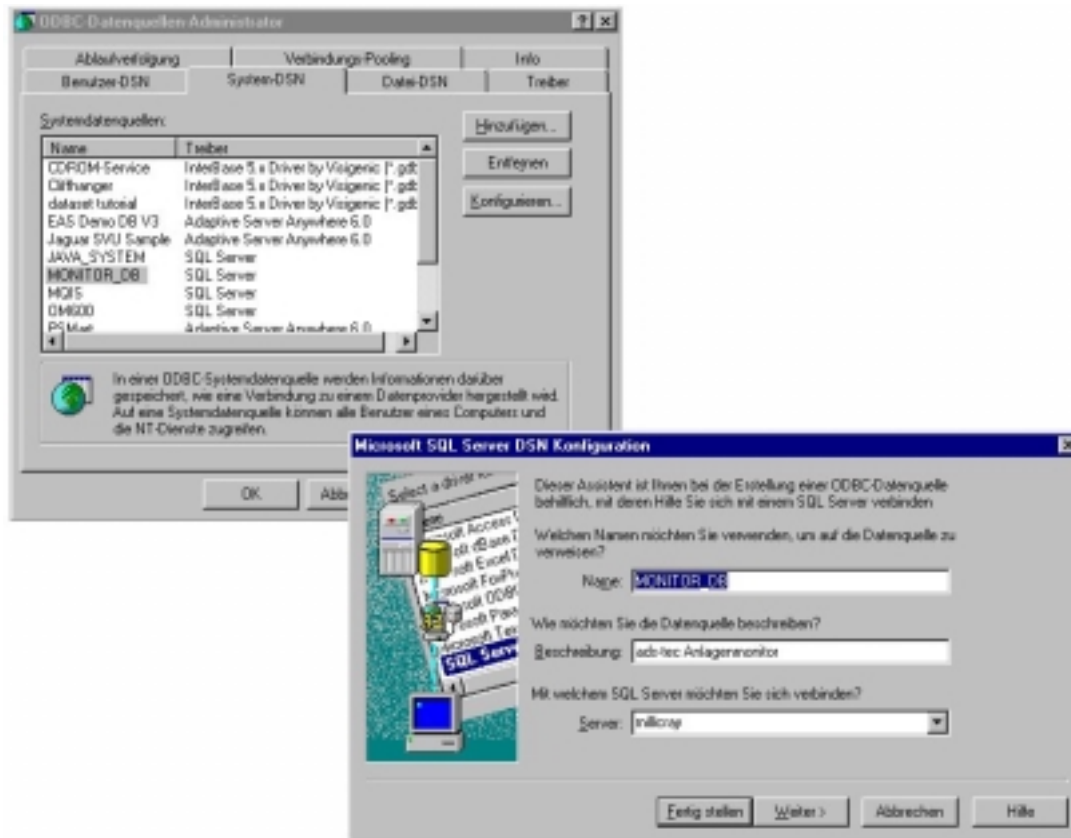


Abbildung 2.5: ODBC-Datenquelleneinrichtung

Diese Parametersätze müssen, wie in der obigen Abbildung gezeigt, zunächst einmal definiert bzw. konfiguriert werden, damit die gewünschte Datenbank genutzt werden kann.

Durch die fehlende Implementierung auf den meisten Betriebssystemplattformen ist man bei der Verwendung von ODBC auf Microsoft Windows beschränkt.

Eine ähnliche Datenbank-Schnittstelle ist JDBC, die in die plattformunabhängige Sprache Java integriert ist.

2.2 Java - Datenbankfunktionen

Java ist eine plattformunabhängige Programmiersprache und Laufzeitumgebung. Die Integration von Datenbankfunktionen in die Standardbibliotheken des Programmsystems stellt eine elegante Methode der Programmierung von Datenbankanwendungen dar.

2.2.1 Java Klassen und Funktionen

Eine Anwendung hat grundsätzlich folgende Schritte durchzuführen, um Daten von einer Datenbank abzufragen:

1. Aufbau der Verbindung zur Datenbank
2. Erzeugen einer SQL-Befehlszeichenkette
3. Ausführung des SQL-Befehls (Unter Anwendung von Java-JDBC-Objekten und deren Methoden.)
4. Auswertung des Resultates
5. Trennung der Verbindung zur Datenbank

Java-JDBC bietet 6 Klassen, 16 Schnittstellen und 4 Exceptions an, um diese Funktionen und deren Fehlerbehandlung zu verwirklichen. Die für eine einfache Abfrage wesentlichen Klassen sind:

- ***DriverManager***
DriverManager ist eine Unterklasse von *Object*, die die Verwaltung von JDBC-Datenbanktreibern übernimmt. Insbesondere ist hierbei eine Methode zu erwähnen, die für das Laden der Treiber zuständig ist.
- ***Connection***
Die Schnittstellenklasse *Connection* kapselt die Methoden zur Verbindungsverwaltung, also den Auf- und Abbau von Verbindungen.
- ***Statement***
Statement ist eine Schnittstellenklasse, deren Methoden für die Übermittlung von SQL-Befehlen zum Datenbank-Server benötigt werden.
- ***ResultSet***
Diese Schnittstellenklasse enthält Methoden zum Zugriff auf die Ergebnisse einer Datenbank-Abfrage.

Das nachfolgend abgedruckte Codebeispiel demonstriert die Anwendung dieser Klassen.

```
// Beispiel Datenbank-Abfrage in Java
```

```
import java.sql.*;

public class Kunden {
    public static void main(String[] arguments) {
        String data = "jdbc:odbc:Kunden";
        try {
            // JDBC-Datenbanktreiber laden
            // Anmerkung: das Beispiel benutzt den JDBC-ODBC-Bridgetreiber,
            // der dem JDK beiliegt.
            this.Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            // 1. Schritt Verbindungsaufbau
            Connection c = DriverManager.getConnection(data, "", "");
            // Statementhandle erzeugen
            Statement s = c.createStatement();
        }
    }
}
```

```
// 2. Schritt SQL-Befehlszeichenkette erzeugen
String sqlString = new String(
    "SELECT KUNDENNR, NAME, STRASSE, ORT FROM KUNDEN "
);
// Abfrage durchfuehren (3. Schritt)
ResultSet r = s.executeQuery(sqlString);
// 4. Auswertung
while(r.next()) {
    System.out.println(r.getInt("KUNDENNR") + ", "
        + r.getString("NAME") + ", "
        + r.getString("STRASSE") + ", "
        + r.getString("ORT") + ".");
}
s.close(); // Statementhandle freigeben (5. Schritt)
c.close(); // Verbindung beenden
} catch (Exception e) {
    System.out.println("Fehler -- " + e.toString());
    e.printStackTrace();
}
}
```

Codebeispiel 2.1: Java-Datenbankfunktionen

Die Methode `Statement.executeQuery()` wird für typische Datenbankabfragen verwendet. Ändert sich der Datenbankinhalt, z.B. durch die Anwendung von SQL-DDL-Befehlen, aber auch von INSERT, UPDATE und DELETE kommt die Methode `Statement.executeUpdate()` zum Einsatz.

Das Ergebnis einer Abfrage wird in einem `ResultSet`-Objekt festgehalten. `ResultSet` kann man sich als virtuelle Tabelle vorstellen, deren Datensätze man u.a. mit der Methode `ResultSet.next()` abrufen kann. Mit `ResultSet.next()` bewegt man einen Zeiger (Cursor) über die Datensätze der Tabelle. Für den Zugriff auf die Werte des Datensatzes besitzt `ResultSet` die Methoden `getString()`, `getInt()`,... usw. (Literatur: [1], [2])

2.2.2 Treiber und Schnittstellen

Zur Anwendung von JDBC ist die Verfügbarkeit eines JDBC-Treibers für das eingesetzte RDBMS notwendig. Erfreulicherweise haben die meisten führenden Datenbankhersteller die Bedeutung von JDBC erkannt und stellen Treiber zur Verfügung.

Man unterscheidet 4 Kategorien von Treibern:

1. JDBC-ODBC-Bridge-Treiber

stellt den Datenbankzugriff der JDBC-API über die ODBC-Schnittstelle zur Verfügung. Er ist allerdings nur auf Betriebssystemen einsetzbar, für die es eine ODBC-Implementierung und ODBC-Treiber für das betreffende DBMS gibt. Zurzeit sind dies lediglich Microsoft Betriebssysteme sowie das Betriebssystem Solaris von Sun. Eine weitere Einschränkung des Treibers entsteht beim Einsatz von Java-Applets: Die Datenquellen, die bei ODBC zum Einsatz kommen, sind Betriebssystemressourcen. Das Sicherheitsmodell von Java erlaubt es Applets nicht, auf Betriebssystemressourcen zuzugreifen, sodass hier der Einsatz des Treibers scheitert. Auf einem System, auf dem der Treiber zum Einsatz kommen soll, müssen zunächst ODBC konfiguriert und Datenquellen eingerichtet werden müssen. Diese unelegante Methode ist leider manchmal

der einzige Weg um Java den Zugriff auf ein RDBMS zu gewähren, dessen Hersteller sich gegen eine Unterstützung von JDBC entschieden hat.

2. Treiber, die JDBC-API-Funktionen auf native RDBMS-API-Funktionen abbilden.

Da dies über die Java Native Interface Schnittstelle geschieht, muss wie bei ODBC zunächst ein betriebssystemspezifischer Treiber installiert werden. Der Unterschied zu JDBC-ODBC liegt lediglich darin, dass es sich um einen datenbankherstellerspezifischen Brückentreiber handelt, der zum Einsatz kommt.

3. Treiber, die JDBC-Funktionen zunächst auf ein DBMS unabhängiges Netzwerkprotokoll abbilden.

Ein zusätzlicher Serverprozess (eine Art von Proxyserver) setzt dieses Netzwerkprotokoll in DBMS-API-Funktionen um. Es entsteht eine im folgenden Kapitel näher erläuterte Three-Tier-Architektur. Dieses Verfahren erfordert für den Einsatz im Internet besondere Anforderungen an das Netzwerkprotokoll im Hinblick auf Sicherheit und Behandlung von Gatewaytechnologien und Firewalls.

4. Treiber, die JDBC-Funktionen auf ein DBMS abhängiges Netzwerkprotokoll abbilden.

Dies erlaubt den direkten Aufruf und Ausführung von Datenbankfunktionen durch JDBC. Da die DBMS-spezifischen Netzwerkprotokolle häufig proprietäre Lösungen des Datenbankherstellers darstellen, wird diese Lösung von den führenden Herstellern am häufigsten angewendet.

2.3 CORBA

CORBA (Common Object Request Broker Architecture) ist eine Technologie, die die Verteilung von Anwendungskomponenten in einer so genannten verteilten Anwendungsarchitektur durch standardisierte Verfahren bei der Definition von Schnittstellen zwischen Anwendungskomponenten und deren Kommunikation unterstützt.

CORBA ist nicht im eigentlichen Sinne ein Standard sondern beruht auf Vereinbarungen führender Softwarehersteller, die sich in einer Vereinigung, der OMG (Object Management Group, <http://www.omg.org>) zusammengeschlossen haben.

Der klassische Weg der Kommunikation von Anwendungen in einem Netzwerk ist das Client-Server-Modell. Eine Anwendung, der so genannte Server stellt Dienste im Netzwerk zur Verfügung; Eine weitere Anwendung, der so genannte Client, nutzt diese Dienste.

CORBA erlaubt Anwendungen gleichzeitig sowohl Dienste zur Verfügung zu stellen als auch zu nutzen.

Besondere Bedeutung hat CORBA beim Aufbau von Multi-tier-Anwendungsarchitektursystemen erlangt.

2.3.1 Architektur verteilter Anwendungen

Eine Anwendungsarchitektur bestimmt auf welche Art und Weise eine Anwendung erstellt wird und wie deren Komponenten in den Systemen verteilt werden. Die meisten Anwendungsprogramme setzen sich aus 3 Grundtypen von Anwendungskomponenten zusammen:

- Eine **Präsentationskomponente** enthält die Funktionalität, mit deren Hilfe dem Benutzer Informationen des Anwendungssystems präsentiert werden. Weiterhin erlaubt sie in der Regel dem Benutzer Eingaben an das System zu richten und auf diese Weise Informationen in das System einzubringen oder gezielt abzurufen.
- Eine **Anwendungslogik** steuert die von der Anwendung ausgeführten Geschäftsfunktionen und Prozesse. Diese Funktionen und Prozesse werden in Abhängigkeit vom Benutzerverhalten entweder von einer anderen Anwendungsfunktion oder von einer Präsentationskomponente aufgerufen.
- Die **Datenzugriffskomponente** schließlich enthält die Schnittstelle zu Datenspeicherungssystemen wie Dateisystemen oder Datenbanken, beziehungsweise zu einer anderen Art externer Datenquelle oder einem anderen Anwendungssystem.

Einzel-Anwendungen (One-tier)

In traditionellen EDV-Systemen werden Anwendungen nicht in Ihre Grundkomponenten zerlegt, wodurch es oft schwierig ist, die Komponentenlogik nachzuvollziehen. Alle 3 der vorgestellten Ebenen sind in einem einzigen Programm zusammengefasst, das auf einem einzelnen Rechner ausgeführt wird, auf dem sich auch die zugehörigen Datendateien befinden. Die Anwendung kann grundsätzlich nur von einem Benutzer gleichzeitig benutzt werden.

Client/Server-Anwendungen (Two-tier)

Die Two-tier oder Client/Server-Architektur trennt eine Anwendung in zwei Teile und verteilt die Bearbeitung auf ein Arbeitsplatzsystem und einen Server. Die ressourcenlastigen Rechen- und Anwendungsprozesse werden auf den Server verlagert. Die Anwendungssteuerung wird auf dem Arbeitsplatzsystem durch eine Client-Anwendung durchgeführt.

Durch die Aufteilung des Gesamtsystems auf zwei Teile wird auch die Anwendungsentwicklung unterstützt, da bei Änderungen nicht unbedingt das gesamte Anwendungssystem ersetzt werden muss.

Multi-tier-Anwendungen (Three-tier und n-tier)

Der Begriff ‚n-tier‘ bezieht sich auf die logischen Komponenten einer Anwendung, aber nicht auf die Anzahl der von einer verteilten Anwendung genutzten Rechner. Die Zählung n bezeichnet hierbei die Zahl der Komponenten in einer Zugriffskette. Mehrere Instanzen einer Komponente (z.B. mehrere Clients (WWW-Browser)) oder unterschiedlich implementierte parallel (entweder/oder) installierte Komponenten ändern nicht die Anzahl der ‚tiers‘. Obwohl zwischen den verschiedenen Komponenten Schnittstellen definiert sind und diese in der Regel in einer Kette angeordnet sind, wäre es falsch von einem Schichtenmodell zu sprechen, da in einer Multi-tier-Umgebung jede Komponente durch den Zugriff auf ihre wohldefinierte Schnittstelle auch weitgehend unabhängig genutzt werden kann.

Wichtige Eigenschaften eines solchen Architekturmodelles sind:

- Innerhalb einer Anwendung können beliebig viele Komponenten jeden Typs vorhanden sein.
- Anwendungskomponenten können von beliebig vielen Anwendungssystemen gemeinsam genutzt werden.

- Jede Komponente kann mit dem für die Aufgabe an besten geeigneten Werkzeug entwickelt werden.
- Die Komponenten können als Prozesse auf einer Servermaschine oder über mehrere Anwendungs- und Servermaschinen verteilt werden.
- Die Komponenten kommunizieren untereinander über abstrakte Schnittstellen, die die jeweils zu Grunde liegende von der Komponente ausgeführte Funktion verbirgt.

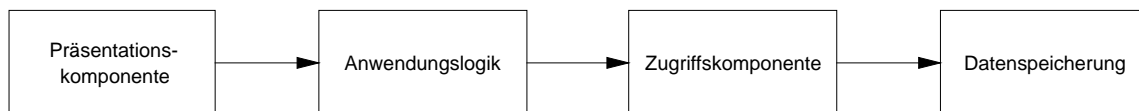


Abbildung 2.6: Beispiel für eine 4-tier-Architektur

Besonders häufig wird ein Three-tier-Anwendungsmodell implementiert. Das Three-tier-Modell trennt, wie oben beschrieben, die Komponenten Präsentation, Anwendungslogik und Datenzugriff. Beliebtheit hat dieses Modell durch den zunehmenden Einsatz von Datenbanksystemen erlangt, erlaubt es doch Systeme mit einem in der Praxis angemessenen Wartungsaufwand zu erstellen.

Weshalb ist gerade CORBA prädestiniert für die Entwicklung verteilter Anwendungen nach einem n-tier Modell?

Verteilte Anwendungssysteme basieren auf der Definition von Schnittstellen zwischen Komponenten und auf dem Vorhandensein verschiedener Dienste, die für eine Anwendung zur Verfügung stehen. So ist z.B. ein Dienst notwendig, um die Komponenten und die durch sie zur Verfügung gestellten Dienste im verteilten System zu finden (Nameservices). CORBA stellt einen Standardmechanismus zum Definieren der Schnittstellen zwischen Komponenten sowie einige Werkzeuge zur Verfügung, die das Implementieren dieser Schnittstellen in fast beliebigen vom Programmierer gewählten Sprachen erleichtern. Schließlich stellt CORBA noch die 'Leitungen' zur Verfügung, die es den verschiedenen Komponenten einer Anwendung oder verschiedener Anwendungen ermöglicht, miteinander zu kommunizieren.

Weitere wichtige Merkmale von CORBA sind Plattformunabhängigkeit und Sprachenunabhängigkeit. Plattformunabhängigkeit bedeutet, dass CORBA-Objekte auf jeder Plattform verwendet werden können, für die eine CORBA-Implementierung vorhanden ist. Sprachenunabhängigkeit bedeutet, dass CORBA-Objekte und -Clients in quasi jeder Programmiersprache implementiert werden können. Ferner ist es für CORBA-Objekte nicht relevant, in welcher Sprache andere CORBA-Objekte implementiert wurden, mit denen sie kommunizieren.

2.3.2 CORBA-Architektur

Bei CORBA handelt es sich um eine objektorientierte Architektur. CORBA-Objekte weisen viele Merkmale auf, die auch in anderen objektorientierten Systemen zu finden sind, hierzu gehören Vererbung und Polymorphie. Interessant ist die Tatsache, dass diese Merkmale auch zur Verfügung stehen, wenn eine nicht objektorientierte Sprache wie C oder Cobol verwendet wird. Allerdings wird CORBA besonders gut durch objektorientierte Sprachen wie C++ und Java abgebildet.

Object Request Broker (ORB)

Einen grundlegenden Bestandteil von CORBA bildet der Object Request Broker oder kurz ORB. Unter einem ORB versteht man eine Software-Komponente, die die Kommunikation zwischen Objekten erleichtern soll. Hierzu werden eine Reihe von Leistungsmerkmalen bereitgestellt:

- Ein Merkmal besteht darin, ein Remote-Objekt aufzufinden, sofern eine Objektreferenz vorhanden ist.
- Ein anderer vom ORB zur Verfügung gestellter Dienst besteht im Übertragen des Formats von Parametern und Rückgabewerten (dies wird als ‚Marshaling‘ bezeichnet), die bei Remote-Methodenaufrufen gesendet bzw. empfangen werden.

Die Hauptaufgabe des ORB ist es Objektreferenzen zur Verfügung zu stellen. Ein Objekt verfügt in der Regel über so genannte Methoden, die die Eigenschaften und das Verhalten des Objektes definieren. Ein Objekt beschreibt über seine Methoden einen Dienst, den es anderen Objekten zur Verfügung stellen kann. Wenn eine Anwendungskomponente einen Dienst nutzen möchte, der von einer anderen Komponente zur Verfügung gestellt wird, muss sie zunächst eine Objektreferenz für das Objekt erlangen, welches den Dienst zur Verfügung stellt. Nachdem die Objektreferenz erlangt wurde, kann die Komponente Methoden des Objekts aufrufen und damit auf die vom Objekt zur Verfügung gestellten Dienste zugreifen.

In der Regel übernehmen diese Methoden Parameter als Eingabe und geben andere Parameter als Ausgabe zurück. Der ORB empfängt nun die Eingabeparameter von der Komponente, welche die Methode aufruft, und überträgt diese Parameter in ein Format, das über das Netzwerk zum Remote-Objekt übertragen werden kann. Dieses Format wird auch als Übertragungsformat bezeichnet. Der ORB nimmt auch die umgekehrte Formatübertragung (Unmarshaling) der zurückgegebenen Parameter vor; diese werden also aus dem Übertragungsformat in ein Format umgesetzt, das die aufrufende Komponente versteht.

Die gesamte Formatübertragung erfolgt ohne Eingriff des Programmierers. Die Client-Anwendung ruft einfach die gewünschte Remote-Methode auf, die für den Client so aussieht, als wäre sie eine lokale Methode, und es wird ein Ergebnis zurückgegeben. Der gesamte Prozess, also die Formatübertragung der Eingabeparameter, das Auslösen des Methodenaufrufs auf dem Server und die umgekehrte Formatübertragung der Rückgabeparameter werden automatisch und transparent durch den ORB durchgeführt.

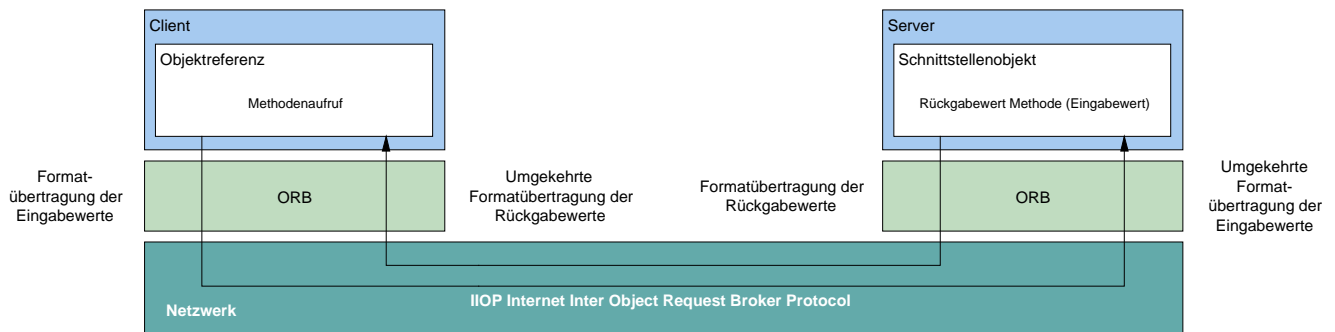


Abbildung 2.7: Arbeitsweise des ORB

Eine wesentliche Eigenschaft der Formatübertragung besteht darin, dass die Kommunikation zwischen den Komponenten plattformunabhängig stattfindet, weil die Parameter für die Übertragung in ein plattformunabhängiges Format und beim Empfang wieder in ein plattformspezifisches Format konvertiert werden (das Übertragungsformat ist Bestandteil der CORBA-Spezifikation). Dies bedeutet, dass ein Client, der beispielsweise auf einem Windows-System läuft, Methoden auf einem Server aufrufen kann, der auf einem Unix-System läuft.

Zusätzlich zur Unabhängigkeit vom verwendeten Betriebssystem werden auch Unterschiede in der Hardware (wie die Reihenfolge der Prozessor-Bytes oder die Art der Adressierung) irrelevant, weil der ORB automatisch die notwendigen Umwandlungen vornimmt. Im Wesentlichen fängt der ORB die Plattform-Unterschiede ab, unabhängig davon, ob es sich um das Betriebssystem, die Art der Adressierung, die Wortlänge usw. handelt.

Interface Definition Language (IDL)

Ein weiterer zentraler Bestandteil der Architektur von CORBA ist die darin verwendete Interface Definition Language (IDL). Die IDL dient zur Definition von Schnittstellen zwischen CORBA-Objekten und ist das Instrument, mit dem die Sprachenunabhängigkeit von CORBA sichergestellt wird. Da die mit der IDL beschriebenen Schnittstellen auf jede beliebige Programmiersprache abgebildet werden können, sind die mit CORBA entwickelten Anwendungen und Komponenten unabhängig von der bzw. den Sprache(n), die zur Implementierung verwendet wurde(n). Ein in C++ geschriebener Client kann mit einem in Java geschriebenen Server kommunizieren, dieser wiederum kann mit einem anderen Server kommunizieren, der z.B. in Cobol geschrieben wurde usw.

Ein wichtiger Aspekt bei der IDL ist die Tatsache, dass es sich nicht um eine Implementierungssprache handelt. Dies bedeutet, dass man in dieser Sprache keine Anwendungen schreiben kann. Die IDL hat nur den Zweck, Schnittstellen zu definieren, die Implementierungen für diese Schnittstellen erfolgen über eine andere Sprache. Die IDL-Spezifikation ist dafür verantwortlich, dass die Daten zwischen Anwendungen in unterschiedlichen Sprachen korrekt ausgetauscht werden.

Wird in IDL eine Variable mit dem Typ `long` definiert, sorgt eine Sprachabbildung (Language Mapping), die vom IDL-Compiler durchgeführt wird, dass diese Variable in C++ mit dem Datentyp `long` und in Java mit den Typ `int` dargestellt wird.

Die OMG hat eine Anzahl von Standard-Sprachabbildungen für viele verbreitete Sprachen wie C, C++, Cobol, Java und Smalltalk definiert. Für weitere Sprache existieren ebenfalls Sprachabbildungen. Diese sind jedoch (noch) nicht standardisiert.

Da beim Arbeiten mit CORBA keine bestimmte Sprache verwendet werden muss, haben die Anwendungsentwickler die Möglichkeit, jeweils diejenige Sprache zu wählen, die für die betreffende Anwendung am besten geeignet ist. Der Entwickler kann sogar mehrere unterschiedliche Sprachen für die einzelnen Komponenten einer Anwendung wählen. CORBA ermöglicht die Kommunikation zwischen diesen ansonsten sehr unterschiedlichen Komponenten.

Das Kommunikationsmodell von CORBA

In CORBA werden Objektreferenzen verwendet (diese werden im CORBA/IIOP-Sprachgebrauch als IORs (Interoperable Object References, Interoperable Objektreferenzen) bezeichnet), um die Kommunikation zwischen den Objekten zu erleichtern. Wenn eine Komponente einer Anwendung auf ein CORBA-Objekt zugreifen möchte, erhält es zunächst eine IOR für das betreffende Objekt. Mit Hilfe der IOR kann die Komponente (die als Client des betreffenden Objekts bezeichnet wird) Methoden des Objekts aufrufen (dieses wird als Server bezeichnet).

In CORBA erfolgt die gesamte Kommunikation zwischen Objekten über Objektreferenzen. Die Sicht auf diese Objekte ist nur dadurch möglich, dass Referenzen an diese Objekte weitergegeben werden (Variablenparameter, pass by reference), die Weitergabe der Objekte kann nicht durch Werte (Werteparameter, pass by value) erfolgen (zumindest ist dies in der aktuellen Spezifikation von CORBA Version 2.3 der Fall). Die Tatsache, dass eine Weitergabe von Objekten nicht durch Werteparameter möglich ist, bedingt, dass Remote-Objekte in CORBA entfernt bleiben und es derzeit keine Möglichkeit für ein Objekt gibt, sich selbst an einen anderen Ort zu verschieben oder zu kopieren. Die Methoden des Remote-Objektes werden daher auf dem Server ausgeführt.

CORBA-ORBs verwenden für die Kommunikation untereinander das Internet-Inter-Object-Request-Broker-Protocol (IIOP). Das IIOP ist ein wichtiger Faktor bei der Kompatibilität von CORBA-Anwendungen.

Der CORBA Standard legt als allgemeines Protokoll das GIOP (General Inter Object Request Broker Protocol) für die Kommunikation zwischen ORB's fest. Zusätzlich zu diesem allgemeinen Protokoll GIOP legt der Standard weitere Protokolle fest, die das GIOP mit dem Protokoll der Transportschicht verbinden. Das GIOP existiert so in verschiedenen Anpassungen an Transportschichtprotokolle, wie TCP/IP (Transport Control Protocol / Internet Protocol), DCE (Distributed Computing Environment) und verschiedenen hersteller-spezifischen Protokollen.

Die am häufigsten verwendete Protokollanpassung ist das IIOP, das auf dem Transportprotokoll TCP/IP aufsetzt. Gemäß Corba-Standard Version 2.0 müssen Hersteller das IIOP implementieren, wenn ihr Produkt als CORBA-kompatibel bezeichnet werden soll. Durch diese Festlegung hat sich IIOP als das native Protokoll für CORBA-Anwendungen etabliert, sodass einige Hersteller dazu übergegangen sind, Ihren Produkten CORBA-kompatible ORB's hinzuzufügen.

So hat z.B. Netscape in den WWW-Browser und Serverprodukten einen ORB einer Fremdfirma (Visibroker von Inprise) integriert. In dem OpenSource-Projekt KDE, das sich die Entwicklung einer Benutzungsoberfläche für UNIX-Plattformen zur Aufgabe gemacht hat, wird ein CORBA-kompatibler ORB verwendet, um die in Microsoft Windows vorhandenen OLE-Funktionen für Verbunddokumente nachzubilden.

Das Objektmodell von CORBA

CORBA ist eine Architektur verteilter Objekte. Dabei unterscheidet sich das in CORBA definierte Objektmodell von dem der meisten objektorientierten Programmiersprachen. Die Hauptunterschiede liegen:

- in der Verteilung der Objekte,
- der Fehlerbehandlung beim Methodenaufruf,
- der Behandlung von Objektreferenzen und
- der Aktivierung von Objekten.

Für einen CORBA-Client sieht der Aufruf eines entfernten Objektes genauso aus, wie ein lokaler Methodenaufruf. Der Client nimmt zunächst nicht wahr, dass er Objekte verwendet, die in einem Netzwerk verteilt sind. Da aber die Verteilung von Objekten im Netzwerk eine höhere Fehlerwahrscheinlichkeit – bedingt durch Netzwerkstörungen und Serverausfälle - mit sich bringt, muss CORBA einen Mechanismus anbieten, um die Fehler behandeln zu können. Dies geschieht durch Exceptions. Jede Operation, die auf ein CORBA-Objekt angewendet wird kann eine CORBA-System-Exception auslösen, die einen Netzwerkfehler, die Nichtverfügbarkeit eines Servers, oder eine ähnliche Situation anzeigt. Mit Ausnahme dieser zusätzlichen durch CORBA-Objektmethode ausgelösten Exceptions ist jede Remote-Methode identisch mit dem lokalen Gegenstück.

In einer verteilten Anwendung gibt es grundsätzlich zwei Möglichkeiten für eine Anwendungskomponente, um auf ein Objekt in einem anderen Prozess zuzugreifen:

Übergabe durch Referenz (Referenzübergabe, pass by reference)

Ein Prozess A, Eigentümer eines Objektes übergibt hierbei eine Objektreferenz an einen Prozess B. Wenn Prozess B eine Methode des Objektes aufruft, wird diese Methode von Prozess A ausgeführt. Der Prozess B hat über die Objektreferenz nur eine Sicht auf das Objekt und kann damit Prozess A nur auffordern, Methoden für B auszuführen.

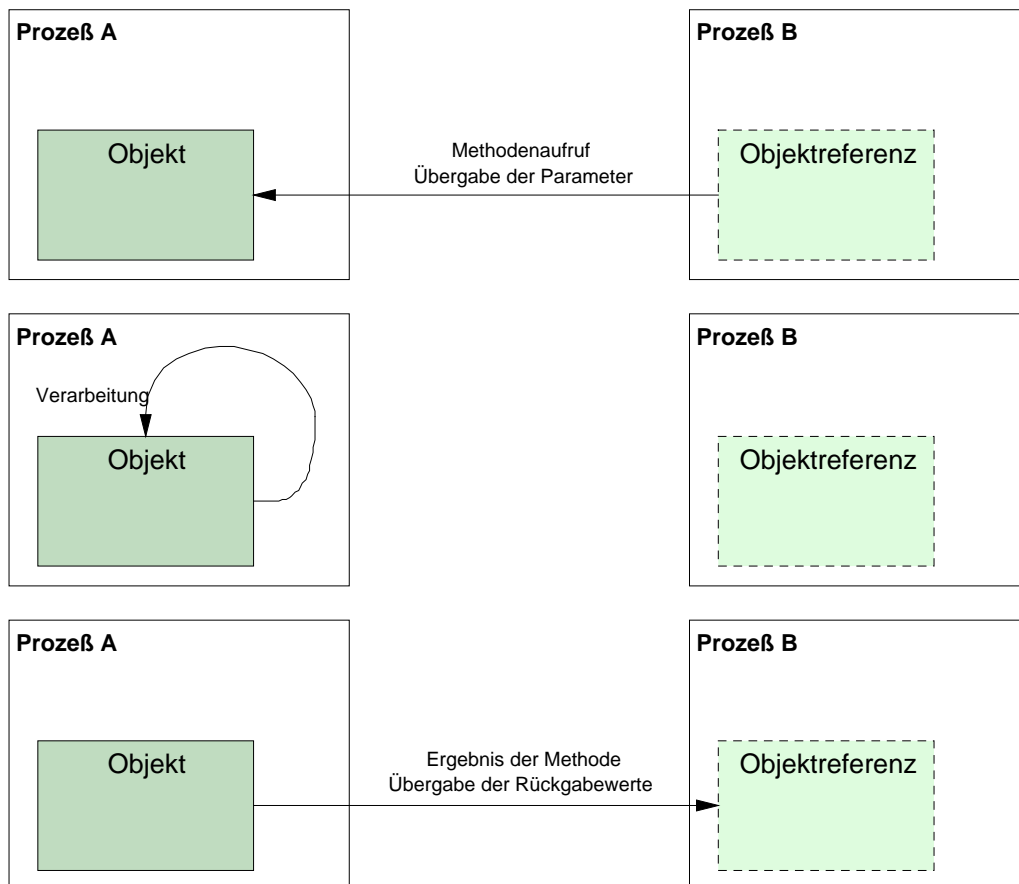


Abbildung 2.8: Übergabe durch Referenz (pass by reference)

Übergabe durch Werte (pass by value)

Bei diesem Verfahren bildet Prozess A eine Instanz eines Objektes mit allen für die Verwendung notwendigen Initialisierungen. Der Status des Objektes wird an die anfordernde Komponente weitergegeben, sodass die anfordernde Komponente quasi eine Kopie des Objektes anlegen kann. Diesen Vorgang der Weitergabe bezeichnet man als Serialisierung des Objektes. Wenn Prozess B eine Methode des Objektes aufruft, wird diese von Prozess B ausgeführt. Der Zustand der ursprünglich von Prozess A gebildeten Instanz des Objektes ändert sich hierbei nicht. Alle Operationen werden nur auf der Kopie des Objektes von Prozess B ausgeführt. Zusätzlich zu der Übergabe des Objektstatus, muss sichergestellt werden, dass die Komponente, die das Objekt empfängt, Implementierungen der von dem Objekt bereitgestellten Methoden kennt.

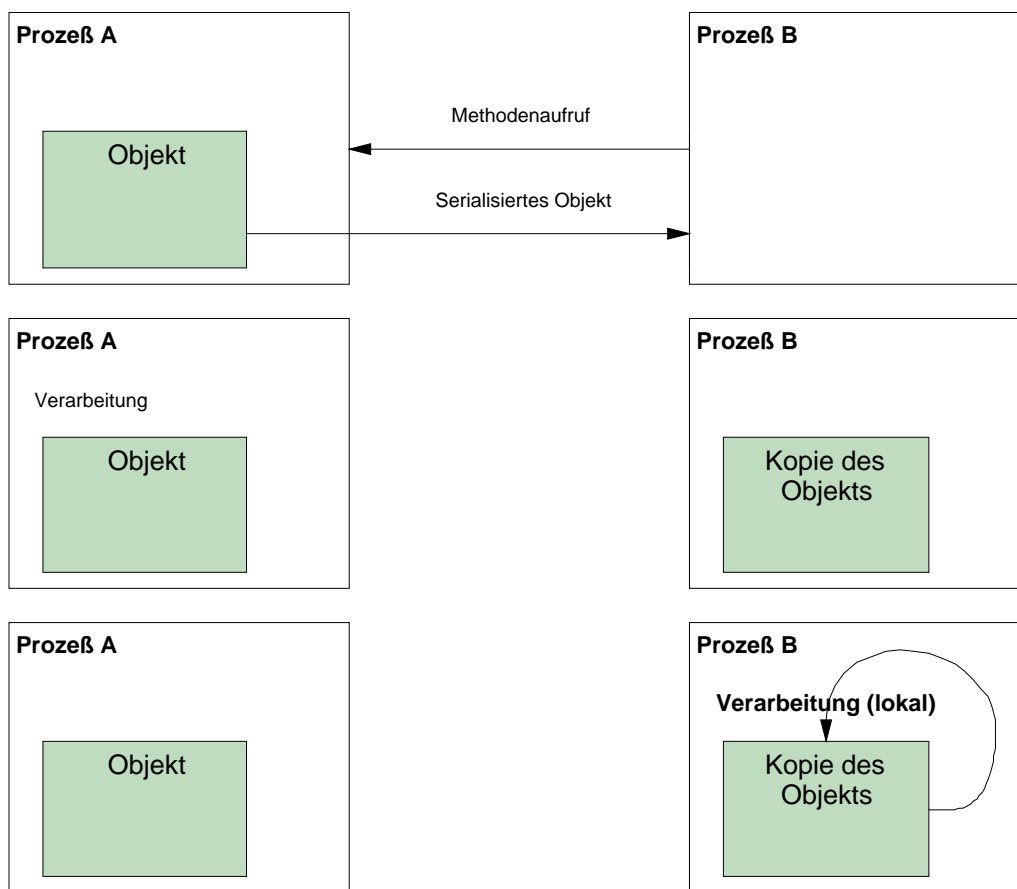


Abbildung 2.9: Übergabe durch Werte (pass by value)

Die Übergabe von Objekten durch Werte ist da von Vorteil, wo durch häufigen Aufruf von Methoden eines Objektes ein großer Systemaufwand und Netzbelastung entsteht. Schließlich müssen bei der Referenzübergabe Eingabe- und Rückgabewerte der Methoden für jeden Aufruf in ein für die Übertragung geeigneteres Format, das so genannte Übertragungsformat, umgewandelt werden.

Das Verfahren *Übergabe durch Werte* ist in den aktuellen CORBA-Implementierungen nach CORBA Version 2.3 noch nicht enthalten. Erst der neue CORBA-Standard Version 3.0 bietet

eine Semantik bei der Schnittstellendefinition an, die die Nutzung dieses Verfahrens für CORBA ermöglicht (Literatur [4], [8]).

Die Schnittstelle zwischen Objektimplementierung und ORB ist in CORBA der Objektadapter. Der Standard stellt verschiedene Gebrauchsmuster des Objektadapters zur Verfügung. Der wichtigste ist hiervon der Basic Object Adapter (BOA). Der BOA stellt CORBA eine allgemeine Gruppe von Methoden für den Zugriff auf ORB-Funktionen zur Verfügung. Hierzu gehören Funktionen zur Benutzeridentifikationsprüfung, zur Objektaktivierung und Persistenz. Insbesondere soll hier die Aktivierung und Deaktivierung von Objekten erläutert werden. Die Aktivierung von Objekten erfolgt nach verschiedenen Strategien:

- Gemeinsam verwendeter Server

Ein einziger Server wird von mehreren Objekten verwendet.

- Nicht gemeinsamer Server

Jedes Objekt wird mit einem eigenen Server betrieben.

- Server-pro-Methode

Bei jedem Methodenaufruf wird ein Server gestartet. Hat die Methode die Rückgabewerte übertragen, wird der Server automatisch beendet.

- Permanenter Server

Der Server wird ‚manuell‘ gestartet. Dies kann durch einen Benutzer an der Systemkonsole, durch einen Systemprozess (Daemon) oder einen anderen externen Agent erfolgen.

In einer herkömmlichen Client-Server-Anwendung stellt der Server die Komponente dar, die anderen Anwendungskomponenten, den Clients, Dienste zur Verfügung stellt. Die Architektur einer CORBA-Anwendung unterscheidet sich hiervon nicht. Wenn man allerdings einen einzelnen Aufruf einer fernen Methode betrachtet, so können die Rollen von Client und Server vorübergehend vertauscht werden. In CORBA wird jede Komponente, die eine Implementierung und damit Methoden für ein Objekt zur Verfügung stellt, als Server bezeichnet. Eine CORBA-Komponente kann allerdings während Sie von anderen Komponenten Dienste nutzt, wiederum anderen Komponenten Dienste zur Verfügung stellen. Sie fungiert als Client einer Komponente und als Server für alle übrigen Komponenten. Obwohl eine Komponente beide Aufgaben ausfüllen kann, wird Sie in der Regel als Client oder als Server bezeichnet. Dies ist damit zu begründen, dass es in der Regel Hauptaufgabe einer Anwendungskomponente A ist, die Dienste einer anderen Komponente B zu nutzen und die eigenen Objektmethoden nur zur Verfügung stellt, damit die Komponente B Initialisierungen der Objektinstanzen vornehmen kann oder zusätzliche Parameter für einen Dienst abfragen kann. Die Methoden der Client-Anwendungskomponente werden daher häufig auch als Callbackmethoden bezeichnet.

CORBA-Dienste

So wie CORBA ermöglicht eigene Dienste zu definieren und zu implementieren, gibt es im CORBA-Standard eine Reihe vordefinierter Schnittstellen von Diensten. Diese betreffen sowohl allgemeine Anwendungen, als auch spezialisierte branchenabhängige Dienste.

CORBAservices und CORBAfacilities:

CORBAservices: Allgemeine Dienste für Anwendungen:
Verzeichnisdienste, Objektpersistenzdienste, Ereignisbehandlung

CORBAfacilities: Branchenabhängige Dienste:
Finanzwesen, Gesundheitswesen und Telekommunikation.

Allen Diensten ist gemeinsam: Es existiert eine standardisierte Schnittstellendefinition. Dadurch, dass eine standardisierte Schnittstelle existiert, ist die Nutzung dieser Dienste von jeder Anwendungskomponente herstellerunabhängig möglich. Dienste sind ein Güte Merkmal und ein gewichtiges Verkaufsargument von CORBA-Produkten.

Ein wichtiger Dienst ist der so genannte Nameservice. Er stellt in einer hierarchischen Verzeichnisstruktur Bezeichner und Objektreferenzen für die Verwendung durch ORB's zur Verfügung. Die Anwendung eines Nameservice wird im folgenden Kapitel skizziert.

2.3.3 Erstellung einer CORBA-Anwendung in der Praxis

Anhand einer einfachen Anwendung soll die Entwicklung einer CORBA-Anwendung beschrieben werden. Der erste Schritt beim Erstellen einer CORBA-Anwendung, nachdem man sich über die zu implementierenden Komponenten und der von Ihnen zu leistenden Aufgaben und Dienste klar geworden ist, besteht in der Definition der Schnittstellen dieser Komponenten. Wie bereits geschildert erfolgt diese Definition in der implementierungsunabhängigen Sprache IDL. Im Beispiel soll dies anhand einer einfachen Anwendung demonstriert werden, die zwei Zahlen addiert. Die IDL-Datei enthält folgende Definitionen:

// IDL-Definitionsdatei für einen Dienst, der zwei Zahlen addiert.

```
module CORBATest
{
    interface math
    {
        double add
        (
            in double m1,
            in double m2
        );
    };
};
```

Codebeispiel 2.2: IDL-Definitionsdatei

Nach der Definition der Schnittstellen in IDL erfolgt die Erzeugung der sprachabhängigen Implementierungsdateien durch den IDL-Compiler. Dieser erzeugt so genannte Client-Stubs und Server-Skeletons. Der Client-Stub ist ein Quelltext, der einem Client eine CORBA-Serverschnittstelle zur Verfügung stellt. Das Server-Skeleton ist ein ebenso vom IDL-

Compiler erzeugter Quelltextabschnitt, der das Gerüst darstellt auf dem der Server-Implementierungsquelltext für eine bestimmte Schnittstelle erzeugt wird. Für welche Zielprogrammiersprache die Dateien erzeugt werden hängt vom verwendeten IDL-Compiler ab. Um den Server in C++ und den Client in Java zu erzeugen muss die IDL-Datei in der Regel mit zwei verschiedenen Compilern bearbeitet werden.

Die oben dargestellte Definition erzeugt so mit einen IDL-Compiler (im Beispiel das Produkt JavaIDL), der eine Java-Sprachabbildung beherrscht, die folgenden Dateien:

- `_mathImplBase.java`

Diese Klasse ist das Gerüst (Skeleton) für den Server. Die CORBA-Grundfunktionen werden hier auf die zu implementierende Serverklasse beim im Folgenden erläuterten Vererbungsansatz durch Vererbung übertragen.

- `_mathStub.java`

Diese Klasse stellt dem Client, die CORBA-Funktionen zur Verfügung. Es handelt sich hier um den so genannten Client-Stub.

- `math.java`

In dieser Datei ist bei Java die Schnittstelle definiert. Die Schnittstelle erbt die CORBA-Funktionalität von zwei weiteren Schnittstellen. Im Beispiel wird hier eine einzige Funktion `add()` definiert.

- `mathHelper.java`

In dieser Klasse sind weitere CORBA-Hilfsfunktionen enthalten. Wichtig ist diese Klasse insbesondere im Zusammenhang mit der Suche nach Objektreferenzen.

- `mathHolder.java`

Diese Klasse ist für die Formatübertragung der Methodenparameter und der Rückgabewerte zuständig.

Zu Implementierung des CORBA-Servers ist zunächst nur die Klasse `_mathImplBase` von Interesse. Von ihr leitet man durch Vererbung eine Klasse ab, die zur Implementierung, der durch die Schnittstelle definierten Dienste verwendet wird (`MathServant`). Die eigentliche Serverimplementierung muss lediglich von dieser Schnittstellenklasse eine Objektinstanz bilden und bei einem Nameservice registrieren. Der Nameservice ermöglicht der Clientanwendung das Auffinden der Objektreferenz.

```
package CORBATest;
```

```
import org.omg.CosNaming.*;
```

```
import org.omg.CosNaming.NamingContextPackage.*;
```

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

```
// Die Klasse MathServant erbt die CORBA-  
// Grundfunktionen von der Klasse _mathImplBase  
// Da _mathImplBase die Schnittstelle implementiert,  
// die bisher nur als Rumpf vorliegt wird die
```

```
// Schnittstellenmethode add() hier mit einer
// Definition überschrieben, die den gewünschten Dienst
// ausführt. Soweit die Theorie - in der Praxis ist
// _mathImplBase eine abstrakte Klasse, die keine echten
// Implementierungen bereitstellt. Die Implementierung muss
// daher durch MathServant erfolgen.

class MathServant extends _mathImplBase
{
    public double add(double m1, double m2)
    {
        return m1 + m2;
    }
}

// Die Klasse Server bildet eine Instanz des MathServant-
// Objektes, initialisiert und bindet die Instanz an den
// ORB und registriert zuletzt die Objektinstanz beim Nameservice

public class Server {

    public Server() {

    }

    public static void main(String args[])
    {
        try{
            // Erzeugt eine Instanz des ORB-Objektes
            // und initialisiert es
            ORB orb = ORB.init(args, null);

            // Erzeugen der Instanz und Binden an ORB
            MathServant helloRef = new MathServant();
            orb.connect(helloRef);

            // Initialisierung des Zugriffs auf Nameservice
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // Registrierung der Objekt Referenz beim Nameservice
            NameComponent nc = new NameComponent("Math", "");
            NameComponent path[] = {nc};
            ncRef.rebind(path, helloRef);

            // Endloswarteschleife
            java.lang.Object sync = new java.lang.Object();
            synchronized (sync) {
                sync.wait();
            }

        } catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Codebeispiel 2.2: CORBA-Server (Vererbungsansatz)

Dieses Vorgehen birgt allerdings beim Einsatz von Java als Implementierungssprache einen gewichtigen Nachteil: Es gibt in Java keine Mehrfachvererbung bei Klassen. Eine

Mehrfachvererbung ist nur bei Java-Schnittstellen vorhanden. Soll nun eine Schnittstellenklasse von einer beliebigen anderen Klasse abgeleitet werden, kann sie auf diesem Weg keine CORBA-Funktionalität erben. Daher gibt es bei der weiteren Implementierung grundsätzlich zwei Vorgehensweisen: den bereits dargestellten **Vererbungsansatz** und den so genannten **Delegationsansatz**.

Beim Aufruf des IDL-Compilers kann man in der Regel wählen, welchen Ansatz man wählt. Für den Delegationsansatz werden zusätzliche Klassen, die so genannten Tie-Klassen erzeugt. Angewendet auf das Beispiel ergeben sich so die zwei zusätzlichen Klassen:

- `_mathOperations.java`

Eine Java-Schnittstellendefinition, die die Methoden der zu implementierenden Schnittstelle für den Delegationsansatz enthält.

- `_mathTie.java`

Eine Klasse, die als Gerüst (Skeleton) dient, Aufrufe vom ORB empfängt und an die zu implementierende Klasse weiterleitet.

Gleichgültig wie man den IDL-Compiler aufruft ist der Wechsel und das Mischen der Verfahren jederzeit bei der weiteren Implementierung möglich.

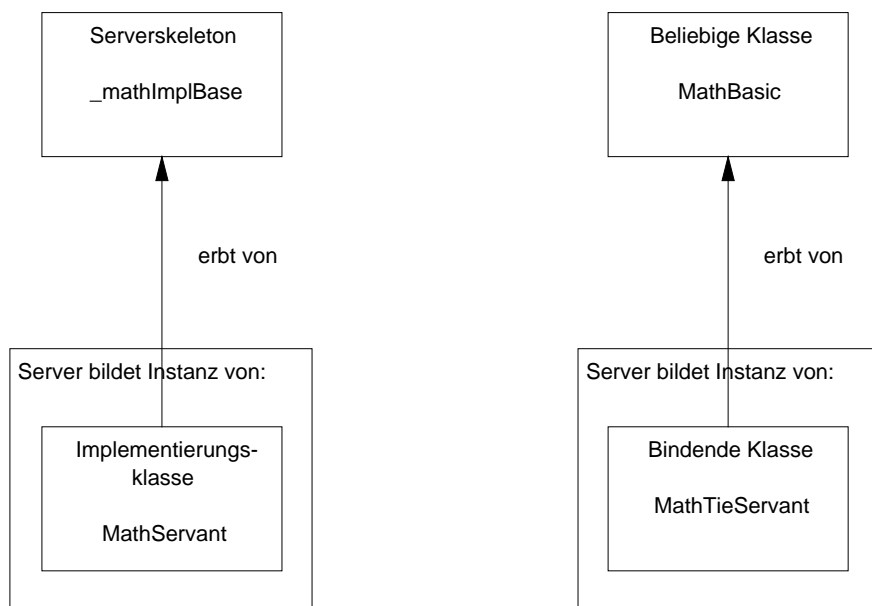


Abbildung 2.10: Gegenüberstellung Vererbungsansatz / Delegationsansatz

Der Delegationsansatz ist dadurch gekennzeichnet, dass die Implementierungsklassen nicht von einer bestimmten Klasse durch Vererbung abgeleitet werden müssen. Dies kommt dem Entwickler bei Java aus oben genannten Gründen sehr entgegen.

Das Serverbeispiel auf den Delegationsansatz angewendet:

```
package CORBATest;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

// Hier kann eine beliebige Klasse MathBasic
// verwendet werden um den Dienst zu implementieren

// Änderungen an der Klassenvererbung

class MathBasic
{
    public double add(double m1, double m2)
    {
        return m1 + m2;
    }
}

class MathServantTie extends MathBasic implements _mathOperations
{
}

// Ende der Änderungen an den Klassen

// Die Klasse ServerTie bildet eine Instanz des MathServantTie-
// Objektes, initialisiert und bindet die Instanz an den
// ORB und registriert die Klasse beim Nameservice

public class ServerTie {

    public ServerTie() {
    }

    public static void main(String args[])
    {
        try{
            // Erzeugt eine Instanz des ORB-Objektes
            // und initialisiert es
            ORB orb = ORB.init(args, null);

            // Erzeugen der Instanz und Binden an ORB
            MathServantTie servant = new MathServantTie(); // Hier wurde
            math helloRef = new _mathTie(servant); // geändert !
            orb.connect(helloRef);

            // ... Vergleiche Beispiel Vererbungsansatz
        }
    }
}
```

Codebeispiel 2.3: CORBA-Server mit Delegationsansatz
(Das vollständige Beispiel befindet sich im Anhang)

Die Implementierung einer Client-Anwendung geschieht in der Regel unabhängig von den verschiedenen Implementierungsansätzen.

Die Client-Anwendung benötigt lediglich eine Objektreferenz auf die Schnittstellenklasse:

```
package CORBATest;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class Client {

    public Client() {

    }

    public static void main(String args[]) {
        try{
            // Erzeugt eine Instanz des ORB-Objektes
            // und initialisiert es
            ORB orb = ORB.init(args, null);

            // Initialisierung des Zugriffs auf Nameservice
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);

            // Auffinden der Objektreferenz beim Nameservice
            NameComponent nc = new NameComponent( "Math" , "");
            NameComponent path[] = {nc};
            math helloRef = mathHelper.narrow(ncRef.resolve(path));

            // Aufrufen der Remote-Methode
            double result = helloRef.add(5.3, 2.9);
            System.out.println(
                "Das Ergebnis der Berechnung aus 5.3 und 2.9 ist: "
                + result);

        } catch (Exception e) {
            System.out.println("ERROR : " + e) ;
            e.printStackTrace(System.out);
        }
    }
}
```

Codebeispiel 2.4: CORBA-Client

2.3.4 Alternativen zu CORBA

Beim Entwerfen und Implementieren verteilter Anwendungen ist CORBA nicht die einzige Möglichkeit für den Entwickler. Es gibt auch noch andere Mechanismen, mit deren Hilfe solche Anwendungen entwickelt werden können. In Abhängigkeit von der Art der Anwendung, angefangen bei ihrer eigenen Komplexität über die Plattform(en), auf denen sie läuft, bis hin zu der bzw. den zum Implementieren verwendeten Sprache(n), gilt es für jeden Entwickler, folgende Alternativen zu bedenken:

Socket-Programmierung

In den meisten neueren Betriebssystemen erfolgt die Kommunikation zwischen den Rechnern und manchmal auch zwischen den Prozessen auf demselben Rechner über Sockets. Einfach gesagt, ein Socket ist ein Kanal, über den Anwendungen miteinander verbunden werden und kommunizieren. Verwendet man, z.B. TCP/IP als Transportprotokoll für eine Anwendung ist das Socket eine Adresse, die aus der IP-Adresse des Rechners und der Port-Nummer des Dienstes besteht. Bei einem Rechner mit der IP-Adresse 129.69.235.30 findet man so den HTTP-Dienst, den „Web-Server“, über die Port-Nummer 80 (Diese Portnummer ist von den Standards für diesen Dienst vorbestimmt). Das einfachste Verfahren zur Kommunikation zwischen Anwendungskomponenten ist die direkte Verwendung von Sockets (dies wird als Socket-Programmierung bezeichnet), was bedeutet, dass der Entwickler Daten auf das Socket schreibt und/oder daraus liest. Dies bedeutet für den Entwickler, dass er das eigentliche Kommunikationsprotokoll der Anwendung selbst implementieren muss. Obwohl die direkte Socket-Programmierung sehr effektive Anwendungen ergeben kann, ist dieser Ansatz allerdings für komplexe Anwendungen in der Regel zu aufwändig.

Remote-Prozeduraufruf (RPC)

Remote-Prozeduraufruf (Remote Procedure Call, RPC) bietet eine funktionsorientierte Schnittstelle zur Kommunikation auf Socket-Ebene. Anstatt die zum und vom Socket strömenden Daten direkt zu bearbeiten, definiert der Entwickler mit RPC eine Funktion, in einer funktionalen Sprache wie C, und generiert Code, der dafür sorgt, dass die Funktion für die aufrufende Anwendung wie eine normale Funktion aussieht. Genau genommen verwendet die Funktion tatsächlich Sockets, um mit einem Remote-Server zu kommunizieren, der die Funktion ausführt und das Ergebnis zurückgibt, wobei wiederum Sockets verwendet werden. Da RPC eine funktionsorientierte Schnittstelle bietet, ist dieses Verfahren oft einfacher zu verwenden als die reine Socket-Programmierung. Obwohl es unterschiedliche, nicht miteinander kompatible Implementierungen des RPC-Protokolls gibt, steht ein Standard-RPC-Protokoll zur Verfügung, das auf den meisten Plattformen ohne Änderungen eingesetzt werden kann.

Distributed Computing Environment (DCE) von OSF

Die DCE (Distributed Computing Environment, verteilte Rechnerumgebung) umfasst eine Reihe von Standards, für die die Open Software Foundation (OSF) die Pionierarbeit geleistet hat, und enthält einen Standard für RPC. Obwohl es den DCE-Standard bereits eine ganze Weile gibt und er sinnvoll ist, hat er nie breite Anerkennung gefunden.

Distributed Component Object Model (DCOM) von Microsoft

Das DCOM (Distributed Component Object Model, Objektmodell für verteilte Komponenten) stellt Microsofts Einstieg in den Bereich der verteilten Anwendungssysteme dar und bietet teilweise Funktionen, die denen von CORBA entsprechen. DCOM wird von den Betriebssystemen von Microsoft besonders gut unterstützt, weil es in Windows 95 und Windows NT integriert ist. Da es sich um eine Microsoft-Technologie handelt, ist DCOM außerhalb der Windows-Welt kaum anzutreffen. Microsoft arbeitet jedoch daran, dies abzustellen, indem es eine Partnerschaft mit der Software AG eingegangen ist, um DCOM auch auf anderen Plattformen als nur Windows verfügbar zu machen (Literatur [7]).

Remote Methode Invocation (RMI) im Java-Standard

RMI, in der Standard-Java-Distribution enthalten, ist eine CORBA-ähnliche Architektur mit einigen Eigenheiten. Ein Vorteil von RMI besteht darin, dass dieses Verfahren die Weitergabe eines Objekts durch seinen Wert unterstützt (pass-by-value), eine Funktion, die derzeit erhältliche Produkte nach CORBA Version 2.0 Standard nicht unterstützen.

Nachteil ist jedoch, dass es sich bei RMI um eine Java-spezifische Lösung handelt, d.h., RMI-Clients und Server müssen in Java geschrieben sein. Für alle Java-Anwendungen, insbesondere diejenigen, die die Funktion zur Weitergabe von Objekten durch Werte nutzen, ist RMI eine gute Wahl, aber wenn die Wahrscheinlichkeit besteht, dass die Anwendung später mit Anwendungen zusammenarbeiten muss, die in anderen Sprachen geschrieben wurden, ist CORBA die bessere Alternative (Literatur [6]).

2.3.5 CORBA-Produkte

JavaIDL (Sun)

JavaIDL ist ein Zusatzprodukt zum Java Development Kit 1.2 von Sun und kann wie dieses kostenlos vom WWW-Server von JavaSoft (<http://www.javasoft.com>) heruntergeladen werden.

Die CORBA-Funktionalität ist den neueren Java Development Kits bereits beigegeben, lediglich der IDL-Compiler ‚idltojava‘ fehlt, da er sich in einem Betastadium befindet. In zukünftigen Versionen des JDK soll er bereits enthalten sein. Zu diesem Produkt gehört auch ein Nameservice, der als ‚nameserv‘ in der Standarddistribution enthalten ist.

Die zu entwickelnde Anwendung wird kostenlos weitergeben. Da für die Weitergabe einer Anwendung auf der Basis des Produktes JavaIDL keine zusätzlichen Lizenzgebühren gezahlt werden müssen, ist dieses Produkt für die Entwicklung einer einfachen CORBA-Lösung prädestiniert.

Es wird im Verlauf des folgenden Kapitels *Realisierung* dargestellt werden, warum dieses Produkt für die Entwicklung des Anwendungssystems ausreichend war.

Ein leicht zu verschmerzender Nachteil des Produktes JavaIDL ist die fehlende Integration in die Entwicklungsumgebungen.

Jaguar

Jaguar ist nicht nur ein CORBA kompatibler ORB sondern vereinigt noch zusätzliche Dienste, wie HTTP-Server, Datenbank-Server-Proxy und Certifying Authority (CA, für gesichertes IIOP über SSL (Secure Sockets Layer)-Verbindungen). Jaguar beinhaltet über ein grafisches Frontend eine assistentengestützte Definition von IDL-Dateien, und den IDL-Compiler.

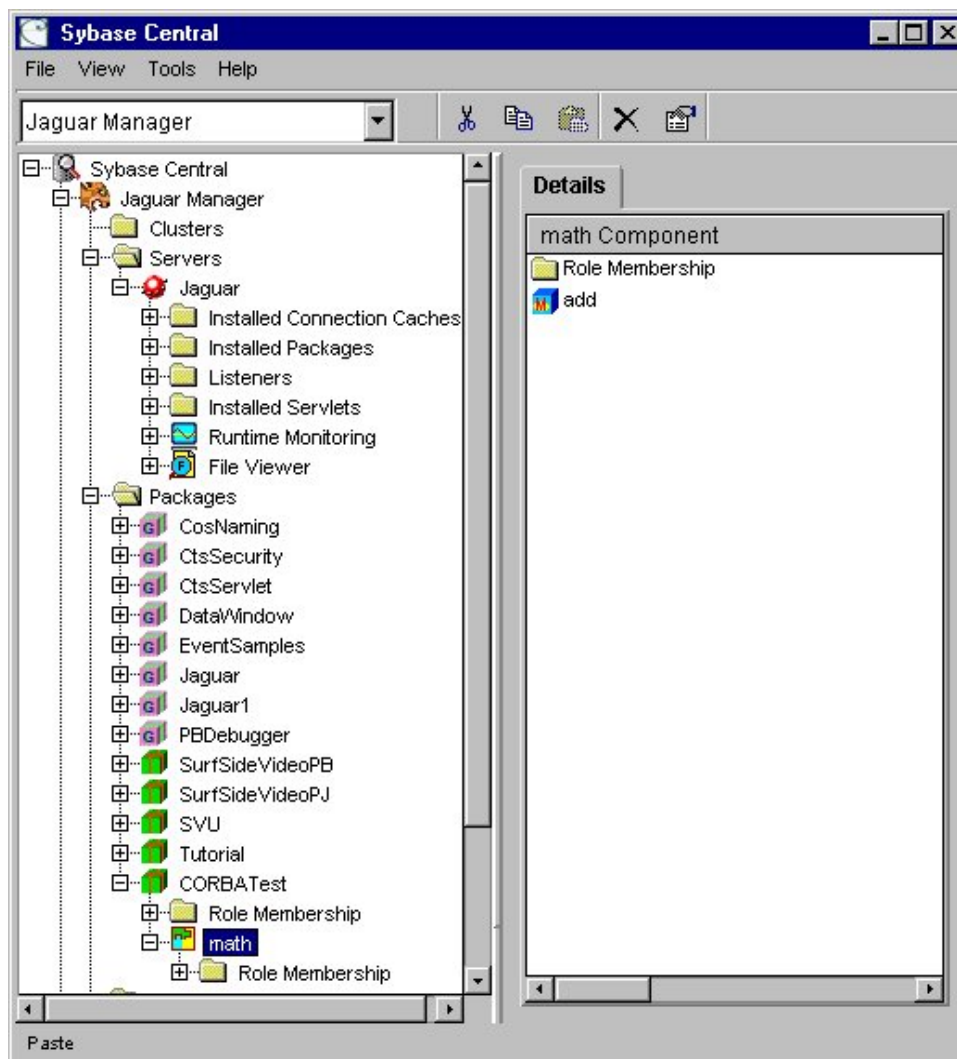


Abbildung 2.11: Administration Jaguar-Server

Jaguar verwirklicht das Server-pro-Methode-Konzept bei der Objektaktivierung. Jaguar befreit dabei den Entwickler von der Notwendigkeit, den Code zur Bindung der Schnittstellenimplementierung an den ORB und die Registrierung des Schnittstellenobjektes bei einem Nameservice zu implementieren. Es genügt für den Server den Code der Schnittstellenimplementierung zu schreiben. Die Aktivierung erfolgt bei den von Jaguar registrierten Komponenten automatisch.

Eine weitere bisher nicht besprochene Komponente der CORBA-Technologie wird ebenfalls durch Jaguar implementiert: das Interface-Repository.

Das Interface-Repository ist eine Komponente, die einen permanenten Speicher für Schnittstellen-Definitionen, die in IDL verfasst wurden, einem ORB zur Verfügung stellt. Ein ORB kann nur dann Dienstanfragen korrekt behandeln, wenn ihm die Schnittstellen-Definitionen bekannt gemacht werden. Dies kann auf zwei Arten geschehen:

1. Wie bereits dargestellt, kann die Information prozedural über Client-Stubs, also Routinen, die das Kommunikationsprotokoll auf implementierungssprachliche Methoden bzw Funktionen abbilden, weitergegeben werden.
2. Die Information kann auch dynamisch über eine CORBA-Schnittstelle des Interface Repository's zur Verfügung stehen. Das Interface Repository ist also ein CORBA-Dienst, der Schnittstellen-Definitionen verwaltet.

3 Realisierung

Bei der Implementierung des Systems wurde das Sun Java Development Kit (JDK) Version 1.2 eingesetzt. Das JDK in dieser Version besitzt bereits viele Eigenschaften, die die Implementierung von Benutzungsoberflächen, die Programmierung von Datenbankfunktionen sowie die Entwicklung verteilter Anwendungen mit CORBA bzw. RMI unterstützen (Literatur [5]).

Leider ist die Integration dieser Java-Version in WWW-Browser und Entwicklungsumgebungen z.T. noch sehr ungenügend. Dies gilt insbesondere für die im Projekt zu verwendende Entwicklungsumgebung PowerJ. Die Unzulänglichkeiten der Oberfläche, der Projektverwaltung und des Codeframeworks ließen den Einsatz dieser Entwicklungsumgebung scheitern. Daran änderte auch die Integration des zu untersuchenden CORBA-Produktes Jaguar in die IDE nichts. Es stellte sich vielmehr heraus, dass auch das Produkt Jaguar auf Grund hoher Runtime-Lizenzgebühren für das Projekt ungeeignet war. Es blieb für die Entwicklung also lediglich das JDK.

Erfreulicherweise gibt es allerdings für einige Plattformen zur Implementierung des JDK ein WWW-Browser-Plugin, das den Betrieb von Applets im WWW-Browser mit einem zusätzlich zu installierenden JDK 1.2.x ermöglicht. Dieses Plugin ist jedoch noch nicht für alle Betriebssystemplattformen für die das JDK Version 1.2.x erhältlich ist, implementiert. Daher wurde ein Teil der Anwendung, d.h. die Präsentationskomponente, so konzipiert, dass sie sowohl als Applet im WWW-Browser, als auch als Anwendungsprogramm ohne WWW-Browser betrieben werden kann. Die CORBA-Server-Komponente muss als Java-Anwendung betrieben werden, da in einem Applet, wie bereits beschrieben, der Zugriff auf Systemkomponenten, nur sehr eingeschränkt möglich ist.

3.1 Grundkonzepte der Anwendung

Das Anwendungssystem wurde als Three-tier-Architekturmodell konzipiert.

Eine Präsentationskomponente (Abbildung 3.2), die für die Visualisierung der Datenbankdaten zuständig ist und die Navigation durch die Datensätze erlaubt, wird als Java-Applet im WWW-Browser ausgeführt.

Die Anwendungslogik wird durch eine CORBA-Server-Komponente gebildet, die als Java-Anwendung auf einer Servermaschine betrieben wird. Der Betrieb dieser Komponente als Java-Anwendung ist deshalb notwendig, da als dritte Komponente der Anwendungsarchitektur ein Microsoft SQLServer benutzt wird, auf den nur mit einem JDBC-ODBC-Bridge-Treiber von Java aus zugegriffen werden kann. Dieser Treiber ist innerhalb von Java-Applets nicht nutzbar, da er wie bereits in Kapitel 2.2.2 erwähnt Zugriff auf Betriebssystemressourcen nehmen muss, die durch das Applet-Sicherheitsmodell geschützt werden.

Darüberhinaus bietet die Einführung einer Anwendungslogikkomponente die Möglichkeit, die Datenbankdaten für die Ausgabe in der Präsentationskomponente aufzubereiten. So ist z.B. eine Leistungssteigerung dadurch zu erwarten, indem die Menge der zwischen Datenbank und Präsentationskomponente übertragenen Daten auf den für die Anzeige notwendigen Datenbestand eingeschränkt wird. So ist es z.B. für die grafische Anzeige der Ofentemperaturen nicht notwendig, sämtliche Ofentemperaturdaten zu übertragen, da das Auflösungsvermögen

der Anzeige beschränkt ist. In diesem Fall würde es genügen, genügend Datenpunkte zu äquidistanten Zeitpunkten zu ermitteln, die dann für die grafische Anzeige in einer Kurvendiagrammdarstellung als ausreichend erachtet werden können. Neben dieser 'Sampling'-Methode zur Begrenzung der Datenmenge kann auch ein seitenweises Durchblättern der Datenbestände für bestimmte Ansichten implementiert werden, wenn einerseits alle Datensätze einer Datenbankansicht vollständig angezeigt werden müssen andererseits aber nur jeweils ein Datensatz in der Präsentationskomponente gleichzeitig dargestellt werden kann.

Die dritte Komponente der realisierten Anwendungsarchitektur ist eine Datenbank, die als Produkt bereits durch die Aufgabenstellung vorgegeben war. Diese Datenbank enthält Prozessdaten einer Zylinderkopfhärteanlage. Eine vom Kunden gestellte Anforderung an das System war die Prozessdatenbank in vorgegebenen Zeitabständen in eine Archivdatenbank übertragen zu können. Die Anwendung sollte dann transparenten Zugriff auf die Datenbank mit den aktuellen Daten und die gleichstrukturierte Archivdatenbank erlauben.

Eine wichtige Anforderung an das Anwendungssystem war die Möglichkeit, das Software-system so zu gestalten, dass es an quasi beliebige Prozessdatenbanksysteme angepasst werden kann. Dies wird zum Einen erreicht durch die Verwendung von Java als Programmiersprache und Plattform, zum Anderen durch die Nutzung des vorgegebenen Datenbanksystems zur Speicherung von Konfigurationsdaten.

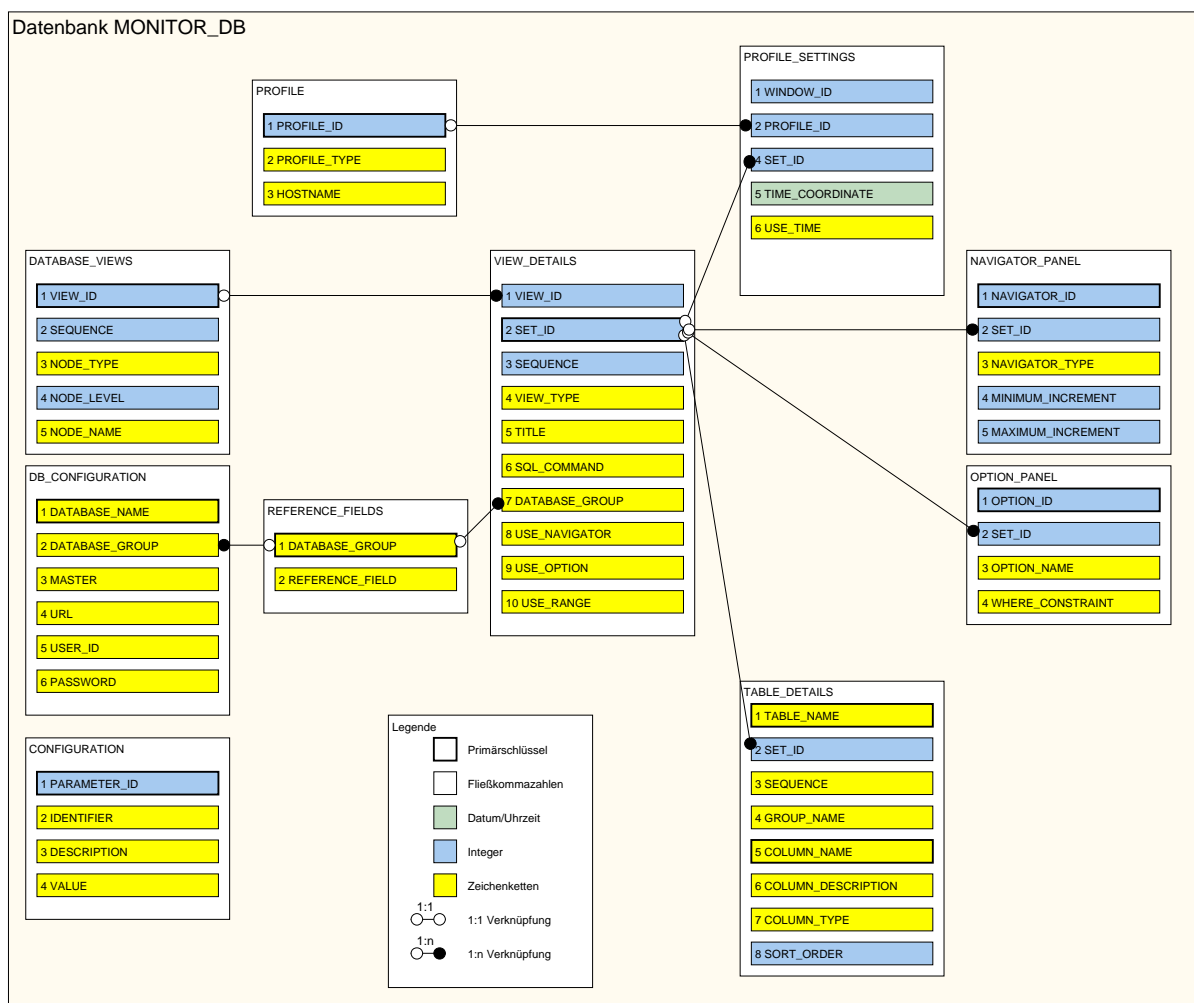


Abbildung 3.1: Konfigurationsdatenbank

Die Konfigurationsdatenbank (Abbildung 3.1) speichert Informationen:

- zur allgemeinen Systemkonfiguration (Tabelle CONFIGURATION),
- Verbindungs- und Konfigurationsparameter der verwendeten Prozessdatenbanken unter Berücksichtigung der Tatsache, dass die Speicherung der Prozessdaten in mehreren Datenbanken mit Archiv und aktuellen Daten erfolgen soll (Tabellen DB_CONFIGURATION und REFERENCE_FIELDS),
- Informationen zu den dargestellten Menüs und Ansichten (Tabellen DATABASE_VIEWS, VIEW_DETAILS),
- der darzustellenden Tabellendaten (Tabelle TABLE_DETAILS)
- und verschiedene Informationen zur Initialisierung von Navigationskomponenten.
- Die Daten einer Benutzerprofilverwaltung werden durch die beiden Tabellen PROFILE und PROFILE_Settings in der Datenbank festgehalten.

Die folgende Abbildung zeigt, auf welche Bereiche der Präsentationskomponente sich die Tabellen der Konfigurationsdatenbank auswirken:

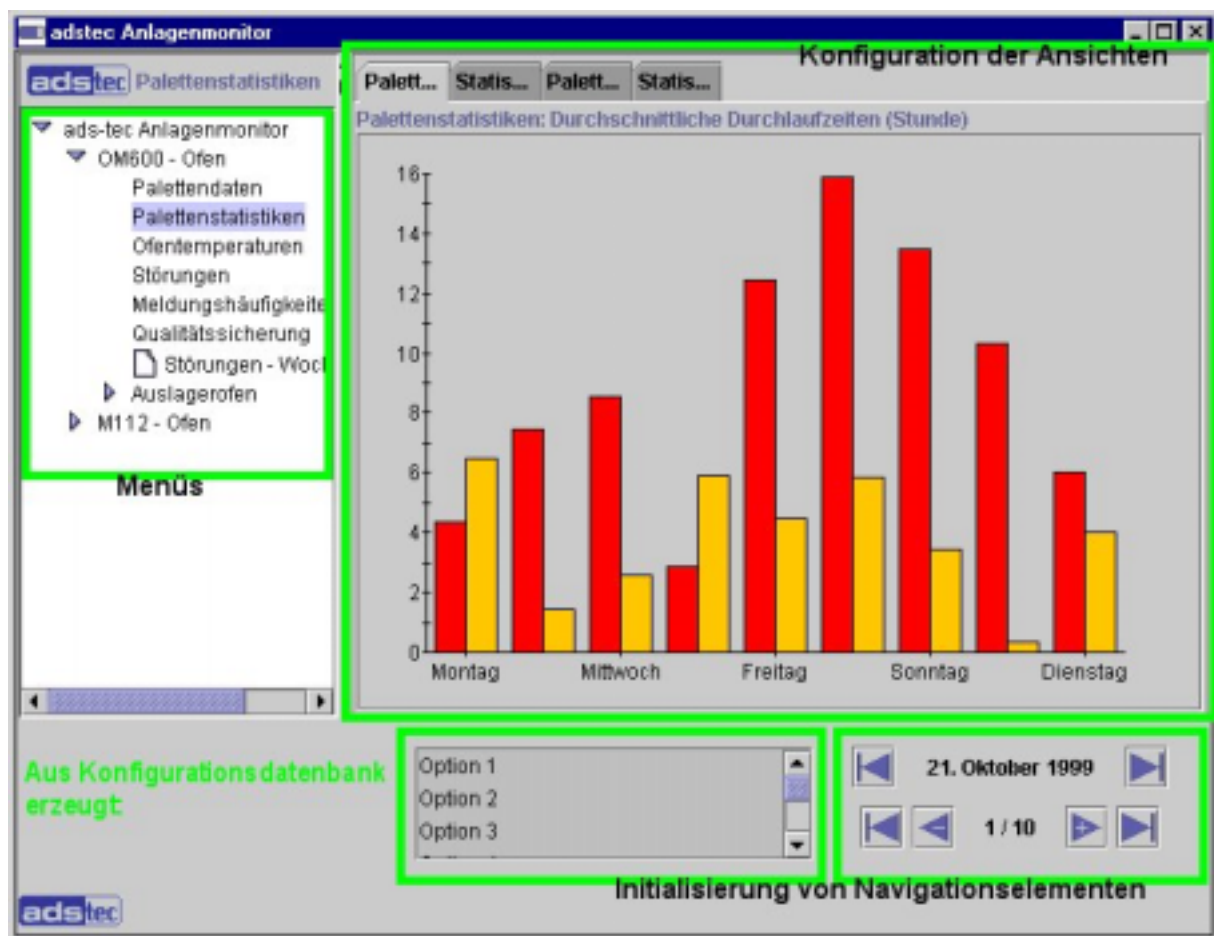


Abbildung 3.2: Präsentationskomponente
Initialisierung der Ansichten aus der Konfigurationsdatenbank

Eine weitere Anforderung an das System war durch den Kundenwunsch gegeben, dass nach einem Neustart der Präsentationskomponente die Einstellungen der letzten Ansicht der Komponente wiederhergestellt werden können. Dies wurde durch die Integration zusätzlicher Tabellen in der Konfigurationsdatenbank erreicht, die die Verwaltung von Benutzerprofilen ermöglichen. Die Dienste der Benutzerwaltung werden ebenfalls von der Anwendungslogikkomponente mittels CORBA-Diensten der Präsentationskomponente zur Verfügung gestellt.

Neben dem Zugriff auf Benutzerprofildaten ermöglicht die CORBA-Serverkomponente das Erstellen von Reports in Form von HTML-Dateien. Diese enthalten bestimmte Datenbankauszüge in tabellarischer Form und werden entweder in der Anwendung dargestellt oder bei Ausführung als Applet in einem WWW-Browser in einem zusätzlichen Browserfenster dargestellt. Damit die Übertragung der Reports in den Browser bzw. die Anwendung möglich wird, ist als zusätzliche Komponente im Anwendungssystem ein Web-Server vorgesehen, der zudem auch das Herunterladen der Präsentationskomponente in den WWW-Browser von einer zentralen Stelle aus erlaubt.

Das folgende Diagramm stellt die verschiedenen Komponenten des Systems, die Kommunikationswege und die ausgetauschten Daten dar:

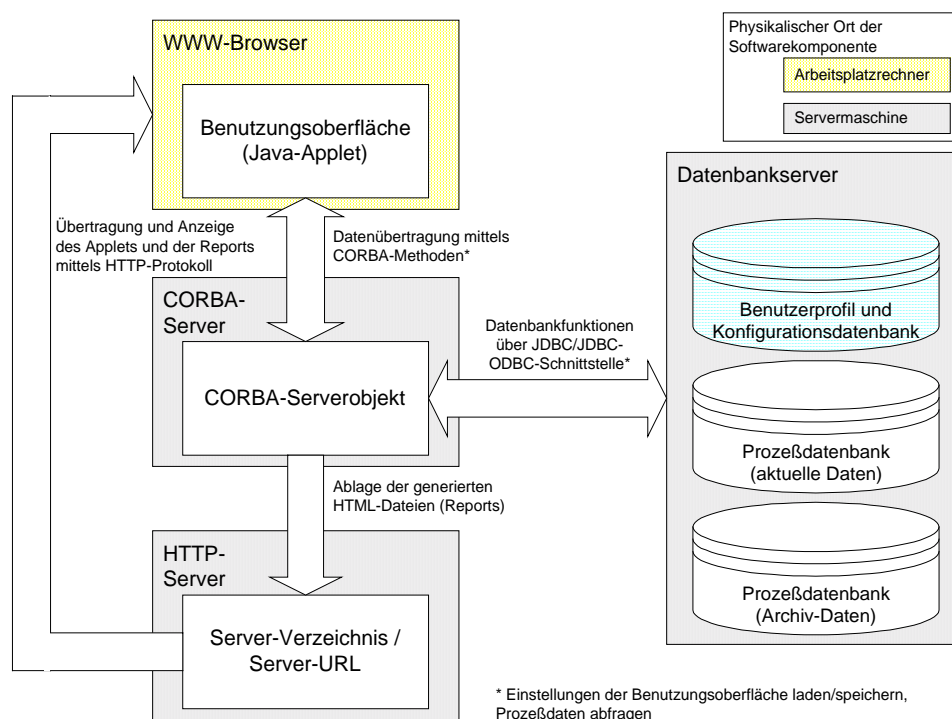


Abbildung 3.3: Architekturmodell der Anwendung

3.2 Applet oder Anwendung ?

Eine Anwendung ist in Java dadurch gekennzeichnet, dass eine als Anwendungsklasse bezeichnete Objektdefinition eine Methode mit dem Namen main() aufweist. Damit diese Methode beim Start der Anwendung ausgeführt werden kann, muss sie zusätzlich mit den Modifizierern public und static gekennzeichnet werden, da das Sicherheitsmodell von Java auch bei Anwendungen den externen Zugriff auf nicht öffentliche Objektinstanzen verbietet.

Ein Applet hingegen verfügt über die Methoden `init()` und `run()`, die unmittelbar nach dem Laden des Applets ausgeführt wird. Definiert man nun eine Klasse, die man von der Appletklasse `java.applet.Applet` bzw. `javax.swing.JApplet` ableitet und mit allen 3 Methoden ausstattet, so kann eine Anwendung sowohl als Applet als auch als Anwendung ausgeführt werden, wenn diese Anwendung in der `main()`-Methode:

1. Für die Anwendung eine Instanz von `java.awt.Frame` bzw. `javax.swing.JFrame` bildet.
2. Eine Instanz von `java.applet.Applet` bildet und diese dem Frame-Objekt mit der Methode `Frame.getContentPane().add()` hinzufügt.
3. die Methoden `init()` und `run()` des Applet-Objektes aufruft.

Dies wird im folgenden Codebeispiel (aus `adstec.Monitor`) gezeigt:

```
//Main-Methode
public static void main(String appargs[]) {
    cmdArgs = appargs;
    isApplet = false;

    // vgl 2.
    Monitor mapplet = new Monitor();
    mapplet.isStandalone = true;

    // vgl 1.
    JFrame frame = new JFrame();

    // vgl 2.
    frame.getContentPane().add(mapplet, BorderLayout.CENTER);

    // vgl 3.
    mapplet.init();
    mapplet.start();

    // weitere Initialisierungen
    frame.setTitle("adstec Anlagenmonitor");
    frame.setSize(640, 500);
    Dimension d = Toolkit.getDefaultToolkit().getScreenSize();
    frame.setLocation((d.width - frame.getSize().width) / 2,
                     (d.height - frame.getSize().height) / 2);
    frame.setVisible(true);
} // End main()
```

Codebeispiel 3.2: `main()`-Methode aus `adstec.Monitor`

Auf diese Art und Weise sind ein Applet bzw. eine Anwendung (`adstec.Monitor`) entstanden, die als Einzelfensterlösung ausgeführt werden kann. Zusätzlich wurde eine Variante definiert, die als Mehrfenster-MDI-Anwendung (`adstec.MDIApplet`) ausgeführt werden kann. `MDIApplet` ist von `Applet` abgeleitet und überschreibt lediglich die Methoden `main()` und `init()` von `Applet`, sodass bei Änderungen an der Definition von `Applet` lediglich darauf zu achten ist, ob die Definitionen in den überschriebenen Methoden von `MDIApplet` geändert werden müssen.

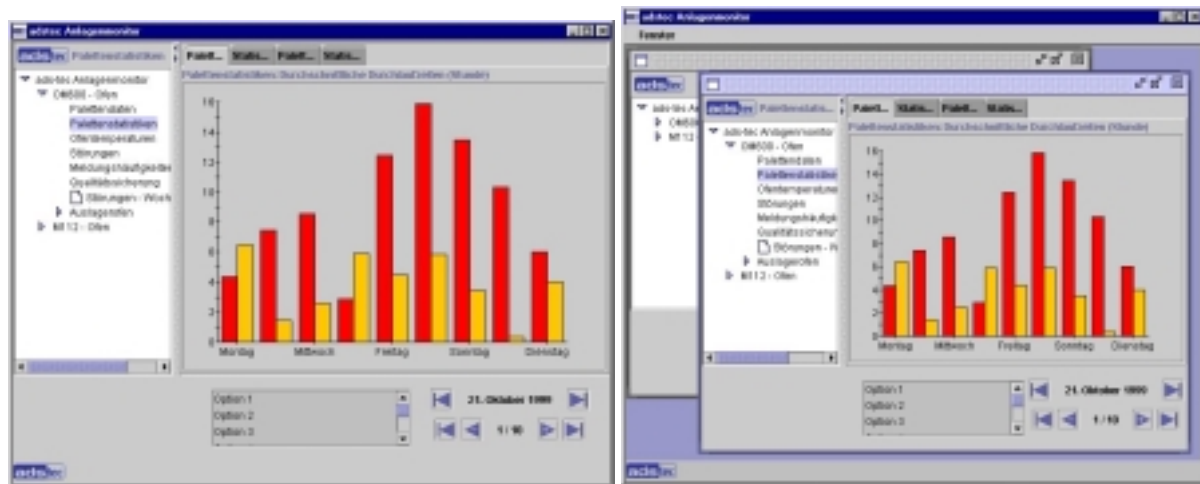


Abbildung 3.4: adstec.Applet und adstec.MDIApplet im Vergleich

Gleichgültig ob adstec.Applet oder adstec.MDIApplet gestartet und genutzt wird; beide Anwendungen bilden ein oder mehrere Instanzen der Klasse `DataPanel`, die einerseits das Fenster in die drei Bereiche Menü (links), Ansichtsbereich (rechts) und Navigationsbereich (unten) teilt, andererseits Methoden zur Ereignisbehandlung der enthaltenen Komponenten besitzt. `DataPanel` ist die 'Schaltzentrale' der Präsentations-Anwendung: die Methoden der Klasse führen die Funktionen zur Bildung von Objektreferenzen und schließlich die CORBA-Methodenaufrufe aus.

3.2.1 Unterschiede im Sicherheitskonzept

Java Anwendungen werden in vier verschiedenen Sicherheits-Kontexten betrieben:

1. Anwendung

Eine Java-Anwendung besitzt wie andere nicht in Java implementierte Programme vollständigen Zugriff auf Betriebssystem-Ressourcen. Einer Java-Anwendung ist es z.B. erlaubt auf das Dateisystem zuzugreifen, um Dateien zu lesen und zu schreiben, auf Betriebssystemressourcen wie Speicher, Netzwerk und Peripheriegeräte zuzugreifen und zu nutzen. Mögliche Sicherheitsmechanismen wie Speicherschutz, Dateisystem- und allgemeiner Systemzugriffsschutz müssen daher durch das Betriebssystem bereitgestellt werden.

2. Applet im Appletviewer

Der Appletviewer ist eine Anwendung im JDK, die das Testen von Applets, die normalerweise im WWW-Browser angezeigt und betrieben werden, ermöglicht. Ganz im Sinne dieser Aufgabenstellung bietet der Appletviewer nur sehr beschränkten Zugriffsschutz. Darüberhinaus lässt sich der Zugriffsschutz durch eine Kommandozeilenoption fast vollständig ausschalten. Leider erlaubt der Appletviewer dadurch nur eine sehr eingeschränkte Beurteilung des Betriebs der Anwendung im WWW-Browser.

3. Lokales Applet im WWW-Browser

Legt man die Klassendateien einer Java-Applet-Anwendung im lokalen Dateisystem ab, so kann man das Applet im WWW-Browser anzeigen und betreiben, wenn man eine

entsprechend vorpräparierte HTML-Datei hinzufügt, die das so genannte <APPLET>-Tag oder das <EMBED>-Tag enthält. Mit Hilfe dieser Tags wird in einer HTML-Datei beschrieben, wo das Java-Applet zu finden ist und mit welchen Parametern es initialisiert und dargestellt werden soll. Wird das Java-Applet auf diese Weise vom lokalen Dateisystem aus betrieben, benötigt es in diesem Fall vollen Lesezugriff auf das Dateisystem. Hierbei wirkt sich die Sicherheitsarchitektur von Java, die den Zugriff auf Systemressourcen beschränken sollte, nicht aus. Weitere Sicherheitsmechanismen des Java-Applet-Betriebs sind jedoch aktiviert. So ist es z.B. nicht möglich, auf andere Stationen im Netzwerk Zugriff zu erlangen.

4. Applet im WWW-Browser über HTTP (Hypertext-Transport-Protocol) geladen

Hierbei greifen sämtliche für den Java-Applet-Betrieb vorgesehene Sicherheitsmechanismen. Ein von einem Web-Server geladenes Applet besitzt grundsätzlich keine Möglichkeit des Zugriffs auf Systemressourcen der lokalen Arbeitsstation. Weiterhin erlaubt das Sicherheitsmodell einem Applet lediglich eine Netzverbindung zu der Station im Netzwerk aufzubauen von der es selbst geladen wurde.

Der Browser unterscheidet die beiden Zugriffsarten des Applets nach den hier vorgestellten Punkten 3. und 4. auf Grund der URL, die als Adressangabe im WWW-Browser eingegeben wird. Eine URL, die mit dem Protokolldiskriminator [file:///](#) beginnt erlaubt für das Applet ein Sicherheitsmodell nach Punkt 3., jede andere URL mit Protokolldiskriminatoren wie [http:///](#) und [ftp:///](#) erlaubt einem Applet nur den Betrieb mit höchster Sicherheitsstufe. Über spezielle Programmiermethoden und einen Vorgang, den man als Signieren eines Applets bezeichnet, lassen sich allerdings einzelne Rechte freischalten. Dies geschieht jedoch nicht unbemerkt vom Benutzer: Bevor ein Applet diese Rechte nutzen kann, muss der Anwender durch Bestätigen eines Dialogfensters, die Nutzung dieser Rechte dem Applet zugestehen.

Die verschiedenen Sicherheitskontexte einer Java-Anwendung besitzen wesentlichen Einfluss auf die Entwicklung von Anwendungen als Applets. Das Verhalten eines Applets innerhalb der Sicherheitskontexte muss bei der Entwicklung berücksichtigt und getestet werden. Während der Entwicklung des Anwendungssystems trat mehrfach der Fall ein, dass sich das System auf Grund von Sicherheitskonflikten nicht mehr in Betrieb nehmen ließ. Die Sicherheitskonflikte ließen sich nur durch Verwendung anderer Methoden und Objektklassen umgehen.

So löst z.B. die Methode `Object.class.getResource("Bilddateiname")`, die häufig zum Laden von Bilddateien verwendet wird eine Sicherheitsproblemmeldung im WWW-Browser aus, während die Verwendung der Funktion in einer Anwendung unkritisch ist. Andere Methoden, die zum Laden von Bilddateien verwendet werden können z.B. `javax.swing.ImageIcon()` versagen allerdings in einem Applet, da meist von der Methode der Quellpfad der Datei nicht korrekt ermittelt werden kann. Im folgenden Kapitel 2.2.2 wird eine Funktion gezeigt, die die Unterschiede zwischen Java-Applet und Anwendung bei der Ermittlung der Quellpfade bzw. URL beim Laden von Bilddateien in das Java-Programm ausgleicht.

3.2.2 Laden von Bilddateien

Externe Bilddateien werden zur Dekoration von Schaltflächen und anderer Elemente in eine Java-Anwendung geladen. Die in Java-Umgebungen eingesetzte JFC-Bibliothek (Java Foundation Classes, Codename: Swing) zur GUI-Gestaltung bietet ein Objekt

(`javax.swing.ImageIcon`) an mit dem auf einfache Weise Bilder geladen und Bildobjekte erzeugt werden können. `ImageIcon` erwartet bei der Erzeugung einer Instanz seiner Klassen den Bilddateinamen oder die URL (Uniform Resource Locator) unter der diese Datei von einem WWW-Server geladen werden kann. Bei der Verwendung in einem Applet muss allerdings zunächst mit der Methode `getCodeBase()`; die URL ermittelt werden unter der das Applet von einem WWW-Server geladen wurde. Anschließend kann mit Hilfe der ermittelten URL die Bilddatei von demselben WWW-Server geladen werden von dem aus auch das Applet geladen wurde.

```
// isApplet = false (bei der Ausführung als Anwendung)
...
// folgende Zeile ist in der init()-Methode von adstec.Monitor enthalten
// Definition: static boolean isApplet = false;
    if (isApplet) myURL = getCodeBase();
...
public static ImageIcon loadImageIcon(String filename, String description)
{
    if(!isApplet) {
        return new ImageIcon(filename, description);
    } else {
        URL url;
        try {
            url = new URL(myURL, "adstec/" + filename);
        } catch(MalformedURLException e) {
            System.err.println("Error trying to load image " + filename);
            return null;
        }
        return new ImageIcon(url, description);
    }
}
```

Codebeispiel 3.3: Statische Methode für das Laden von Bilddateien (aus `adstec.Monitor`)

Die dargestellte Methode ist als statisch gekennzeichnet, damit andere am Projekt beteiligte Klassen diese Methode als Klassenmethode, d.h. ohne Instanzbildung, nutzen können.

3.2.3 Initialisierung des ORB's

Die Initialisierung von ORB-Objekten bei der Nutzung der CORBA-Funktionen von Java ist bei der Erstellung von Java-Applets und Java-Anwendungen verschieden.

In einer Anwendung wird das ORB-Objekt gewöhnlich mit den verschiedenen Kommandozeilen-Optionen, die man der Anwendung beim Start übergeben hat, initialisiert. In einem Applet geschieht dies noch sehr viel einfacher, indem man den ORB mit einem Zeiger auf das Applet-Objekt selbst initialisiert. (Dabei wird meist der Operator `this` verwendet.):

Java-Anwendung: `ORB myORB = new ORB.init(args, null);`
 (`args` ist ein Array mit `String`-Objekten, das gewöhnlich von der Kommandozeile beim Start eines Programms an das Programm übergeben wird.)

Java-Applet: `ORB myORB = new ORB.init(this, null);`

Innerhalb einer 'Zwitterapplikation', d.h. einer Applikation, die als Java-Applet und als Java-Anwendung einsetzbar sein soll, ist dieser Unterschied störend. Der zweite Parameter des ORB-Objekt-Konstruktors `init()` ist ein Property-Objekt. Das Property-Objekt (`java.util.Properties`) erlaubt mit seinen Methoden anhand von Schlüsselwerten String-Objekte in einem Java-Programm zu speichern und zu laden. Damit bei der Initialisierung des ORB-Objektes eine Gleichbehandlung bei der 'Zwitterapplikation' möglich ist, werden daher zunächst alle Kommandozeilenparameter bzw. alle Applet-Parameter in einem Property-Objekt gespeichert, sodass das es später bei der Instanzbildung eines ORB-Objektes verwendet werden kann.

Die Instanzbildung ist dadurch im Einsatz in Applet und Anwendung einheitlich:

```
String args[] = {" "};  
ORB myORB = ORB.init(args, props);
```

3.3 Visualisierungsklassen

Die Datenbankdaten sollen in verschiedenen Ansichten präsentiert werden:

- Karteikartenartiges Formular
- Tabellen
- Grafiken (Histogramme und Kurven)

Da in einem Menüpunkt der Anwendung mehrere Ansichten auf die Datenbanken zusammengefasst sind, werden die Visualisierungsklassen durch eine Swing-Container-Komponente `JTabbedPane` gruppiert.

Alle oben beschriebenen Ansichten auf die Datenbank sind von einer Grundklasse `ViewPanel` abgeleitet, die wiederum von der Swing-Container-Klasse `JPanel` abgeleitet ist. `ViewPanel` enthält an visuellen Komponenten lediglich eine Titelzeile (`JLabel`), deren Text mit einer Methode `setTitle()` geändert werden kann. Ferner wurde für den verbleibenden Bereich der Panel-Komponente eine weitere Swing-Container-Komponente eingesetzt: `JScrollPane`. Auf diese Weise wird sichergestellt, dass eine weitere Komponente, die in diesen Container gesetzt wird, auch bei Verkleinerung des sichtbaren Bereiches des Visualisierungsbereiches, darstellbar bleibt. Wird der Bereich zu klein, werden automatisch Verschiebepfeile eingeblendet, die ein Anzeigen der nicht sichtbaren Bereiche einer eingesetzten Komponente ermöglichen. Die Klasse `ViewPanel` besitzt eine für abgeleitete Klassen zu überschreibende Methode `setData()`.

Die von `ViewPanel` abgeleiteten Klassen erlauben die gewünschten Darstellungen der Datenbankdaten sowie:

- einfache Texte (`TextPanel`)

In den Bereich von `JScrollPane` wurde eine `JTextArea`-Komponente eingefügt. mit Methoden wie `addText()`, `print()` und `println()` lassen sich Texte darstellen.



Abbildung 3.5: TextPanel

- Bilder (ImagePanel)

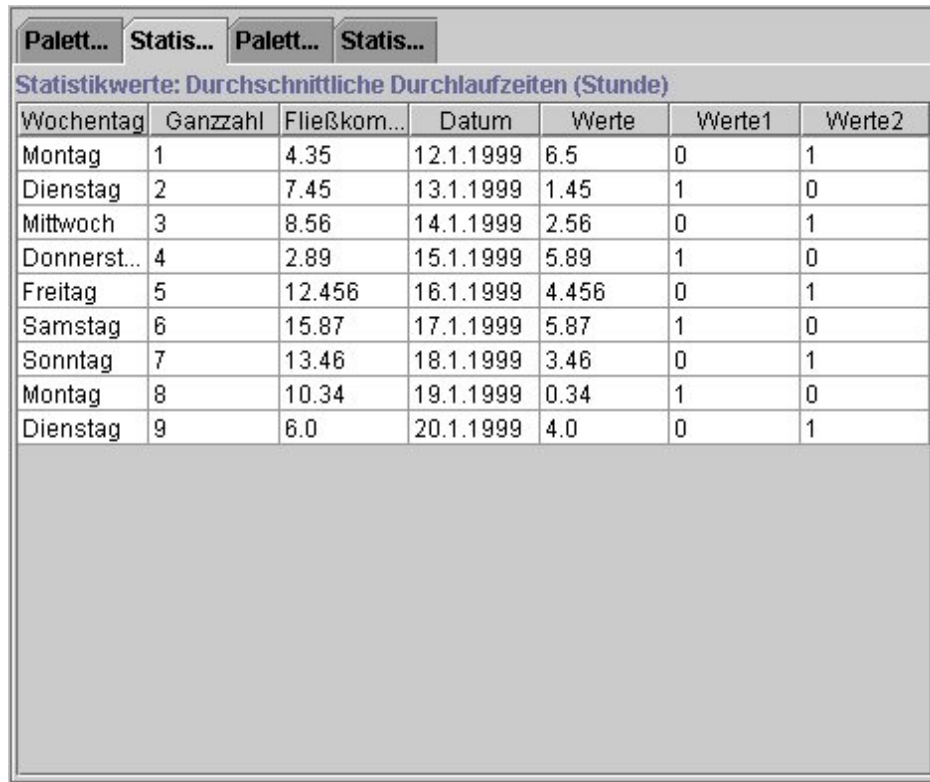
Diese Klasse besitzt eine JLabel-Komponente, die es erlaubt, über eine URL ein Bild zu laden und anzuzeigen (setImage()).



Abbildung 3.6: ImagePanel

- Tabellen (TablePanel)

Diese Klasse erlaubt die Darstellung von Datenbankdaten in einer einfachen Tabelle. Die Methode setData() erzeugt aus den mittels CORBA-Methoden übertragenen Daten eine Instanz der Klasse TablePanelModel. Dieses Tabellenmodell wird von einer Instanz der Klasse JTable genutzt. Typisch für das Swing-Framework ist das zur Änderung des Aussehens und der Eigenschaften von visuellen Swing-Komponenten zunächst die Erstellung einer Modellklasse aus einer meist abstrakten Klasse vorgenommen wird. So auch bei der JTable-Komponente.



Wochentag	Ganzzahl	Fließkom...	Datum	Werte	Werte1	Werte2
Montag	1	4.35	12.1.1999	6.5	0	1
Dienstag	2	7.45	13.1.1999	1.45	1	0
Mittwoch	3	8.56	14.1.1999	2.56	0	1
Donnerst...	4	2.89	15.1.1999	5.89	1	0
Freitag	5	12.456	16.1.1999	4.456	0	1
Samstag	6	15.87	17.1.1999	5.87	1	0
Sonntag	7	13.46	18.1.1999	3.46	0	1
Montag	8	10.34	19.1.1999	0.34	1	0
Dienstag	9	6.0	20.1.1999	4.0	0	1

Abbildung 3.7: TablePanel

- Grafiken (GraphPanel)

Grafiken in Form von Kurven und Histogrammen werden mit Hilfe eines Fremdproduktes der Firma KLLGroup (<http://www.kllgroup.com>) angezeigt. Die mit den üblichen Entwicklungswerkzeugen (Sybase PowerJ, Inprise Borland JBuilder und Symantec Visual Cafe) ausgelieferte Komponentenbibliothek JClass Lite bietet bereits mehr Möglichkeiten zur grafischen Darstellung der Datenbankdaten als letztendlich genutzt werden können. Ähnlich wie bei Tabellen erzeugt die Methode setData() eine Instanz einer Modellklasse (GraphPanelDataSource), die dann von der eigentlichen visuellen Klasse jclass.SimpleChart zur Darstellung genutzt wird.

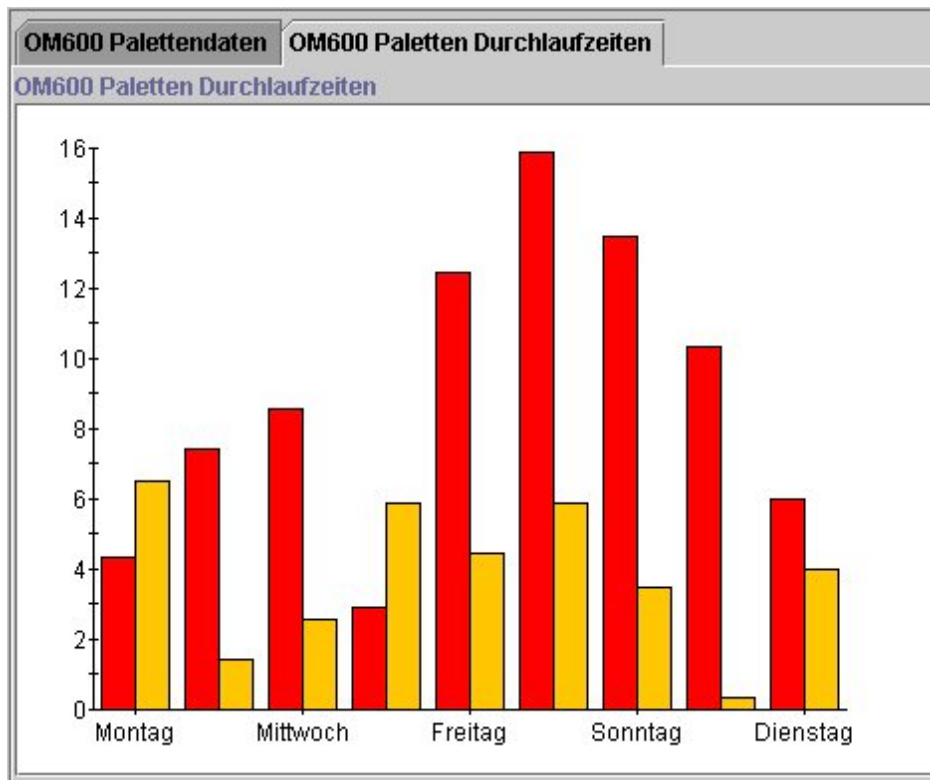


Abbildung 3.8: GraphPanel

- Reports (HtmlPanel)



Abbildung 3.9: HtmlPanel

Wie bereits beschrieben erzeugt der CORBA-Server bei der Auswahl eines Menüpunktes, der durch ein Symbol im Menü als Report markiert ist, eine HTML-Datei, die von einem Web-Server heruntergeladen werden kann. Wird die Präsentationskomponente als Anwendung ausgeführt, erfolgt die Darstellung mittels des HtmlPanel, das hierzu eine JEditorPane-Komponente enthält. Die hierzu verfügbare Methode heißt setPage() und erwartet als einzige Angabe eine URL. Wird die Präsentationskomponente als Applet im WWW-Browser ausgeführt, öffnet sich nach Aufruf der setPage()-Methode ein zusätzliches Browserfenster mit dem gewünschten Report. Diese Komponente besitzt eine zusätzliche Schaltfläche, um einen aktualisierten Report abrufen zu können.

- Karteikartenartige Formulare (CardPanel)

Diese Komponente erlaubt die Darstellung von Datenbankdaten in einem Formular, das jeweils einen einzigen Datensatz anzeigt. Mit den Methoden next() und previous() kann man durch die in die Komponente geladenen Datensätze navigieren. Dies kann durch ein externes Kontrollelement (siehe 3.4.2) geschehen oder durch eine komponenteninterne Navigationsleiste. Die interne Navigationsleiste kann mit den Methoden showInternNavigator() bzw. hideInternNavigator() ein und ausgeblendet werden.

The image shows a Java Swing window titled 'OM600 Palettendaten' with a sub-tab 'OM600 Paletten Durchlaufzeiten'. The main content area is titled 'OM600 Palettendaten' and contains several data fields:

Wochentag:	Montag
Zahlen	
Ganzzahl:	1
Fließkommazahl:	4.35
Datum:	12.1.1999
Weitere Werte	
Werte:	6.5
Boolean	
Werte1:	<input type="checkbox"/>
Werte2:	<input checked="" type="checkbox"/>

At the bottom of the form, there is a navigation bar with left and right arrow buttons and the text '1/9' in the center.

Abbildung 3.10: CardPanel

3.4 Datenbanknavigation

3.4.1 Menükomponente

Die Menükomponente adstec.Tree.TreePanel stellt der Anwendung ein hierarchisches Menü zur Verfügung. Das Menü wird hierbei zur Laufzeit aus Daten der Konfigurationsdatenbank erzeugt.

Die dem Menü zu Grunde liegende Datenbanktabelle besitzt folgende Struktur:

VIEW_ID	SEQUENCE	NODE_TYPE	NODE_LEVEL	NODE_NAME
0	0	D	0	OM600 - Ofen
1	1	V	1	Palettendaten
2	2	V	1	Palettenstatistiken
3	3	V	1	Ofentemperaturen
4	4	V	1	Störungen
5	5	V	1	Meldungshäufigkeiten
6	6	V	1	Qualitätssicherung
7	9	D	1	Auslagerofen
8	10	V	2	Ofentemperaturen
9	11	V	2	Störungen
10	12	V	2	Meldungshäufigkeiten
...
21	14	R	2	Störungen - Wochenbericht
23	8	R	1	Störungen - Wochenbericht

Das aus der Datenbank erzeugte Menü wird wie folgt dargestellt:

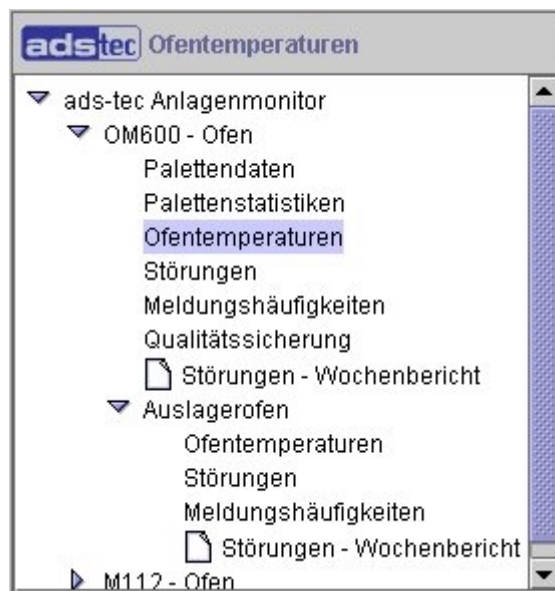


Abbildung 3.11: TreePanel

Bei der Datenbankabfrage werden zunächst mit der SQL-Anweisung

```
SELECT VIEW_ID, NODE_LEVEL, NODE_TYPE, NODE_NAME
FROM DATABASE_VIEWS
ORDER BY SEQUENCE;
```

die Daten ermittelt. Hierbei wurde bereits die Reihenfolge der einzutragenden Menüpunkte durch das SEQUENCE-Feld festgelegt. Nun lässt sich mittels des Feldes NODE_TYPE feststellen, ob es sich hierbei um einen auszuwertenden Menüpunkt V (View, Ansicht) bzw. R (Report) oder um den Knoten eines Unterverzeichnisses D (Directory, Verzeichnis) handelt. Ein weiteres Kriterium zur Strukturdefinition liefert das Feld NODE_LEVEL, das die Verzeichnistiefe der einzelnen Menüpunkte wiedergibt. NODE_NAME schließlich ist der Name der einem Menüpunkt oder Verzeichnisknoten zugewiesen wird. Mit dem Feld VIEW_ID werden die unter diesem Menüpunkt zugänglichen Datenbankansichten abgerufen.

Die Menükomponente bekommt die Daten zum Menüaufbau vom CORBA-Server in folgender Struktur:

(Auszug aus der IDL-Definition)

// DisplayManager Structures and Interface Definition

```
struct menuitem {
    long view_id;
    long level;
    string type;
    string label;
};

typedef sequence <menuitem> menu;

...

interface DisplayManager {

    menu getMenu();

    ...
};
```

Codebeispiel 3.4: IDL-Definition DisplayManager

Wie die IDL-Definition erkennen lässt, handelt es sich hierbei um eine ARRAY-Struktur, die Strukturen vom Typ menuitem enthält. Aus dieser Struktur lässt sich durch einen iterativen Algorithmus die Menüstruktur aufbauen.

Für den Anzeige des Menüs ist eine javax.swing.JTree Komponente zuständig. Wie bei Tabellen ist eine Klasse TreeModel für die Bereitstellung der Daten, die dann mit der JTree-Komponente angezeigt werden, zuständig.

Eine Komponente der Klasse adstec.Tree.TreePanel wird aus den folgenden Swing-Komponenten aufgebaut:

- TreePanel (erbt von JPanel)
 - JLabel logoLabel (eine Swing-Komponente zur Darstellung einzeliger Texte und von Bildern)
 - JScrollPane treeScrollPane (Swing-Container-Komponente, die Verschiebepalken anzeigt)
 - JTree treeMenu (Swing-Komponente zur Darstellung hierarchischer baumartiger Listen)

Eine besondere Herausforderung im Umgang mit Swing JTree-Komponenten stellt die Ereignisbehandlung dar. Wenn mit der Maus auf einen Menüeintrag geklickt wird, wird ein

Ereignisobjekt erzeugt, das durch eine Methode im Rahmen der Ereignisbehandlung ausgewertet werden kann. Das Ereignisobjekt enthält dem ausgelösten Ereignis entsprechende Daten. Im Falle der JTree-Komponente, sind dies zeichenkettenartige Pfadstrukturen. Ein Pfad enthält in einer Array-Struktur die Namen sämtlicher Baumknotenpunkte angefangen vom dem obersten Knotenpunkt. Für die weitere Ereignisbehandlung wäre es allerdings günstiger, die in der Datenbank zu jedem Menüeintrag gespeicherte VIEW_ID als Eigenschaftswert eines Ereignisobjekt auszugeben. Darüberhinaus muss das Ereignis das für die TreePanel-Klasse intern durch Klicken auf die JTree-Komponente ausgelöst wird, an externe Komponenten weitergeleitet werden. Das bedeutet, dass die JTree-Komponente eigene Ereignisse auslösen muss, die wiederum ausgelöst werden durch Ereignisse der internen JTree-Komponente.

Der gesamte Prozess der Ereignisauslösung durch die TreePanel-Komponente geschieht nun wie folgt:

1. Ein Mausklick löst ein Ereignis (TreeSelectionEvent) in der internen JTree-Komponente aus
2. Das Ereignis wird durch eine Methode der TreePanel-Komponente behandelt. Die hierbei gewonnenen Pfaddaten eines Menüpunktes müssen auf die aus der Datenbank ermittelten VIEW_IDs des Menüpunktes abgebildet werden.
3. Ein Ereignisobjekt einer neuen Klasse TreePanelEvent wird erzeugt. Das TreePanelEvent-Objekt enthält u.a. als Eigenschaftswert die VIEW_ID des angeklickten Menüpunktes.
4. Das Ereignis wird durch einen Aufruf der folgende Methode ausgelöst:

```
protected void fireItemSelected(TreePanelEvent e) {
    if(treePanelListeners != null) {
        Vector listeners = treePanelListeners;
        int count = listeners.size();
        for (int i = 0; i < count; i++) {
            ((TreePanelListener) listeners.elementAt(i)).ItemSelected(e);
        }
    }
}
```

Codebeispiel 3.5: Auslösen von Ereignissen (TreePanel)

Damit nun eine Klasse die von TreePanel ausgelösten Ereignisse erkennen kann, existiert eine Schnittstellendefinition TreePanelListener. Diese Schnittstelle registriert die Ereignisbehandlung von TreePanel bei der behandelnden Komponente (DataPanel). Dazu ruft die behandelnde Komponente die folgende Methode auf:

```
// treeMenu ist eine Instanz von TreePanel
treeMenu.addTreePanelListener(
    new adstec.Tree.TreePanelListener() {
        public void itemSelected(TreePanelEvent e) {
            treeMenu_itemSelected(e);
        }
    }
);
```

Codebeispiel 3.6: Registrieren der Ereignisbehandlungsmethode (DataPanel)

Hierzu muss die ereignisbehandelnde Komponente (hier: DataPanel) in einer Methode `treeMenu_itemSelected(e){}` definieren, was nach Auslösen des Ereignisses mit den Daten des Ereignisobjektes `e` (im Falle des TreePanels also die `VIEW_ID`) geschehen soll.

3.4.2 Weitere Navigationselemente

Außer der eben beschriebenen Menükomponente sind noch weitere Komponenten notwendig, um kontextsensitiv durch die Datensätze der Datenbank navigieren zu können.

Konventionelle Datenbankanwendungen bieten für die Navigation durch die Daten einer Datenbank lediglich Steuerelemente an, die ein seitenweises bzw. Datensatzweises Blättern durch die Datensätze ermöglichen. Die Steuerelemente bieten meist keine Möglichkeit kontextsensitiv, d.h. nach bestimmten Kriterien wie Datum, Uhrzeit durch die Datenbank zu navigieren.

Hierzu wurde die Komponente `adstec.Navigation.NavigationPanel` entwickelt.



Abbildung 3.12: NavigationPanel

NavigationPanel erlaubt sowohl seitenweises Blättern durch Datensätze, als auch das Navigieren durch Datensätze nach Datum und Datumsbereichen. Durch entsprechende Methoden lässt sich die Komponente programmgesteuert initialisieren, d.h. auf bestimmte Startwerte einstellen. Die Ereignisbehandlung der Komponente entspricht dem bei TreePanel aufgezeigten Vorgehen und soll hier nicht weiter besprochen werden.

Neben der Navigation durch die Datensätze der Datenbank ist es wünschenswert, die Daten nach bestimmten Kriterien zu filtern. Hierzu ist ein weiteres Steuerelement vorgesehen - OptionPanel:

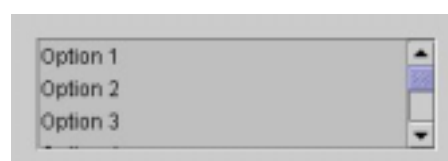


Abbildung 3.13: OptionPanel

Die Einstellungen der Navigationselemente wirken sich bei der Datenbank-Abfrage auf die WHERE-Einschränkungen eines SQL-Kommandos aus. Zu jeder Ansicht auf die Datenbank enthält die Konfigurationsdatenbank hierzu eine generischen SQL-Befehlsvorlage.

In den Tabellen VIEW_DETAILS und OPTION_PANEL sind die für die Auswertung der Datenbank wesentlichen SQL-Befehle und deren variable Bestandteile eingegeben. Die Tabelle TABLE_DETAILS dient dabei als Quelle für die im Befehl anzugebenden Spaltennamen.

Befehlsvorlage können in einer der folgenden Formen angegeben werden:

```
SELECT * FROM tabellenname
```

```
SELECT * FROM tabellenname WHERE spaltenname <...>
```

```
SELECT *  
  FROM tabellenname  
  WHERE spaltenname <...> AND <option>
```

Hier kann anstatt von AND auch eine andere Bedingung (OR, NOT, etc.) gewählt werden.

```
SELECT * FROM tabellenname WHERE <option>
```

```
SELECT <option> FROM tabellenname WHERE spaltenname <...>
```

In der Befehlsvorlage sind folgende Platzhalter vorgesehen:

Platzhalter	Bedeutung	Quelle
*	Spaltenliste der abzufragenden Tabelle	wird aus Einträgen in TABLE_DETAILS erzeugt
<option>	Filter oder alternative Spaltenlisten Darf nur einmal in der Vorlage auftreten !	wird aus Einträgen in OPTION_PANEL erzeugt
spaltenname <...> d.h. spaltenname <day> spaltenname <week> spaltenname <month> spaltenname <year>	wird durch eine Zeitbedingung ersetzt	wird aus den Einstellungen im NavigationPanel erzeugt.

Tabelle 3.2: Platzhalter in der SQL-Befehlsvorlage

Beispiele:

```
SELECT * FROM stoerungen WHERE datum <day>
```

```
SELECT * FROM stoerungen WHERE datum <day> AND <options>
```

Hier werden die Spalteneinträge aus TABLE_DETAILS ermittelt. Der Stern wird anschließend durch die Spaltenliste ersetzt. Aus den Einstellungen des NavigationPanels wird die Zeitkoordinate ermittelt und die Angabe

datum <day> wird durch die Bedingung datum > 'Tag' AND datum < 'Tag + 1' (Beispiel: datum > '27.07.1999' AND datum < '27.07.1999') ersetzt.
<options> wird ersetzt durch die Angaben, die man in der Spalte WHERE_CONSTRAINT in der Tabelle OPTION_PANEL vorgenommen hat.

WHERE_CONSTRAINT	Beispiele
Bedingungen	temperatur < 500 (Vorausgesetzt ist, dass in der abgefragten Tabelle, die durch die unter SET_ID in VIEW_DETAILS gemachten Angaben definiert ist, eine Spalte temperatur existiert) name LIKE '%MAIER%' spaltenname <...> (vgl. Tabelle 1.11)
Feldlisten <option> ersetzt dann * in der Befehlsvorlage	Beladezeit, Entladezeit, Palettensnummer Liste der Werte, durch Komma getrennt

Tabelle 3.3 Angaben für WHERE_CONSTRAINT in OPTION_PANEL

Beispiel für die Anwendung von <option>:

```
SELECT <option> FROM Temperaturen WHERE datum <day>
```

Hier werden die Spalteneinträge aus OPTION_PANEL ermittelt.

Mit Hilfe der Daten der Navigationselemente und der Konfigurationsdatenbank ersetzt der CORBA-Server die Platzhalter durch korrekte Werte.

So wird dann z.B. aus der SQL-Befehlszeilenvorlage:

```
SELECT * FROM tabellenname WHERE spalte3 <week>
```

die Abfrage:

```
SELECT spalte1, spalte2  
FROM tabellenname  
WHERE (spalte3 > '1.3.1999' AND spalte3 < '1.4.1999')
```

3.5 Anwendung der CORBA-Technologie

Wie bereits geschildert, wird die Anwendungslogik der Präsentationsanwendung, durch eine CORBA-Server-Komponente gebildet. Eine im Lastenheft definierte Aufgabe war das System mit dem Produkt Jaguar CTS 3.0 von Sybase zu implementieren. Dies erwies sich in der Praxis jedoch als problematisch, da die Lizenzgebühren für eine Jaguar Laufzeitumgebung sich als zu teuer für das Projekt erwiesen. Erfreulicherweise erwies sich das im Sun JDK 1.2 integrierte CORBA-Produkt JavaIDL als hinreichend leistungsstark, um eine Implementierung auf der Basis dieses Produktes zu realisieren.

3.5.1 Anwendung von JavaIDL

JavaIDL ist ein einfaches sehr im Funktionsumfang eingeschränktes Produkt. Immerhin wird ein Nameservice mitgeliefert, der das Auffinden von Objektreferenzen im Netzwerk erleichtert. Ein unbestreitbarer Vorteil des Produktes ist die Tatsache, dass es bereits in die Java-Laufzeitumgebung integriert ist und daher bei einer Weitergabe des Produktes sofort bereitsteht.

Da für das Anwendungssystem sehr viele Dienste zu implementieren waren, wurden diese aus Gründen der Übersichtlichkeit auf mehrere Funktionsgruppen bzw. Schnittstellen aufgeteilt:

- DisplayManager

Eine Schnittstelle, die Dienste zum Menüaufbau und zur Initialisierung der Anzeige definiert.

- DataManager

Diese Schnittstelle führt den Zugriff auf die Prozessdatenbanken aus.

- ProfileManager

stellt Funktionen zur Benutzerprofilverwaltung zur Verfügung

- StatusManager

erzeugt Reports und Statusmeldungen

Die Schnittstellen, deren Eingabeparameter und Rückgabewerte wurden in IDL definiert.

Beispiel aus der Schnittstellendefinition DisplayManager:

```
// DisplayManager Structures and Interface Definition

struct menuitem {
    long view_id;
    long level;
    string type;
    string label;
};

typedef sequence <menuitem> menu;

struct tabitem {
    string label;
    string type;
    long set_id;
};

typedef sequence <tabitem> tabs;

interface DisplayManager {

    menu getMenu();

    tabs initTabs(
        in long view_id
    );

}; // End DisplayManager
```

Codebeispiel 3.7: IDL-Definition DisplayManager

Anschließend müssen mit dem IDL-Compiler die Java-Implementierungsdateien (Client-Stubs und Server-Skeletons) erzeugt werden:

```
idltojava -fno-cpp -fverbose -p adstec adstec.idl
```

Gemäß der im CORBA-Standard Version 2.0 definierten Java-Sprachabbildung erzeugt der IDL-Compiler aus der interface-Definition eine Java-Schnittstellendefinition, die von einer abstrakten Java-Klasse, dem Server-Skeleton, formal implementiert werden.

Von dieser abstrakten Klasse wurde die eigentliche Implementierung der Schnittstellenklasse DisplayManagerImpl abgeleitet (Vererbungsansatz).

Interessant im Zusammenhang mit der Java-Sprachabbildung ist die Behandlung des IDL-Datentyps struct.

Java kennt im Gegensatz zu C++ diesen Datentyp nicht. Die Abhilfe besteht in der Definition einer Java Klasse:

```
public final class menuitem implements org.omg.CORBA.portable.IDLEntity {
    //      instance variables
    public int view_id;
    public int level;
    public String type;
    public String label;
    //      constructors
    public menuitem() { }
    public menuitem(int __view_id, int __level,
                    String __type, String __label) {
        view_id = __view_id;
        level = __level;
        type = __type;
        label = __label;
    }
}
```

Codebeispiel 3.7: Beispiel: Java-Sprachabbildung des IDL-Datentyps struct

Bei der Schnittstellenimplementierung ist zu berücksichtigen, dass bei der Instanzbildung der Klassen, wie `menuitem`, die Parameter gleich im Constructor-Aufruf angegeben werden sollten.

Das folgende Beispiel aus der DisplayManager-Implementierung soll stellvertretend für die anderen Funktionen aufzeigen, wie innerhalb einer der Schnittstellen-Objektdefinitionen, die Datenbankabfrage (hier: Konfigurationsdatenbank) und die Übermittlung der Daten an Client-Objekte gelöst wurde.

```
//Titel:          adstec Anlagenmonitor
//Version:        0.1
//Copyright:      Copyright (c) 1999
//Autor:          Ralf Engelmann
//Organisation:   Institut für Automatisierungstechnik (IAS),
//                Universität Stuttgart
//                adstec Automation, Daten- und Systemtechnik GmbH
//Beschreibung:   Schnittstellenobjekt für Bildschirmaufbau

package adstec;

import java.sql.*;
import java.util.*;
import adstec.Monitor.*;

public class DisplayManagerImpl extends _DisplayManagerImplBase {
    Properties props;
    adstec.Monitor.menuitem[] sqlmenu;

    public DisplayManagerImpl() {
    }

    public DisplayManagerImpl(Properties p) {

//...
}
```

Codebeispiel 3.8a: Auszug aus der DisplayManager-Implementierung

Wie der obige Codeausschnitt zeigt, wird den Schnittstellenobjekten im Konstruktor ein Property-Objekt übergeben. Dieses Property-Objekt enthält alle Programmparameter, die der instanzbildenden Server-Applikation übergeben wurden, insbesondere die Parameter zum Verbindungsaufbau mit der Konfigurationsdatenbank. Diese Parameter werden im Folgenden zum Aufbau der Datenbankverbindung genutzt:

```
//... Fortsetzung
```

```
props = p;
int maxItems = 1;
int i = 0;
int view;
int level;
String type;
String name;

// DB Connection
try {
    // Class.forName(driver); muss hier nicht mehr ausgeführt werden, da
    // die instanzbildene Klasse adstec.Server, dies schon erledigt hat.
    Connection c = DriverManager.getConnection(
        props.getProperty("DBURL"),
        props.getProperty("DBUsername"),
        props.getProperty("DBPassword"));
    Statement s = c.createStatement();
    // Zahl der Einträge ermitteln
    ResultSet r = s.executeQuery("SELECT COUNT(*) FROM DATABASE_VIEWS");
    r.next();
    maxItems = r.getInt(1);
    sqlmenu = new adstec.Monitor.menuitem[maxItems + 1];
    sqlmenu[0] = new adstec.Monitor.menuitem(0, 0, "V",
        "Kein gültiger Eintrag");

    // Einträge abfragen
    r = s.executeQuery(
        "SELECT VIEW_ID, NODE_LEVEL, NODE_TYPE, NODE_NAME "
        + "FROM DATABASE_VIEWS "
        + "ORDER BY SEQUENCE");
    while (r.next()) {
        i = i + 1;
        view = r.getInt(1);
        level = r.getInt(2);
        type = r.getString(3);
        name = r.getString(4);
        sqlmenu[i] =
            new adstec.Monitor.menuitem(view, level, type, name);
    };
    s.close();
    c.close();
}
catch (Exception e){
    sqlmenu = new adstec.Monitor.menuitem[1];
    sqlmenu[0] =
        new adstec.Monitor.menuitem(0, 1, "V",
            "Menü konnte nicht geladen werden");
    System.err.println("Error: " + e);
    e.printStackTrace();
};
System.out.println( "DisplayManager-Dienste initialisiert" );
}
```

```
public adstec.Monitor.menuitem[] getMenu() {
    System.out.println( "Menuedaten uebertragen" );
    return sqlmenu;
};

public adstec.Monitor.tabitem[] initTabs(int view_id) {
    // ...
};
}
```

Codebeispiel 3.8b: Einlesen der Datenbankdaten in eine 'Struktur'

Wie der obige Codeausschnitt zeigt, werden die Datenbankdaten, die das Menü (TreePanel-Komponente) aufbauen in ein Array mit Elementen vom Typ `adstec.Monitor.menuitem` eingelesen. Dies geschieht immer noch im Konstruktor des Schnittstellenobjektes, da die Menüdaten sich zur Laufzeit des Servers nicht ändern. (Werden diese Daten geändert, muss der CORBA-Server neu gestartet werden.). Die Präsentationskomponente ruft nach dem Erstellen der Objektreferenz auf ein vom CORBA-Server instantiiertes Schnittstellenobjekt der Klasse `DisplayManagerImpl`, die hier dargestellte Methode `getMenu()` auf, die die bereits gelesenen Daten überträgt.

Die Serverinitialisierung und Instanzbildung der Schnittstellenobjekte und Objektreferenzen erfolgt analog zu den in 2.3.2 dargestellten Sachverhalten.

3.5.2 Anwendung von Jaguar

Obwohl Jaguar auf Grund der sehr hohen Lizenzgebühren nicht zum Einsatz kam, soll hier kurz skizziert werden, wie die Entwicklung der Anwendung mit Jaguar vorgenommen werden kann.

Jaguar besitzt einige mächtige Assistenten zur Erstellung von Schnittstellendefinitionen. Leider ist es mit diesen Assistenten nicht möglich Datentypen wie `struct` oder `sequence` zu definieren, man ist auf einfache Datentypen wie `long`, `double` und `string` oder vorgefertigte strukturierte Typen beschränkt.

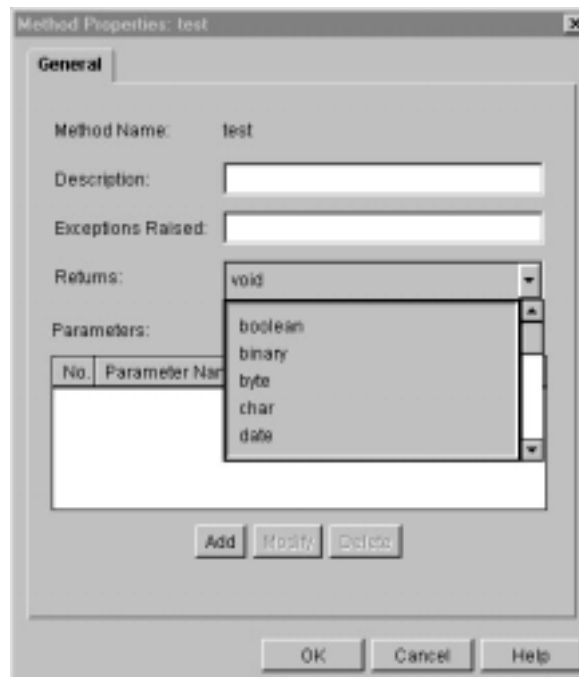


Abbildung 3.14: Jaguar Methoden-Definition

Die Assistenten erlauben nur die Festlegung vordefinierter Datentypen als Methodenparameter

Daher führt auch hier nur der Weg über eine IDL-Definitionsdatei zum Ziel: Öffnet man im so genannten Jaguar-Manager den Zweig Module (in Abbildung 3.15 bereits geschehen), so kann man verschiedene Paketsymbole erkennen. Mit Hilfe eines Kontextmenüs muss man zunächst eine neue IDL-Definition erzeugen. Jaguar besitzt z.T. eine etwas eigenwillige Nomenklatur für IDL-Datentypen: z.B. bezeichnet Jaguar den IDL-Datentyp `module` mit dem Begriff `,package'`, der eigentlich für die Java-Sprachabbildung des IDL-Datentyps `module` steht.

Nachdem man die IDL-Definitionsdatei erzeugt hat kann sie mittels eines weiteren Menüpunktes im Kontextmenü bearbeitet werden (Abbildung 3.15).

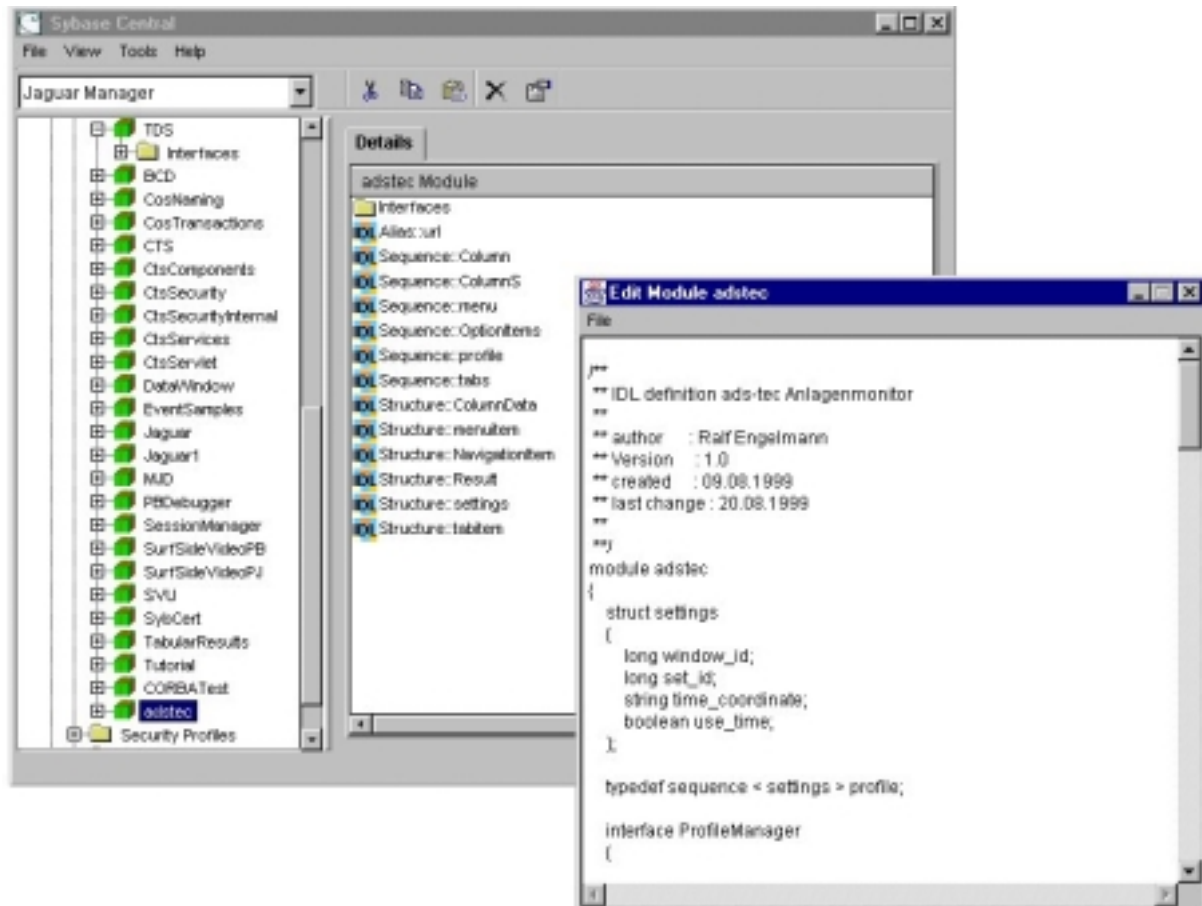


Abbildung 3.15: IDL-Definition in Jaguar

Mit einem weiteren Kontextmenü im Zweig Packages des Jaguar-Managers lassen sich dann die Stubs und Skeletons erzeugen.

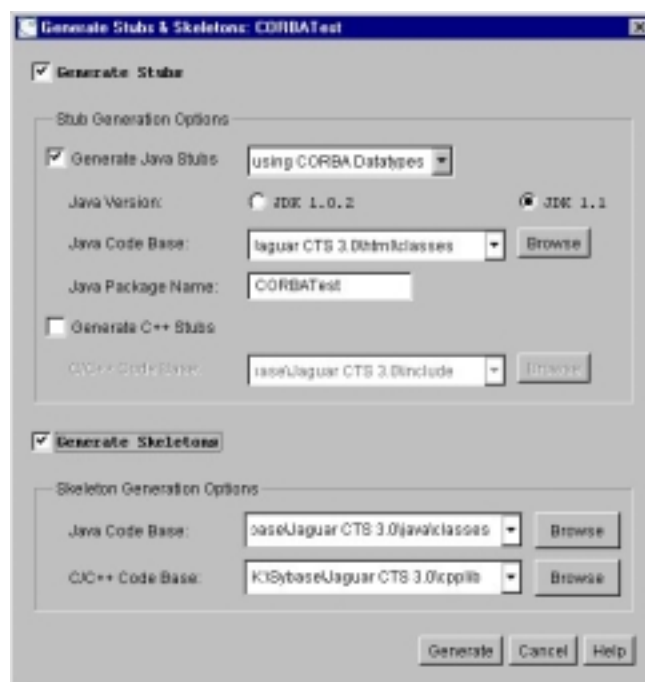


Abbildung 3.16: Erzeugen der Stubs und Skeletons in Jaguar

Die weitere Implementierung der Schnittstellenklassen verläuft analog zu JavaIDL bis auf den Unterschied, dass man keine Serverklasse implementieren muss. Die Instanzen der Schnittstellenklasse bildet und bei einem Nameservice registriert. Als weiterer Unterschied fällt auf, dass für den Server lediglich zwei Klassendateien pro Schnittstelle gebildet werden: eine abstrakte Klasse mit dem Skeleton und eine Beispiel-Schnittstellenimplementierung. Die bei JavaIDL üblichen Schnittstellendefinitionsdateien entfallen, da im Gegensatz zum prozeduralen Ansatz bei JavaIDL, Jaguar ein dynamisches Interface-Repository zur Verfügung stellt, das die Schnittstellendefinitionen persistent, d.h. über einen Jaguar-Server-Neustart hinaus, speichert.

4 Erfahrungsbericht

Die Entwicklung verteilter Anwendungen mit verschiedenen Softwarekomponenten, die zum Teil von Fremdhersteller stammen, stellt erhöhte Anforderungen an den Systementwickler. Dies ist wohl darin begründet, dass der Entwickler sich mit sehr unterschiedlichen Technologien, Programmiersprachen und Betriebssystemplattformen auseinandersetzen muss.

Standardisierte Technologien wie CORBA erleichtern den Entwicklungsprozess, die spätere Weiterentwicklung und Wartung der Systemkomponenten. Durch die Standardisierung wird ein einheitliches Architekturmodell und Programmiersystem möglich, das dem Entwickler erlaubt sich auch bei fremden Softwareprodukten zurechtzufinden.

Leider ist die Programmiersprache Java noch ungenügend standardisiert. Ein Java-Entwickler kann sich nicht darauf verlassen, dass die von ihm entwickelte Anwendung wirklich auf allen Plattformen lauffähig ist. Die Java-Spezifikation erlaubt zwar die Ausführung derselben Binärdatei auf verschiedenen Systemen, dies scheitert in der Regel jedoch daran, dass die für die verschiedenen Systeme verfügbaren Java-Laufzeitumgebungen sich auf unterschiedlichen und z.T nicht untereinander kompatiblen Versionsständen befinden.

Darüberhinaus hemmen zahlreiche Implementierungsfehler in den Java-Laufzeitumgebungen den Entwicklungsprozess. Die in diesem Projekt eingesetzte Java Laufzeitumgebung der Version 1.2.2 zeigte bei der Verwendung der Datenbankfunktionen mit der JDBC-ODBC-Schnittstelle erhebliche Stabilitätsprobleme: Nach mehreren Datenbankzugriffen stürzte die Laufzeitumgebung ab. Die ließ sich nur durch den Einsatz der Vorgängerversion 1.2.1 beheben, die dafür erhebliche Defizite bei der WWW-Browser-Integration aufwies.

Die Entwicklung von Java-Anwendungen, wird sehr vereinfacht durch eine Entwicklungsumgebung, die die Entwicklung von Anwendungen in einer aktuellen Java-Version ausreichend unterstützt. Dies ist bei der Entwicklungsumgebung PowerJ von Sybase leider nicht der Fall. Abgesehen von Fehlern beim visuellen Erstellen von Benutzeroberflächen, bei denen die Komponenten häufig nicht richtig dargestellt werden und das Platzieren von Komponenten sehr fehleranfällig und umständlich erscheint, erlauben die integrierten Assistenten nur einen sehr beschränkten Blick auf die Methoden einer visuellen oder nicht-visuellen Komponente. Ist dem Entwickler die Bedeutung einer Komponenten-Methode oder die für diese Komponente verfügbaren Eingabeparameter und Rückgabewerte unbekannt, hat er die Möglichkeit über einen assistentengestützten aber eher umständlichen Suchprozess an die Informationen des Hilfesystems zu gelangen. Viele Parameter der verfügbaren Assistenten müssen von Hand eingegeben werden, ohne dass der Benutzer über die Richtigkeit der Eingaben eine Kontrollinformation bekommt. Das Auffinden von Fehlern bei der Eingabe wird dadurch unnötig erschwert, dass der Entwickler nicht den kompletten Quelltext einer Klasse in den Editoren angezeigt wird, sondern nur Auszüge aus dem Quelltext. Eine weitere Schwäche zeigen der integrierte Editor und das Projektmanagement darin, dass sie den Import und die Bearbeitung fremder Quelltexte nicht oder nur sehr umständlich zulassen.

Eine weitere Schwäche des Entwicklungssystems ist der von den Assistenten erzeugte generierte Code. Viele Bereiche des für den Entwickler sichtbaren Codes sind als nicht editierbar gekennzeichnet und mit so vielen Kommentaren versehen, dass der Quelltext sehr unübersichtlich wirkt. Überhaupt wird von den integrierten Assistenten der Entwicklungsumgebung ein Framework geschaffen, das sich wohl nur einem sehr geübten PowerJ-Anwender erschließt.

Ein grundsätzliches Problem der Sybase Entwicklungsprodukte ist die Tatsache, dass mit dem aktuellen Stand der Technik verglichen, eher veraltete Produkte verwendet werden. Ein deutliches Indiz dafür ist die mangelhafte Unterstützung des seit rund einem Jahr aktuellen JDK 1.2. Alle Java-Komponenten des Entwicklungspaketes arbeiten auf der Basis des sehr alten JDK Version 1.1.7b. Die Integration des JDK Version 1.2 ist zwar vorhanden, jedoch nur ungenügend in die Entwicklungsumgebung integriert. Zahlreiche Funktionen der IDE, u.a. das Debuggen von Anwendungen, sind bei Verwendung des JDK 1.2 in PowerJ nicht oder nur teilweise verfügbar. Da sich das API (Application Programm Interface) von Java mit der Version 1.2 mittlerweile auch an sehr vielen Stellen der Standardbibliothek geändert hat, ist es wohl wahrscheinlich, dass in absehbarer Zeit viele Anwendungen, die mit PowerJ auf der Basis des alten API gewartet werden müssen, damit Sie auf den neueren Java-Laufzeitumgebungen ausführbar werden.

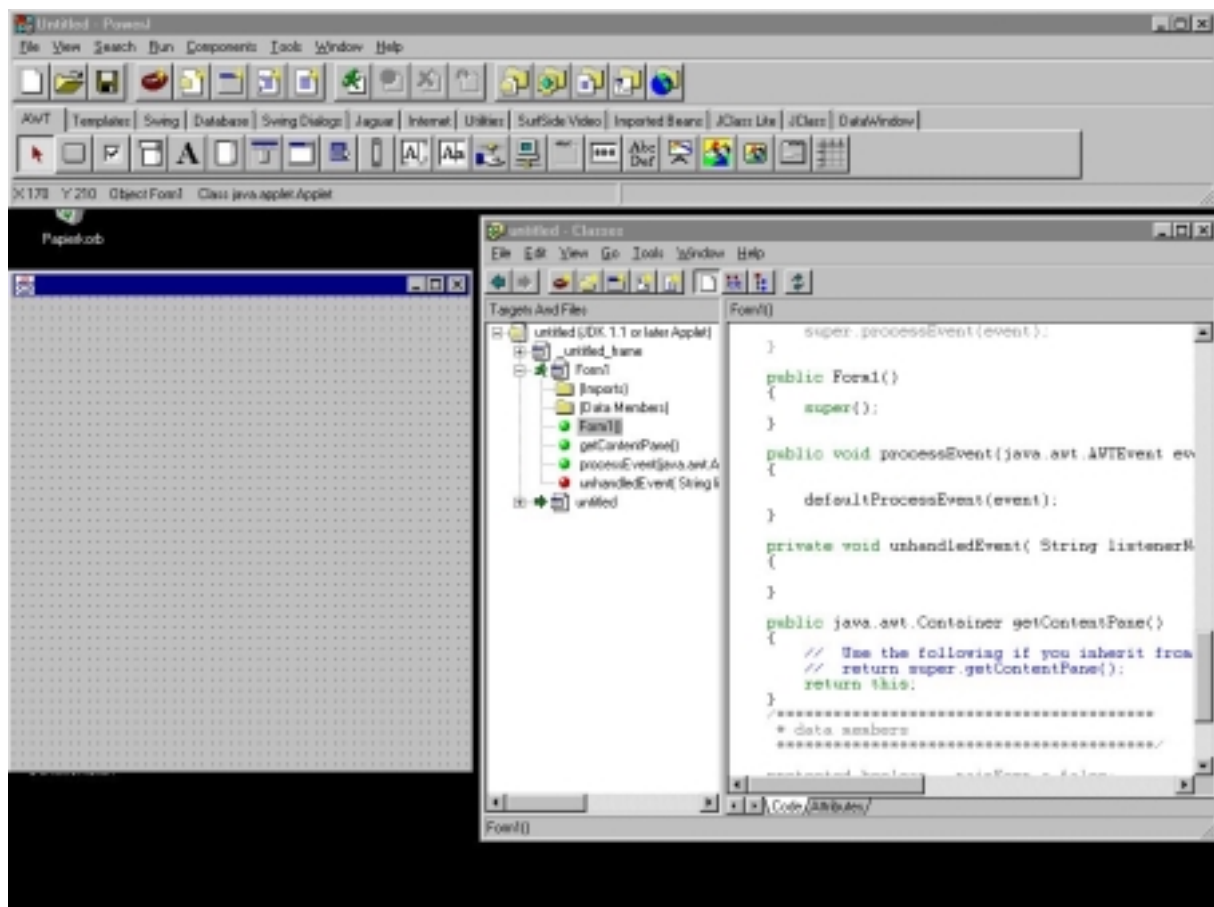


Abbildung 4.1: Sybase PowerJ

Da PowerJ als Entwicklungsumgebung einen eher hemmenden Einfluss auf den Entwicklungsprozess aufwies, wurde für den überwiegenden Teil der Implementierungsphase Inprise JBuilder Version 3.0 verwendet, eine Entwicklungsumgebung, die viele der aufgezeigten Mängel nicht aufweist.

Eine durchaus positive Erfahrung war (abgesehen von der ebenfalls eingesetzten JDK Version 1.1.7) die CORBA-Lösung von Sybase, der Jaguar-Server. Dadurch dass lediglich Schnittstellen-Methoden definiert werden mussten, um eine lauffähige Serveranwendung zu erhalten, ist die Entwicklung von CORBA-Anwendungen recht einfach. Für viele

Anwendungen reichen vermutlich auch die einfachen Assistenten zum Entwurf von CORBA-Schnittstellen-Methoden aus. Gegen den Einsatz von Jaguar als CORBA-Lösung sprechen lediglich die teure Lizenzgebühr und der für viele Anwendungen zu hohe Funktionsumfang.

Denn auch mit dem vergleichsweise einfachen Produkt JavaIDL lassen sich bei beschränkten Ansprüchen CORBA-kompatible Anwendungen entwickeln.

Die Entwicklung von CORBA-Anwendungen ist grundsätzlich nicht schwierig. Die Entwicklung von verteilten Anwendungen an sich ist allerdings sehr aufwändig, da für jede Komponente ein möglichst genauer Entwurf unabdingbar ist. Speziell die Definition der Schnittstellen zwischen den Anwendungskomponenten erfordert ein hohes Maß an Planung und objektorientiertem Design, wenn ein größeres Projekt realisiert werden soll, da bei einer Änderung der Schnittstelle zwischen Anwendungskomponenten, diese unter Umständen vollständig neu definiert und schließlich implementiert werden müssen. Um diese Nachteile zu vermeiden ist, hier ist ein Vorgehensmodell, wie es am IAS für die Studien und Diplomarbeiten realisiert wurde, speziell in den Definitions- und Entwurfsphasen der Anwendung unabdingbar. Es ist dennoch auch nach Definition und Entwurf einer verteilten Anwendung aufwändig, diese in einer Implementierung zu realisieren, da viele Komponenten gleichzeitig zumindest mit einem Teil ihrer Funktionalität schon zur Verfügung stehen müssen, bevor man an die Implementierung und den Test weiterer Komponenten und Funktionen gehen kann.

Der hohe Aufwand bei der Implementierung verteilter Anwendungen birgt allerdings in vieler Hinsicht auch Vorteile: die Anwendungen und Anwendungssysteme bergen in hohem Maß eine hohe Flexibilität beim Erstellen von Softwarelösungen. Dies ist bedingt dadurch, dass viele verschiedene z.T. vorgefertigte und vollständig geprüfte Softwarekomponenten von verschiedenen Herstellern durch den Prozess der Anwendungsentwicklung zur Zusammenarbeit gebracht werden können. Dies setzt aber eine standardisierte Anwendungsarchitektur wie CORBA voraus.

Man könnte vermuten, dass die Aufteilung von Anwendungssystemen in mehrere Komponenten zu einem massiven Performanceverlust führen muss, auf Grund der Tatsache, dass zwischen den Anwendungskomponenten jeweils erst Daten über ein möglicherweise unsicheres und geschwindigkeitsminderndes Netzwerk übertragen werden müssen. Dies war bei diesem Projekt bei dieser Aufgabenstellung nicht zu beobachten. Als geschwindigkeitsmindernde Einflüsse waren im Wesentlichen die Arbeitsgeschwindigkeit der Java-Laufzeitumgebung und die Performance des verwendeten Datenbanksystems auszumachen. Durch die Verwendung von CORBA in diesem Projekt wurde die Geschwindigkeit der Anwendung jedenfalls nicht signifikant begrenzt.

Die Implementierung des in dieser Ausarbeitung beschriebenen Anwendungssystems ist auf Grund des oben beschriebenen Aufwandes noch sehr unvollständig.

Das auf der Basis dieses Produktes entworfene Anwendungssystem hat aber durchaus nach einer Weiterentwicklung bis zur Marktreife das Potenzial für Kunden interessant zu werden. Für dieses System sprechen die zentrale und flexible Konfigurierbarkeit und die im Rahmen des Java Development Kits Version 1.2 zu findende weit gehende Plattformunabhängigkeit.

Anhang

Literaturverzeichnis

- [1] **Zukowski, John** : *Mastering Java 2*, San Francisco - Paris - Düsseldorf: SYBEX Inc, 1998.
- [2] **Lemay, L., Cadenhead R.**: *Teach yourself Java 2 in 21 Days*, Indianapolis, Indiana: SAMS Publishing, 1998.
- [3] **Foley M., McCulley M.**: *JFC Unleashed*, Indianapolis, Indiana: SAMS Publishing, 1998.
- [4] **Rosenberger J.**: *Corba in 14 Tagen*, München - Indianapolis, Indiana: Markt & Technik Buch- und Software Verlag - SAMS Publishing, 1998.
- [5] **Johann, Michael**: *Java mit Swing*, c't Magazin für Computertechnik - Heise Verlag Hannover, Ausgabe 5/98
- [6] **Weber, Marco**: *Teamwork mit Methode*, c't Magazin für Computertechnik - Heise Verlag Hannover, Ausgabe 11/97
- [7] **Schwarz, Martin**: *'Moment, ich verbinde...'*, c't Magazin für Computertechnik - Heise Verlag Hannover, Ausgabe 3/97

Online-Ressourcen

- [8] *CORBA Spezifikation*, <http://www.omg.org>.
- [9] *Java Development Kits und Dokumentation*, <http://www.javasoft.com>.
- [10] *The Free CORBA page, Links zu kostenlosen und kommerziellen CORBA-Produkten*, <http://adams.patriot.net/~tvalesky/freecorba.html>.

Abbildungsverzeichnis

Abbildung 1.1: Zylinderkopfhärteanlage OM600 und Prinzip der Datenerfassung	4
Abbildung 1.2: Programm zur Auswertung der Datenbank.....	5
Abbildung 2.1: Beispieldatenbank Bibliothek	7
Abbildung 2.2: Beispiel für eine redundanzreduzierte Datenbank (2. Normalform)	8
Abbildung 2.3: Struktur der Beispieldatenbank.....	10
Abbildung 2.4: Client-Anwendung (Interbase-Server 5.5, Inprise).....	12
Abbildung 2.5: ODBC-Datenquelleneinrichtung.....	13
Abbildung 2.6: Beispiel für eine 4-tier-Architektur.....	18
Abbildung 2.7: Arbeitsweise des ORB	20
Abbildung 2.8: Übergabe durch Referenz (pass by reference)	23
Abbildung 2.9: Übergabe durch Werte (pass by value).....	24
Abbildung 2.10: Gegenüberstellung Vererbungsansatz / Delegationsansatz	29
Abbildung 2.11: Administration Jaguar-Server	34
Abbildung 3.1: Konfigurationsdatenbank.....	37
Abbildung 3.2: Präsentationskomponente	38
Abbildung 3.3: Architekturmodell der Anwendung	39
Abbildung 3.4: adstec.Applet und adstec.MDIApplet im Vergleich.....	41
Abbildung 3.5: TextPanel.....	45
Abbildung 3.6: ImagePanel	45
Abbildung 3.7: TablePanel	46
Abbildung 3.8: GraphPanel	47
Abbildung 3.9: HtmlPanel.....	47
Abbildung 3.10: CardPanel	48
Abbildung 3.11: TreePanel.....	49
Abbildung 3.12: NavigationPanel	52
Abbildung 3.13: OptionPanel	52
Abbildung 3.14: Jaguar Methoden-Definition	60
Abbildung 3.15: IDL-Definition in Jaguar	61
Abbildung 3.16: Erzeugen der Stubs und Skeletons in Jaguar.....	61
Abbildung 4.1: Sybase PowerJ.....	64

Verzeichnis Codebeispiele und vervollständigte Codetexte

Codebeispiel 2.1: Java-Datenbankfunktionen	15
Codebeispiel 2.2: IDL-Definitionsdatei	26
Codebeispiel 2.2: CORBA-Server (Vererbungsansatz)	28
Codebeispiel 2.3: CORBA-Server mit Delegationsansatz	30
Codebeispiel 2.4: CORBA-Client	31
Codebeispiel 3.2: main()-Methode aus adstec.Monitor.....	40
Codebeispiel 3.3: Statische Methode für das Laden von Bilddateien (aus adstec.Monitor)	43
Codebeispiel 3.4: IDL-Definition DisplayManager	50
Codebeispiel 3.5: Auslösen von Ereignissen (TreePanel).....	51
Codebeispiel 3.6: Registrieren der Ereignisbehandlungsmethode (DataPanel)	51
Codebeispiel 3.7: IDL-Definition DisplayManager	56
Codebeispiel 3.7: Beispiel: Java-Sprachabbildung des IDL-Datentyps struct	57
Codebeispiel 3.8a: Auszug aus der DisplayManager-Implementierung	57
Codebeispiel 3.8b: Einlesen der Datenbankdaten in eine 'Struktur'	59

Codebeispiel 2.3: CORBA-Server mit Delegationsansatz (vollständig)

```
package CORBATest;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

// Hier kann eine beliebige Klasse MathBasic
// verwendet werden um den Dienst zu implementieren

// Änderungen an der Klassenvererbung

class MathBasic
{
    public double add(double m1, double m2)
    {
        return m1 + m2;
    }
}

class MathServantTie extends MathBasic implements _mathOperations
{
}

// Ende der Änderungen an den Klassen

// Die Klasse ServerTie bildet eine Instanz des MathServantTie-
// Objektes, initialisiert und bindet die Instanz an den
// ORB und registriert die Klasse beim Nameservice

public class ServerTie {

    public ServerTie() {
    }
}
```

```
public static void main(String args[])
{
    try{
        // Erzeugt eine Instanz des ORB-Objektes
        // und initialisiert es
        ORB orb = ORB.init(args, null);

        // Erzeugen der Instanz und Binden an ORB
        MathServantTie servant = new MathServantTie(); // Hier wurde
        math helloRef = new _mathTie(servant); // geändert !
        orb.connect(helloRef);

        // Initialisierung des Zugriffs auf Nameservice
        org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
        NamingContext ncRef = NamingContextHelper.narrow(objRef);

        // Registrierung der Objekt Referenz beim Nameservice
        NameComponent nc = new NameComponent("Math", "");
        NameComponent path[] = {nc};
        ncRef.rebind(path, helloRef);

        // Endloswarteschleife
        java.lang.Object sync = new java.lang.Object();
        synchronized (sync) {
            sync.wait();
        }

    } catch (Exception e) {
        System.err.println("ERROR: " + e);
        e.printStackTrace(System.out);
    }
}
}
```

Index

I

1. Normalform 7

2

2. Normalform 8

A

ads-tec 4, 5

Aktivierung 22, 25, 34

Anwendungsarchitektur 2, 16,
36, 37, 65

Anwendungslogik 17, 18, 36,
55

Anwendungsprotokoll 12

Architekturmodell 17, 36

Archivdatenbank 37

Aufgabenstellung 6

B

Basic Object Adapter 25

Benutzerprofildaten 39

Benutzerprofilverwaltung 38,
55

Betriebssystem 4, 6, 20, 41

Betriebssystem 15

BOA 25

C

Callbackmethoden 25

CardPanel 48, 67

Client 12, 16, 17, 19, 20, 21,
22, 25, 26, 27, 30, 31, 35,
56, 57, 67, 68

Compiler 20, 26, 27, 29, 33,
34, 56

CORBA 2, 3, 6, 16, 18, 19, 20,
21, 22, 24, 25, 26, 27, 28,
29, 30, 31, 32, 33, 34, 35,
36, 39, 41, 43, 46, 48, 50,
54, 55, 56, 57, 59, 63, 64,
65, 66, 68, 69

CORBAfacilities 26

CORBAservices 26

D

DataManager 55

DataPanel 41, 51, 52, 68

Datenbank 7

Datenbank-Server 12

Datenbankverbindung 58

Datenzugriffskomponente 17

DCE 21, 32

DCL 11

DCOM 33

DDL 11

Definition 9, 11, 16, 18, 20,
26, 27, 28, 34, 40, 43, 50,
56, 57, 60, 61, 65, 67, 68

Delegationsansatz 29, 30, 67,
68

Dienst 18, 19, 25, 26, 28, 30,
32, 35, 68

Dienste 16, 18, 19, 25, 26, 27,
34, 39, 55, 58

DisplayManager 50, 55, 56,
57, 58, 68

DML 11

E

eine Implementierungssprache
20

Entitäten 9

Entwicklungsumgebung 6, 36,
63, 64

Exceptions 14, 22

F

Fehlerbehandlung 14, 22

Fehlerwahrscheinlichkeit 22

Format 19, 20, 24

Formatübertragung 19, 20, 27

Fremdschlüssel 9

G

GIOP 21

Grammatik 7

GraphPanel 46, 47, 67

H

HtmlPanel 47, 48, 67

I

IDL 20, 26, 27, 29, 33, 34, 35,
50, 55, 56, 57, 60, 61, 67, 68

idltojava 33, 56

IIOP 21, 34

ImagePanel 45, 67

Implementierung 13, 15, 18,
20, 25, 27, 28, 29, 30, 36,
55, 56, 57, 62, 65, 68

Inprise 22

Instanzbildung 44

Integration 13, 33, 36, 39, 63,
64

Integritätsregeln 10

Interface Definition Language
20

Interface-Repository 34, 35,
62

ISDN 5

J

Jaguar 3, 6, 34, 36, 55, 60, 61,
62, 64, 67

Java 1, 2, 3, 4, 6, 12, 13, 14,
15, 16, 19, 20, 21, 27, 28,
29, 33, 36, 37, 39, 41, 42,
43, 44, 55, 56, 57, 60, 63,
64, 65, 66, 68

Java Development Kit 33, 36

Java-Anwendung 36, 41, 44

Java-Applet 2, 4, 36, 41, 42,
43, 44

JavaIDL 3, 27, 33, 55, 62, 65

JClass 46

JDBC 13, 14, 15, 16, 36, 63

JDK 14, 33, 36, 41, 55, 64

JLabel 44, 45, 50

JPanel 44, 50

JScrollPane 44, 50

JTabbedPane 44

JTable 46

JTextArea 44

JTree 50, 51

K

KDE 22

KLGroup 46

Kommunikation 12, 16, 19, 20,
21, 32

Komponente 17, 18, 19, 21,
24, 25, 34, 35, 36, 37, 39,
44, 45, 46, 48, 50, 51, 52,
55, 59, 63, 65

Konfigurationsdaten 37

Konfigurationsdatenbank 37,
38, 39, 48, 53, 54, 57, 58, 67

L

LAN 5

Lastenheft 6

M

MDI 40

Mehrfachvererbung 28

Menükomponente 3, 48, 50, 52

Metadaten 10, 11

Methode 13, 14, 15, 19, 22,
23, 24, 25, 31, 33, 34, 37,
39, 40, 42, 43, 44, 46, 48,
51, 52, 59, 63, 66, 68

Methoden 11, 12, 14, 15, 19,
20, 21, 23, 24, 25, 29, 35,
40, 41, 42, 44, 46, 48, 52,
60, 63, 64, 67

Microsoft 5, 6, 11, 13, 15, 22,
33, 36

Microsoft SQL-Server Version
6.5 6
Multi-tier 16, 17

N

Nameservice 26, 27, 28, 30,
31, 33, 34, 55, 62, 68, 69
Navigation 36, 52
NavigationPanel 52, 53, 67
Netscape 22
Netzwerkstörungen 22
Normalisierung 8

O

Object Request Broker 19
Objektinstanz 27, 28
Objektreferenz 19, 21, 23, 27,
31, 59
Objektreferenzen 19, 22, 26,
27, 41, 55, 59
ODBC 12, 13, 14, 15, 16, 36,
63, 67
OLE 22
OMG 16, 21
OptionPanel 52, 67
ORB 3, 19, 20, 21, 22, 25, 26,
28, 29, 30, 31, 34, 35, 43,
44, 67, 68, 69
OSF 32

P

pass by reference 21, 23, 67
pass by value 21, 24, 67
Performance 5
Performanceprobleme 5
Persistenz 25
Plattformunabhängigkeit 18,
65
Polymorphie 19
PowerJ 36, 46, 63, 64, 67
Präsentationskomponente 17,
36, 38, 39, 48, 59, 67
Primärschlüssel 8
ProfileManager 55
Prozess 19, 23, 24, 51, 65
Prozessdaten 2, 4, 6, 37, 38
Prozessdatenbank 4, 5, 37

Prozesse 17, 18

R

RDBMS 11, 12, 15, 16
Redundanz 7, 8, 9
Referenz 9, 23, 28, 67, 69
referenzielle Integrität 10
relationale Datenbanksysteme
10
relationales
Datenbankmanagementsyste
m 11
relationales Modell 7
Relationen 9
Remote Procedure Call 32
RMI 33, 36
RPC 32

S

Schnittstelle 12, 13, 15, 16, 17,
25, 26, 27, 29, 32, 35, 51,
55, 62, 63, 65
Schnittstellendefinition 25, 26,
29, 51, 56
Schnittstellenklasse 14, 27, 29,
31, 56, 62
Semantik 25
Serialisierung 24
Server 4, 6, 12, 14, 16, 17, 19,
20, 21, 25, 26, 27, 28, 30,
32, 33, 34, 36, 39, 42, 43,
48, 50, 54, 55, 56, 58, 59,
62, 64, 67, 68
Servermaschine 6, 18, 36
Sicherheitsmodell 15, 36, 39,
42
Skeleton 26, 27, 29, 56, 62
Skeletons 26, 56, 61, 67
Socket 32
Solaris 15
Sprachabbildung 20, 27, 56,
57, 60, 68
Sprache 10, 13, 18, 19, 20, 21,
26, 32
SQL 3, 4, 6, 10, 11, 12, 14, 15,
49, 53, 54
SQLServer 36

StatusManager 55
Stored Procedure 11
Sub 26, 27
Subs 26, 35, 56, 61, 67
Sybase PowerJ 6

T

Tabelle 7, 8, 9, 11, 15, 38, 46,
53, 54
TablePanel 46, 67
TCP/IP 21, 32
Textdatei 7
TextPanel 44, 45, 67
Three-tier 2, 17, 18, 36
Tie-Klassen 29
TreePanel 48, 49, 50, 51, 52,
59, 67, 68
TreePanelEvent 51
TreeSelectionEvent 51
Trigger 11

Ü

Übertragungsformat 19
Unix 20
Unmarshaling 19
URL 42, 43, 45, 48

V

Variablenparameter 21
Vererbung 19, 27, 29
Vererbungsansatz 27, 28, 29,
30, 56, 67, 68
Verknüpfung 9
verteilte Anwendung 2, 3, 6,
16, 18, 23, 32, 36, 63, 65
Visibroker 22
Visualisierungsklassen 44

W

Werteparameter 21
Windows 6, 20
WWW-Browser 2, 6, 17, 22,
36, 39, 41, 42, 48, 63

Z

Zylinderkopfhärteanlage 4, 5