Konzeption und Implementierung eines Prototypen zur Präsentation von Versicherungsdiensten im WWW

Diplomarbeit

Universität Rostock, Fachbereich Informatik

vorgelegt von
Bojak, Thoralf
geboren am 7.10.1972 in Greifswald

Betreuer: Prof. Dr. Andreas Heuer
Prof. Dr. Peter Forbrig
Dipl.-Inf. Jürgen Schlegelmilch

Abgabedatum: 15.07.1998

Danksagung

Ich möchte mich bei Prof. Dr. Andreas Heuer für die hilfreiche Kritik sowie bei Prof. Dr. Peter Forbig für das Erstellen des Gutachtens bedanken.

Ein besonderes Dankeschön gilt Dipl.-Inf. Jürgen Schlegelmilch für die geduldige Betreuung und fachlichen Hinweise. Viele Nerven hat er sicherlich beim stetigen Kampf gegen Softwareprobleme lassen müssen.

Zusammenfassung

Diese Diplomarbeit ist ein Bestandteil einer Kooperation zwischen der AVSG, der TU Ilmenau (FB Telematik) und des Fachbereiches Informatik der Universität Rostock zur Erstellung eines Prototypen, mit dessen Hilfe sich Kunden im WWW Angebote für eine Kapital- und Lebensversicherung der Gothaer Versicherungen a.G. erstellen lassen können. Die Angebote werden in einer Datenbank verwaltet und automatisch an zuständige Sachbearbeiter weitergeleitet. Diese Arbeit beschäftigt sich mit der Datenbankanbindung und dem Weiterleiten der Angebote.

Es erfolgt ein Entwurf eines Datenmodells zur Darstellung der relevanten Informationen einer Kapital- und Lebensversicherung. Dieses Datenmodell bildet die Grundlage zur Definition einer Java-Schnittstelle, mit der auf eine relationale und objektorientierte Datenbank zugegriffen wird. Weiterhin werden Möglichkeiten zum automatischen Weiterleiten der Angebote an die zuständigen Sachbearbeiter diskutiert.

Abstract

This dissertation for a M. S. degree is a part of a co-operative project among AVSG, TU Ilmenau (Department Telematik) and the University of Rostock (Departement of Computer Science). The objective of this project is to develop a prototype offer for life insurance by an insurance company, Gothaer Versicherungen a.G., via the WWW. The offer forms are to be stored in a database and automatically forwarded to employees responsible for preparing the offers. This dissertation covers development of the database and the software to forward the offers.

In this paper a design for a data model describing the relevant infor-

mation needed to generate a life insurance offer is presented. This model is used to define an interface (written in Java) to access a relational and an object-oriented database system. In addition, possible approaches to forwarding the offers to employees for processing are discussed.

Schlüsselworte

Kapital- und Lebensversicherung, objektorientierter Entwurf, Java, JDBC, Java-Binding, Relationenmodell, DB2, objektorientiertes Datenbanksystem, O_2, Datenbankzugriffe per WWW, CGI, O_2 Web, ECA-Regeln, Aktive Datenbanksysteme

CR-Klassifikation

D.1.5	Object-oriented Programming
J.1.7	Marketing
H.2.1	Data models
H.2.2	Access methods
H.2.3	Query languages
H.2.8	Database Applications
H.4.3	Electronic mail

Inhaltsverzeichnis

1 Einleitung und Motivation **11**

1.1 Motivation 11

1.2 Die erwartete Leistung des Prototypen 12

1.3 Aufgabenbereiche 13

1.4 Zielsetzung dieser Arbeit 14

1.5 Die verwendeten Systeme 15

1.5.1 Die Datenbanksysteme 15

1.5.2 Java 15

1.6 Der Aufbau dieser Arbeit 16

2 Einführung in die Kapital- und Lebensversicherung **19**

2.1 Allgemeine Begriffe 19

2.2 Der Tarif D4 20

2.2.1 Die Versicherungsdauer 21

2.2.2 Die Todesfallsumme TFS 21

2.2.3 Die Beitragszahlung 22

2.2.4 Die Dynamik 23

2.2.5 Das Zuzahlungsrecht 23

2.2.6 Zusatzversicherungen 23

2.2.7 Die Erhöhungsoption 24

2.2.8 Die Überschußbeteiligung 25

2.2.9 Die Preisklasse 25

2.3 Fazit 26

3 Der Modellentwurf **27**

3.1 Festlegung der Klassen 27

3.2 Entwurf eines Klassenstrukturmodells 29
3.2.1 Die Attribute 29
3.2.2 Die Beziehungen 31
3.3 Fazit 32

4 Die Java-Schnittstelle 33
4.1 Die Abbildung des Datenmodells in Java-Klassen 33
4.1.1 Die Abbildung der Klassenstruktur 33
4.1.2 Die Abbildung der Beziehungen 36
4.1.3 Die Umsetzung der Methoden 37
4.1.4 Die Schnittstelle als Java-Package 38
4.2 Die Test-Applets 39
4.3 Fazit 41

5 Die Schnittstelle für DB2 43
5.1 Das relationale DBMS DB2 43
5.2 Die Abbildung des Modells in das Relationenmodell 44
5.2.1 Abbildung der Klassen 44
5.2.2 Definition von Schlüsseln 44
5.2.3 Abbildung der Objekt-Komponentenobjekt-Beziehungen 45
5.2.4 Abbildung der Typen 45
5.2.5 Das normalisierte Relationenmodell 46
5.3 Die JDBC-Spezifikation 47
5.3.1 JDBC im Überblick 47
5.3.2 Die JDBC-Architektur 51
5.3.3 JDBC in der praktischen Anwendung 51
5.4 Java-Binding für DB2 57
5.4.1 JDBC-Applikationen und -Applets in DB2 57
5.4.2 UDFs und stored procedures 58
5.5 Fazit 58

6 Die Schnittstelle für O_2 61
6.1 Das Java-Binding von O_2 61
6.1.1 Allgemeine Konzepte des JB von O_2 63
6.2 Fazit 67

7 Die Workflow-Komponente **69**
7.1 Beschreibung der erwarteten Leistung 69
7.2 Realisierungsmöglichkeiten 70
7.2.1 Realisierung durch aktive Komponenten 70
7.2.2 Realisierung durch eine Java-Anwendung 75
7.2.3 Export der Daten aus DB2 78
7.3 Fazit 79

8 Vergleich der beiden Datenbanken **81**
8.1 Datenmodellierung 82
8.1.1 DB2 82
8.1.2 O_2 83
8.1.3 Fazit 85
8.2 Datenbankentwurf 85
8.3 Anfragesprache 87
8.3.1 DB2 87
8.3.2 O_2 87
8.3.3 Fazit 88
8.4 Java-Binding 88
8.4.1 Erforderliche Arbeitsschritte 88
8.4.2 Fazit 89
8.5 Export der Daten 90
8.6 Stabilität der Systeme 90
8.7 Fazit 91

9 Generieren von HTML-Seiten aus einer Datenbank **93**
9.1 CGI 93
9.2 O_2 Web 94
9.3 Fazit 95

10 Schlußbetrachtung **97**
10.1 Die Architektur der Applikation 97
10.2 Zusammenfassung 99
10.3 Ausblicke 99

Abbildungsverzeichnis **103**

Tabellenverzeichnis **105**

A Die Java-Schnittstelle **113**

B Definitionen der Relationen für DB2 **116**

C Die Konfigurations-Datei für den Import der Java-Klassen in O_2 **119**

D Die Implementierung der Methoden für die Datenbankzugriffe **120**

D.1 DB2 120

D.2 O_2 129

E Java E-Mail-Clients **133**

E.1 Die Klasse `jSendEMail` 133

E.1.1 Methodenübersicht 133

E.1.2 Beispiel für die Verwendung 134

E.2 E-Mail mit Attachement versenden 135

Kapitel 1

Einleitung und Motivation

1.1 Motivation

In Deutschland wird erwartet, daß im Jahr 1999 mehr PCs als Fernsehgeräte verkauft werden. Da auch die Zahl der Internet-Anschlüße stetig anwächst, gewinnt das Internet als Kommunikations- und Informationsmedium immer mehr an Bedeutung. Neben E-Mail ist vor allem das World Wide Web (WWW) der Grund, warum das Internet für viele Privatpersonen so attraktiv ist.

Das WWW stellt für Firmen eine sehr kostengünstige Form der Werbung dar. Bisher wurde das WWW von der Gothaer Versicherungen a.G. als nahezu reiner Werbeträger benutzt, in Zukunft soll es auch eine stärkere Beratungs- und Servicefunktion übernehmen. Da bislang wenig Erfahrungen mit dem WWW auf Seiten der Gothaer Versicherungen a.G. gesammelt wurden, soll in Zusammenarbeit mit der AVSG (Allgemeine Versicherungs-Software GmbH), dem Fachbereich Telematik der TU Ilmenau und dem Fachbereich Informatik (Datenbanken und Informationsysteme) der Universität Rostock ein Prototyp einer Anwendung (die im folgenden noch genauer beschrieben wird) erstellt werden, mit dessen Hilfe die Gothaer Versicherungen a.G. ihre Präsenz im WWW ausbauen will. Dabei sind weiterhin Möglichkeiten, die das WWW bietet, auszuleuchten und die Tauglichkeit bestehender Systeme und Verfahren zu testen. Eventuell werden weitere Möglichkeiten aufgezeigt; so könnte ein weiterer Schritt die Verwendung von Datenbanken

im WWW in mobiler Umgebung sein, um die Arbeit der Außendienstmitarbeiter beim Kunden vor Ort zu unterstützen.

Ein weiterer Schritt in die oben aufgezeigte Richtung soll die Entwicklung einer Anwendung sein, mit der sich Interessierte konkrete Angebote einer Kapital- und Lebensversicherung erstellen lassen können. Dies ist Gegenstand dieser Arbeit in Zusammenarbeit mit M. Basler [Bas98].

1.2 Die erwartete Leistung des Prototypen

Ein potentieller Kunde startet mit seinem WWW-Browser eine Java-Anwendung, mit der er sämtliche Parameter einer Kapital- und Lebensversicherung so lange anpassen kann, bis das Angebot den persönlichen Gegebenheiten (Leistung und Kosten) entspricht. Dabei soll er auf intelligente Weise so mit Informationen versorgt werden, daß er auch ohne Vorkenntnisse zur Versicherungsterminologie die einzelnen Etappen der Angebotserstellung nachvollziehen kann und für ihn ein sinnvolles Angebot erstellt wird.

Die dabei entstehenden Angebote sollen in einer Datenbank gespeichert werden, um die Daten effizient verwalten zu können. Außerdem ergeben sich so vielfältige Zugriffsmöglichkeiten durch Außendienstmitarbeiter über das WWW. Bedingung für eine Abspeicherung der Daten ist das Einverständnis des Kunden, der damit Interesse an der Kontaktaufnahme zu einem Außendienstmitarbeiter bekundet, denn ein möglicher Vertragsabschluß wird in Verbindung mit einer tiefergehenden Beratung von einem Außendienstmitarbeiter durchgeführt. Vorstellbar ist weiterhin ein Mechanismus, der den entsprechenden Außendienstmitarbeiter automatisch benachrichtigt, wenn Angebote abgespeichert wurden.

Diese gesammelten Daten können ferner die Basis für Analysezwecke zur Akzeptanz der Produkte der Gothaer Versicherungen a.G. sein. Auch für andere Marktforschungsschwerpunkte (wie z.B. die wirtschaftliche Situation bestimmter Zielgruppen) lassen sich Rückschlüsse ziehen.

1.3 Aufgabenbereiche

Bei der Erstellung dieses Prototypen genießt die Darlegung des Machbaren hohe Priorität. Weiterhin sind Kosten und Nutzen aufzuzeigen.

Die Arbeit zerfällt in mehrere Teile:

- Analyse der Informationen:
  Aufbauend auf der Auswahl der verwendeten Informationen muß ein Informationsmodell erstellt werden. Dabei ist auf Vollständigkeit und Abgeschlossenheit ebenso wie auf Verständlichkeit zu achten; auch rechtliche Aspekte sind, in Absprache mit dem Projektpartner, zu berücksichtigen.

- Entwurf der Benutzeroberfläche:
  Die Informationen sind entsprechend dem Informationsmodell strukturiert und in angemessenen Portionen in HTML-Seiten darzustellen. Für die Verknüpfungen der einzelnen Seiten muß eine Benutzerführung konzipiert werden, die die Entscheidungsfindung unterstützt.

- Anbindung an das Datenbankmanagementsystem (kurz DBMS):
  Die Daten sollen in einer Datenbank gehalten werden. Da bei der AVSG das relationale Datenbanksystem DB2 eingesetzt wird, ist dieses System zu verwenden. Zum Vergleich soll auch ein objektorientiertes Datenbanksystem eingesetzt werden, die Wahl fiel dabei auf O_2. Um diesen Teilbereich zu kapseln, ist eine einheitliche Schnittstelle zu definieren, die es der Anwendung erlaubt, in Unabhängigkeit vom verwendeten Datenbanksystem in immer gleicher Weise auf die Datenbank zuzugreifen. Die Anbindung zerfällt in mehrere Teilaufgaben:

  1. Für beide Systeme ist ein Datenbankentwurf für das Informationsmodell aus der ersten Teilaufgabe durchzuführen.
  2. Danach muß die einheitliche Schnittstelle für die beiden Datenbanksysteme implementiert werden. Alternativ ist zu untersuchen, ob die HTML-Seiten der Oberfläche aus der Datenbank generiert werden können, um den Wartungsaufwand zu reduzieren.

- Workflow:
  Mit der WWW-Schnittstelle eingegangene Benutzeranfragen müssen in den Gothaer Versicherungen a.G. weiterbearbeitet werden. Dazu ist ein Mechanismus zu entwickeln, der die Daten auf Wunsch oder automatisch zu konfigurierbaren Zeitpunkten an wählbare Mitarbeiter weiterleitet. Auch die Möglichkeit eines Exports der in der Datenbank enthaltenen Informationen in Dateien soll untersucht werden.

Die aufgeführten Teilbereiche lassen sich in zwei Diplomarbeiten mit dem **Entwurf der Benutzeroberfläche** einerseits und der **Analyse der Informationen** und dem Datenbankanteil mit den Aufgaben **Datenbankanbindung** sowie **Workflow** andererseits trennen; die Analyse der Informationen ist dabei auch Grundlage für den Entwurf der Benutzeroberfläche.

1.4 Zielsetzung dieser Arbeit

In dieser Diplomarbeit sollen die genannten Teilaufgaben **Analyse der Informationen** und der Datenbankanteil mit den Aufgaben **Datenbankanbindung** sowie **Workflow** bearbeitet werden. Dabei strukturiert sich diese Arbeit in folgende Arbeitsschritte:

- Zum umzusetzenden Versicherungstarif werden alle Informationen und deren Zusammenhänge gesammelt. Dabei erfolgt unter Umständen eine Auswahl, welche Informationen für den Prototypen ausschlaggebend sind.

- Nach der Analyse sind die ausgewählten Informationen in einem allgemeinen Datenmodell zu beschreiben.

- Das allgemeine Datenmodell bildet die Grundlage für die Definition einer einheitlichen Schnittstelle. Da Java für die Implementierung gewählt wurde (siehe Abschnitt 1.5.2), muß das allgemeine Datenmodell in Java abgebildet werden.

- Um aus Java heraus über die einheitliche Schnittstelle auf die Datenbanksysteme zugreifen zu können, muß eine Abbildung des Datenmodells von Java auf die Datenmodelle der zu verwendenden DBMS O_2 und DB2 sowie eine Implementierung der Schnittstelle für beide Datenbankensysteme erfolgen.

- Um die Schnittstelle evaluieren zu können, ist eine prototypische Anwendung zu implementieren.

- Die verwendeten DBMS sind hinsichtlich ihrer Eignung zu vergleichen und zu bewerten.

- Möglichkeiten der Umsetzung der Workflow-Komponente sind aufzuzeigen.

Der Entwurf der Benutzeroberfläche und Konzeption der Gesamtanwendung ist Gegenstand der Diplomarbeit [Bas98].

1.5 Die verwendeten Systeme

1.5.1 Die Datenbanksysteme

Da in der Gothaer Versicherungen a.G. das relationale Datenbanksystem DB2 von IBM zum Einsatz kommt, wird DB2 auch für diese Arbeit genutzt werden. Um die Eignung dieses Systems besser einschätzen zu können, wird alternativ auch ein objektorientiertes Datenbanksystem untersucht. Dabei fiel die Wahl auf O_2 von O_2-Technology, da der Fachbereich Informatik der Universität Rostock eine Lizenz für O_2 besitzt. Beide DBMS werden zu einem späteren Zeitpunkt noch genauer vorgestellt.

1.5.2 Java

Als Implementierungssprache wurde Java aus folgenden Gründen gewählt:

- Durch Java-Applets können HTML-Dokumente interaktiv gestaltet werden, daher ist eine vielfältigere Benutzerführung möglich.

- Java ist plattformunabhängig.
  Wenn der Client ein Java-Applet lädt, wird maschinenunabhängiger Bytecode zum Client übertragen, der dort ausgeführt wird.

- Java besitzt ein gutes Sicherheitskonzept für Netzwerkzugriffe.
  Java-Applets können nicht:
  - auf lokale Dateien zugreifen.
  - willkürliche Netzverbindungen herstellen, außer zu dem Host, von dem sie ursprünglich kommen.
  - externe Programme starten.

- Die Java DataBase Connectivity (JDBC) Spezifikation ermöglicht es, auf einfache Weise datenbankunabhängige Java-Clients zu implementieren.

- Die meisten Datenbanken besitzen JDBC-kompatible Treiber.

Innerhalb dieser Arbeit werden zur besseren Verständlichkeit mehrmals Programmausschnitte als Beispiele aufgeführt. Dabei wird auf die Java-Syntax nicht speziell eingegangen, da zahlreiche Literatur zu Java existiert [New97, Mor97, WWW01].

1.6 Der Aufbau dieser Arbeit

Die Teilaufgaben dieser Arbeit können im Abschnitt 1.4 nachgesehen werden.

Diese Arbeit gliedert sich wie folgt:
Im zweiten Kapitel wird in den Bereich der Kapital- und Lebensversicherung eingeführt. Nach der Definition der grundlegenden Begriffe wird der umzusetzende Tarif D4 vorgestellt.

Auf dieser Grundlage wird im dritten Kapitel ein allgemeingültiges Datenmodell definiert, das den in Absprache mit der AVSG gewünschten Umfang des Tarifmodells D4 abbildet.

Das Datenmodell stellt eine Basis für die Definition einer einheitlichen Schnittstelle dar, die den Zugriff auf die DBMS O_2 und DB2 aus Java heraus

ermöglichen soll (Kapitel 4). Diese Java-Schnittstelle wird im fünften bzw. sechsten Kapitel jeweils für DB2 und O_2 umgesetzt und implementiert. Das beinhaltet eine Abbildung der Java-Klassen auf die Datenmodelle der beiden Datenbanken und die Implementierung der Methoden zum Zugriff auf die Datenbanken.

Im Kapitel 7 wird die Workflow-Komponente vorgestellt und deren Realisierungsmöglichkeiten beschrieben.

Beide DBMS werden im Kapitel 8 verglichen und hinsichtlich ihrer Eignung für den Prototypen bewertet.

Im Kapitel 9 werden einige Möglichkeiten der dynamischen Generierung von HTML-Seiten aus der Datenbank aufgezeigt.

Im abschließenden Kapitel 10 werden wesentliche Erkenntnisse zusammengefaßt. Um den Überblick zum Prototypen zu vervollständigen, wird dessen Architektur vorgestellt.

Innerhalb dieser Arbeit werden mehrmals Produkt- und Firmennamen genannt. Es soll darauf hingewiesen werden, daß diese mit Trademarks belegt sind.

Kapitel 2

Einführung in die Kapital- und Lebensversicherung

Im ersten Abschnitt dieses Kapitels sollen zunächst einige Begriffe aus der Welt der Kapital- und Lebensversicherung erläutert werden, um die Beschreibung des konkreten Tarifmodells im sich anschließenden Abschnitt verständlicher zu machen. Die Beschreibung des Tarifes basiert auf [Int97].

2.1 Allgemeine Begriffe

Eine Kapital- und Lebensversicherung kann im allgemeinen folgenden Zwecken dienen:

- Hinterbliebenenversorgung
  Im Falle des Todes soll die Familie finanziell abgesichert sein.

- Berufsunfähigkeitsversorgung
  Tritt Berufsunfähigkeit ein, soll der plötzliche Lohnwegfall durch die Aufstockung der Rente gemindert werden.

- Kapitalbildung
  Für entscheidende Lebensabschnitte (z.B. Gründung einer eigenen Existenz der Kinder, Altersversorgung etc.) soll finanziell vorgesorgt werden.

Die Versicherung wird auf das Leben der **versicherten Person** abgeschlossen. Deren Risikomerkmale (Alter, Geschlecht, Gesundheitszustand, Beruf) haben entscheidenden Einfluß auf die Bestimmung des Beitrages. Je nach Tarifausprägung wird bei Tod, Erleben des Vertragsablaufs bzw. Berufsunfähigkeit der versicherten Person die vereinbarte Versicherungsleistung fällig.

Der **Versicherungsnehmer** schließt den Versicherungsvertrag mit dem Versicherer ab. Er zahlt die Beiträge und hat Anspruch auf die Versicherungsleistung.

Die **Versicherungssumme** ist der vertraglich vereinbarte Geldbetrag, der im Versicherungsfall an den Versicherungsnehmer zur Auszahlung kommen soll. Die Versicherungsleistung wird bei Vertragsende als **Erlebensfallsumme**, bei Tod als **Todesfallsumme** wirksam. Die Versicherungssumme bildet die Basis für diese beiden Summen, die sich je nach Tarifausprägung unterschiedlich entwickeln können.

Bei Kündigung der Versicherung durch den Versicherungsnehmer wird nur ein Teil der eingezahlten Beiträge ausgezahlt. Dieser Wert wird als **Rückkaufswert** bezeichnet.

Weitere Parameter, die eine Lebens- und Kapitalversicherung auszeichnen, werden im folgendem Abschnitt, der den konkreten Tarif D4 beschreibt, aufgelistet und erläutert.

2.2 Der Tarif D4

Eine Kapital- und Lebensversicherung kann je nach Lebenslage des Kunden eine unterschiedliche Zielsetzung haben. Weiterhin sorgen die persönlichen Risikomerkmale und finanzielle Lage der Kunden dafür, daß der Versicherer eine Kapital- und Lebensversicherung als eine große Anzahl von Versicherungen unterschiedlichster Tarifausprägungen anbieten muß, um möglichst alle Zielgruppen bedienen zu können. Auf den Kunden kann das natürlich verwirrend wirken. Weiterhin ist auch die Handhabung und Verwaltung auf Seiten des Versicherers komplex.

Bei der Gothaer Versicherungen a.G. soll jedoch in Zukunft ein Bausteinsystem alle Tarifarten durch einen einzigen Tarif ersetzen. Aus diesem Grund

wurde der Tarif D4 entwickelt, der im wesentlichen alle anderen Tarife abbilden kann. Er ist sehr flexibel, so daß er genau auf die Bedürfnisse des Kunden zugeschnitten werden kann. Somit können viele Produkte mittels eines Tarifes angeboten werden. Der D4 kann unter anderem als Direktversicherung durch Gehaltsumwandlung, zur Anlage der vermögensbildenden Leistungen und zur Gesellschafter-Geschäftsführer-Versorgung verwendet werden. Theoretisch ist auch angedacht, ihn mit Krankenversicherungen etc. zu kombinieren.

Der D4 kann kurz als eine Kapitalversicherung auf den Todes- und Erlebensfall mit variabler Todesfallsumme (kurz TFS), gegen laufende oder abgekürzte Beitragszahlung und mit variablem Auflösungsrecht bezeichnet werden. Was das im einzelnen bedeutet, soll in den folgenden Abschnitten genauer erläutert werden.

Die Bezeichnung D4 ist der interne Name, als Marketing-Name wird die Bezeichnung VarioTime-Police benutzt.

2.2.1 Die Versicherungsdauer

Die Versicherungsdauer besteht aus zwei Phasen: der Grundphase und der Auflösungsphase.

In der Grundphase bleibt die Erlebensfallsumme (kurz EFS) konstant. Die Dauer der Grundphase sollte aus steuerlichen Gegebenheiten mindestens 12 Jahre betragen. Eine Ausnahme bildet der Bereich der betrieblichen Altersversorgung, hier muß die Grundphase nur mindestens 7 Jahre betragen.

In der Auflösungsphase hat der Kunde das Recht, die Versicherung aufzulösen. Anders als in der Grundphase steigt die EFS. Die Dauer der Auflösungphase kann zwischen 0 und 15 Jahren liegen.

Die Dauer der beiden Phasen wird zu Versicherungsbeginn festgelegt.

Das Höchstendalter des Kunden darf dann inklusive der Auflösungsphase maximal 80 Jahre betragen.

2.2.2 Die Todesfallsumme TFS

Die TFS wird immer in Prozent der Versicherungssumme angegeben und entwickelt sich linear von einem im ersten Versicherungsjahr gültigen End-

satz, der zwischen 10% und 200% der Versicherungssumme liegen darf, bis zu einem Endsatz im letzten Jahr der Grundphase, der dann zwischen 100% und 200% der Versicherungssumme liegen darf.

Es kann allerdings zu Beginn der Versicherungsdauer eine Phase mit konstanter TFS vereinbart werden, erst nach dieser Phase steigt oder fällt die TFS auf den gewünschten Endsatz. Die Entwicklung der TFS wird als Leistungsverlauf bezeichnet, der 10 unterschiedliche Möglichkeiten beinhaltet.

Ist die Beitragszahlungsdauer kleiner als die Grundphase, so muß der Anfangssatz mindestens 100% der Basissumme betragen. Ist der Anfangssatz kleiner als 100%, so muß folgendes Verhältnis gelten:

$$\frac{Dauer\ der\ Phase\ mit\ konstanter\ TFS}{Dauer\ der\ Grundphase} \leq Anfangssatz.$$

Das bedeutet, daß z.B. die konstante Dauer maximal 50% der Grundphase betragen darf, wenn der Anfangssatz bei 50% liegt.

In der Auflösungsphase bleibt mindestens die am Ende der Grundphase gültige TFS versichert. Weiterhin gilt ein Prozentsatz, z.Zt. 105%, der die Mindesthöhe der TFS bezogen auf die EFS in der Auflösungsphase festschreibt.

Aus steuerlichen Gründen muß die TFS mindestens 60% der Beitragssumme betragen. Die Differenz zwischen Todes- und Erlebensfallsumme darf maximal 500.000 DM sein. Die Todesfallsummen werden immer auf volle DM aufgerundet.

2.2.3 Die Beitragszahlung

Wie im vorangegangenen Abschnitt schon erwähnt, kann die Beitragszahlungsdauer kürzer als die Dauer der Grundphase sein. Der zu zahlende Beitrag ermittelt sich aus der Versicherungssumme, dem gewählten Verlauf der TFS, der Dauer der Beitragszahlung in der Grundphase und Dauer der Grundphase selbst.

In der Regel bezahlt der Kunde Jahresbeiträge, es ist aber unterjährige Zahlungsweise gegen Zuschläge möglich. Es gibt keine laufende Beitragszahlung in variabler Höhe oder abgesenkte Anfangsbeiträge.

2.2.4 Die Dynamik

Hat der Kunde Dynamik vereinbart, so hat er jedes Jahr einmal das Recht, den Beitrag um einen festgeschriebenen Wert zu erhöhen. Der Wert kann vom Kunden zwischen 5% und 10% gewählt oder an den Zuwachs der gesetzlichen Rentenversicherung (aber mindestens 5%) gekoppelt werden.

Mit Abschluß des Vertrages muß der Kunde einen Rhythmus bestimmen, wann die Dynamik wirksam wird. Der Beitrag wird dann jährlich, alle zwei oder alle drei Jahre erhöht.

Für die Verwendung des erhöhten Beitrages gibt es zwei Möglichkeiten:

- Todes- und Erlebensfall-Dynamik
  Sowohl die TFS als auch die EFS werden über die restliche Dauer der Grundphase im gleichen Verhältnis erhöht.

- Reine Erlebensfalldynamik
  Diese Dynamik ist nur für einen speziellen der zehn Leistungsverläufe möglich. Solange die EFS kleiner als die TFS ist, wird nur die EFS erhöht, die TFS bleibt konstant. Wenn die EFS die TFS übersteigen würde, so werden sie gleichgesetzt und über die restliche Dauer der Grundphase im gleichen Verhältnis erhöht.

Das Erhöhungsrecht endet spätestens ein Jahr, bei Versicherungen ohne Auflösungsrecht sogar spätestens drei Jahre vor dem Ende der Beitragszahlung.

2.2.5 Das Zuzahlungsrecht

Der Kunde hat bei Versicherungen ohne Auflösungsrecht mit laufender Beitragszahlung das Recht, Zuzahlungen zur Abkürzung der Beitragszahlungsdauer oder Erhöhung der Basissumme zu leisten.

2.2.6 Zusatzversicherungen

Zusatzversicherungen können in den D4 eingeschlossen werden.

In der Auflösungsphase erhöhen sich die Leistungen der Zusatzversicherungen weder durch die Erhöhungen der EFS noch durch eine Dynamisie-

rung. Eine Ausnahme bildet die Berufsunfähigkeits-Zusatzversicherung (kurz BUZ), ihre Beitragsfreiheit kann durch die Dynamik mitwachsen.

In dieser Arbeit soll als Zusatzversicherung nur die BUZ berücksichtigt werden. Dabei hat der Kunde die Wahl zwischen zwei Tarifen:

- BJ
  Im Versicherungsfall erfolgt eine Beitragsbefreiung der Haupt- und Zusatzversicherung.

- BJr
  Bis zum Ablauf der Versicherung kann eine zusätzliche Rente vereinbart werden.

2.2.7 Die Erhöhungsoption

Bis zum Alter 65 der versicherten Person darf der Kunde die TFS auf die aktuelle EFS anheben, sogar bis auf 200% der EFS ist eine Anhebung möglich. Dann ist allerdings ab Alter 45 der versicherten Person eine positive Gesundheitsprüfung nötig.

Für die Ausführung der Erhöhungsoption gibt es 4 Optionsanlässe:

- Heirat
- Geburt oder Adaption eines minderjährigen Kindes
- Erwerb eines selbstgenutzten Eigenheimes
- erstmalige Existenzgründung

Die Erhöhungsoption kann vom Kunden innerhalb von 6 Monaten nach Eintritt eines der aufgezählten Optionsanlässe wahrgenommen werden. Das Recht zur Ausübung der Option entfällt, wenn eine mitversicherte BUZ leistungspflichtig ist. Die Option kann nur während der Grundphase ausgeübt werden.

Bei der Ausübung der Option sind die aktuellen steuerlichen Gegebenheiten zu beachten.

2.2.8 Die Überschußbeteiligung

Da die Gothaer Versicherungen a.G. ein Versicherungsverein auf Gegenseitigkeit ist, werden die vom Unternehmen erzielten Überschüsse (abzüglich bestimmter Kosten) nahezu komplett an die Kunden weitergegeben.

Beim Tarif D4 gibt es zwei Überschußbezugssysteme:

- Gewinnsystem BE
  Der jährliche Überschußanteil wird für eine zusätzliche Versicherung verwendet, die vor allem die Versicherungsleistung im Erlebensfall erhöht.

- Gewinnsystem BR
  Der jährliche Überschußanteil wird für eine Reduktion des Beitrages von Beginn an sowie für eine zusätzliche Versicherung verwendet, die vor allem die Versicherungsleistung im Erlebensfall erhöht.

2.2.9 Die Preisklasse

Der D4 ist in allen Preisklassen (unterschiedliche Abschlußkosten mit eventuell weiteren Nachläßen) möglich. Für den Prototypen sollen aber nur folgende vier Preisklassen berücksichtigt werden:

1. Einzeltarif E
   Normale Tarifprämie ohne jegliche Vergünstigung.

2. Kollektivvertrag K
   Kollektivvertrag mit verringerten Abschlußkosten.

3. Kollektivvertrag G
   Kollektivvertrag mit noch geringeren Abschlußkosten als Kollektivvertrag K.

4. Haustarif H
   Tarif mit sehr geringen Abschlußkosten.

2.3 Fazit

Nach der Klärung grundlegender Begriffe und der Analyse der Struktur und Bestandteile des Tarifmodells wird im folgenden Kapitel ein Informationsmodell definiert.

Die AVSG hat sich entschieden, daß einige Bestandteile des Tarifes D4 (z.B. andere Zusatzversicherungen außer der BUZ) im Modell nicht verwirklicht werden müssen. Diese Komponenten sind in der obigen Analyse bereits vernachlässigt worden.

Kapitel 3

Der Modellentwurf

Die Grundlage für die spätere Umsetzung des Tarifmodells D4 in ein Datenbankschema bildet ein Datenmodell, das in diesem Kapitel entworfen wird.

Dem Modellentwurf liegt die Analyse des Tarifmodells D4 im vorangegangenen Kapitel zugrunde.

Da die Applikation in Java implementiert werden soll, bietet sich ein objektorientierter Modellentwurf an. Dazu wurde das Shareware-Tool Object Domain der Firma Object Domain Systems in der Version 1.19a benutzt. Der Entwurf liegt in der Object Modeling Technique-Notation (kurz OMT) vor (siehe Abbildung 3.1).

Nach Definition der benötigten Klassen werden die einzelnen Parameter des D4 hinsichtlich ihrer Bedeutung untersucht und den entsprechenden Klassen hinzugefügt. Das Modell wird dann durch die Definition der Beziehungen zwischen den Klassen komplettiert.

3.1 Festlegung der Klassen

Anhand der Tarifbeschreibungen wurden alle Größen, die eine Lebensversicherung des Tarifmodells D4 enthält, in tabellarischer Form mit ihren Integritätsbedingungen gesammelt.

Dabei zeigte sich, daß Daten zu folgenden Objekten benötigt werden:

- die versicherte Person

VD	Beitrag	TFS	EFS	RKW
1	103,00	52.904,00	52.820,00	996,00
2	106,11	54.475,00	54.268,00	2.094,00
..	..,..	..,..	..,..	..,..
35	232,26	232.130,00	108.826,00	232.130,00

Tabelle 3.1: Beispiel für die Ergebnistabelle eines Angebotes

- der Versicherungsnehmer
- die Tarifdaten der Lebensversicherung

Der Versicherungsnehmer muß zwar nicht die versicherte Person sein, seine Daten spielen allerdings für eine Angebotseinholung keine Rolle. Um ein Angebot zu erstellen, reichen die Risikomerkmale der versicherten Person aus. Wer den Beitrag bezahlt und Anspruch auf die Versicherungsleistung hat, ist zu diesem Zeitpunkt belanglos.

Innerhalb der Tarifdaten der Lebensversicherung gibt es Größen, die unabhängig von der Tarifausprägung in allen Lebensversicherungen einer Tarifgeneration gleich sind, zumindest gilt dies für Vertragsabschlüsse im gleichen Zeitraum. Sie stellen auch in gewisser Weise Rahmenbedingungen seitens des Versicherers dar. Deshalb wurden diese Größen herausgelöst, sie sollen eine eigene Klasse `Kenngröße` bilden. Diese Kenngrößen sollen bei Start der Applikation aus der Datenbank gelesen werden. Durch Änderung von steuerlichen Gegebenheiten können sich diese Größen ändern. Die Änderung der Kenngrößen ist dann nur in der Datenbank zu vollziehen, die Applikation muß nicht angepaßt werden.

Zum Vorgang der Angebotseinholung gehört auch ein Ergebnis: Eine Tabelle soll Auskunft darüber geben, wie sich der Beitrag, der Rückkaufswert (in Tabelle 3.1 RKW), die TFS und EFS im Laufe der Versicherungsdauer (in Tabelle 3.1 VD) entwickeln.

Die Zahlen der Beispieltabelle 3.1 sind frei erdacht, es steckt keine Berechnung nach den Tarifmaßstäben seitens der Gothaer Versicherungen a.G. dahinter.

Im Tarifmodell des D4 existiert eine große Anzahl von Parametern, die allerdings für den Zweck einer Angebotseinholung per WWW keine Bedeutung haben und aus diesem Grund vernachlässigt werden können. Zum Teil sind diese deshalb in der Beschreibung des Tarifes D4 nicht genannt worden. Weiterhin kann auf die Erhöhungsoption verzichtet werden, sie ist erst bei laufendem Vertrag von Bedeutung.

Im Gegensatz zur ersten Grobeinschätzung (siehe oben) besteht das allgemeine Datenmodell, das für die geforderten Zwecke ausreichend ist, also aus folgenden vier Klassen:

- versicherte Person
- Tarifdaten
- Kenngrößen
- Ergebnis

Der Aufbau der Klassen und ihre Beziehungen untereinander werden im folgenden Abschnitt definiert.

3.2 Entwurf eines Klassenstrukturmodells

3.2.1 Die Attribute

Anhand der Tarifbeschreibung und eines gültigen Antrags-Formulars wurden in Absprache mit der AVSG Attribute festgelegt, die für die Applikation relevant bzw. irrelevant sind. Nach Streichung der irrelevanten Attribute wurden die übrigen Attribute den entsprechenden Klassen `Person` und `Tarifdaten` zugeordnet, sie können der Abbildung 3.1 entnommen werden.

Die Wahl der Namen für die Klassen und Attribute ist in Absprache mit M. Basler [Bas98] getroffen worden (so wurde aus `Tarifdaten` einfach `Tarif` und aus `Kenngrößen` wurde `Kenngroessen`).

Die Attribute der Klassen `Person` und `Tarif` sind in der Tarifbeschreibung erläutert worden. Neu sind in dieser Klasse die Attribute `Berechnungsart`

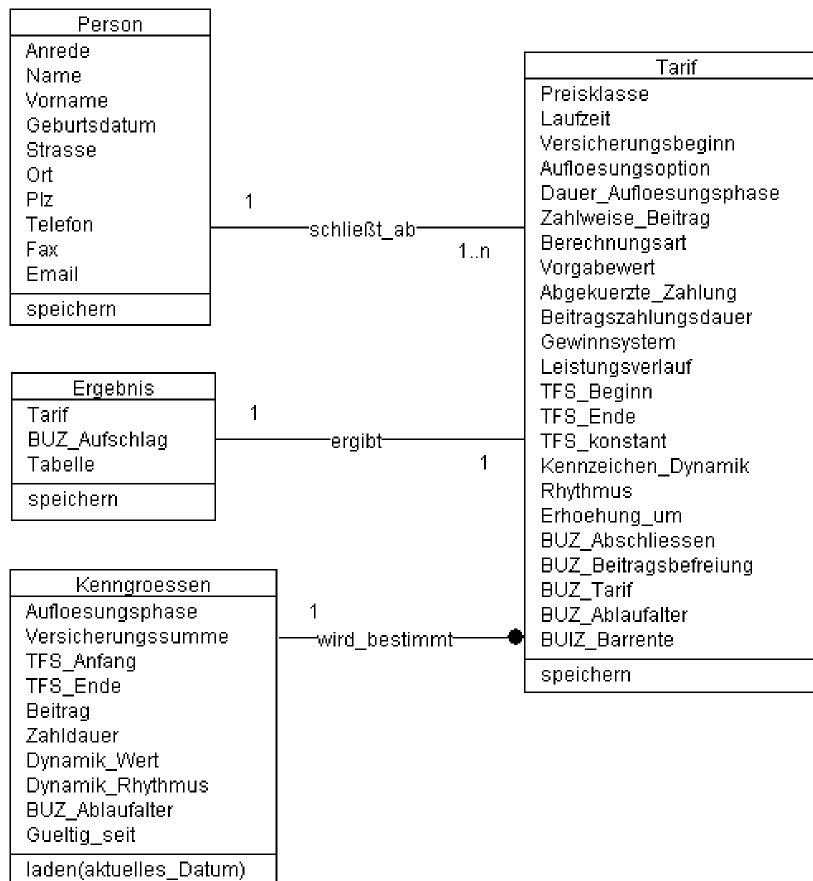

Abbildung 3.1: Das Modell des D4

und `Vorgabewert`, weil die Angebotserstellung auf zwei unterschiedlichen Wegen erfolgen kann. Entweder gibt der Anwender eine Versicherungssumme vor und in Abhängigkeit aller Parameter wird der Beitrag berechnet, oder der Anwender gibt einen gewünschten Beitrag vor, der dann zur Berechnung der Versicherungssumme führt. Somit ermöglicht das Attribut `Berechnungsart` die Interpretation des Vorgabewertes. Die Methode `speichern` soll in beiden Klassen das jeweilige Objekt in der Datenbank persistent machen.

Aus den gesammelten Integrationsbedingungen aus der Analyse des D4 wurden einige ausgesucht (in Absprache mit der AVSG und [Bas98]), die den gültigen Bereich von Eingangsparametern durch ein Intervall festlegen. Die damit festgelegten Attribute werden der Klasse `Kenngroessen` zugeordnet. Die Namen der Attribute entsprechen den jeweiligen Eingangsparametern. Die restlichen Integrationsbedingungen wurden hier nicht berücksichtigt, um das Modell und die spätere Nutzung der Klasse `Kenngroessen` durch die Applikation nicht zu sehr zu komplizieren.

Weiterhin ist ein zusätzliches Attribut `Gueltig_seit` dazugekommen, um bei Erstellung eines Angebotes die aktuelle Version (eigentlich das „aktuelle" Objekt) der Kenngrößen bestimmen zu können. Es könnte sein, daß auch innerhalb einer Tarifgeneration Änderungen dieser Kenngrößen erfolgen. Die Methode `laden` soll über den Parameter `aktuelles_Datum` das gültige Objekt aus der Datenbank holen.

Die Klasse `Ergebnis` hat drei Attribute erhalten. Das Attribut `Tarif` soll die gängige Tarifbezeichnung (eine Zeichenkette aus Großbuchstaben und Zahlen) aufnehmen, die die generelle Tarifausprägung charakterisiert. Wird der Abschluß einer BUZ gewünscht, wird der Beitrag für die BUZ berechnet und als Aufschlag zum bisherigen Beitrag addiert. Das Attribut `BUZ_Aufschlag` nimmt diesen Wert auf. Der Parameter `Tabelle` nimmt die Werte der Ergebnistabelle auf. Die Methode `speichern` soll das Objekt in der Datenbank persistent machen.

3.2.2 Die Beziehungen

Wie der Abbildung 3.1 zu entnehmen ist, wurden drei Beziehungen definiert.

- Die Beziehung `schließt_ab`

Eine Person kann sich mehrere Angebote erstellen lassen, somit kann die gleiche Personen mehrere Tarifdaten eingeben. Ein Objekt der Klasse `Tarif` kann aber nur genau einem Objekt der Klasse `Person` zugeordnet werden.

- Die Beziehung `ergibt`
  Es besteht eine 1:1-Beziehung zwischen einem Ergebnis-Objekt und einem Tarif-Objekt.

- Die Beziehung `wird_bestimmt`
  Tarifdaten unterstehen immer genau einer (der akuellen) Instanz der Klasse `Kenngroessen`. Wird im Gültigkeitszeitraum einer dieser Instanz kein Angebot erstellt, hat diese Instanz auch keine Beziehung zu Objekten der Klasse `Tarif`.

3.3 Fazit

Mit der Definition des Datenmodells stehen die Struktur und die Parameter der Daten fest, die die Applikation benötigt und die in einer Datenbank gehalten werden sollen. Dabei wurde nicht das gesamte Tarifmodell D4 abgebildet, sondern es entstand ein kompakteres Modell, das nach Meinung der AVSG einen ausreichenden Umfang für eine Angebotserstellung per WWW besitzt.

Dieses Datenmodell wird die Grundlage bilden, wenn im nächsten Kapitel die Java-Schnittstelle definiert wird.

Kapitel 4

Die Java-Schnittstelle

In diesem Kapitel soll auf der Definition des Datenmodells aufbauend eine Java-Schnittstelle erstellt werden. Dazu muß das Datenmodell in Java-Klassen abgebildet werden. Diese Schnittstelle wird dann für die beiden DBMS jeweils getrennt implementiert.

4.1 Die Abbildung des Datenmodells in Java-Klassen

Bei der Abbildung des Datenmodells in Java wurde in den folgenden drei Etappen vorgegangen.

1. Abbildung der Klassenstruktur ohne Methoden
2. Abbildung der Beziehungen
3. Umsetzung der Methoden

Warum die Methoden im ersten Schritt zunächst weggelassen wurden, wird im Abschnitt 4.1.3 geklärt.

4.1.1 Die Abbildung der Klassenstruktur

Die Klassen des Datenmodells (siehe Abbildung 3.1) werden mit gleichen Klassen- und Attributnamen in Java als öffentliche Klassen deklariert.

Dabei müssen auch die Typen festgelegt werden. Für die Abbildung sämtlicher Attribute reichen vier Java-Typen:

- `String`
  für alle Zeichenketten
- `int`
  für alle Zahlen, die größere Werte als 127 annehmen können
- `byte`
  für alle Zahlen, die niemals größer als 127 werden;
  für Aufzählungstypen (siehe z.B. Attribut `Anrede` der Klasse `Person`)
- `boolean`
  für alle booleschen Werte.

Java besitzt auch eine Klasse zur Aufnahme von Datumsangaben, die für die Abbildung des SQL-Typen `Date` geeignet ist. Da aber auch DB2 das Datum als Zeichenkette abspeichert, und aus Zeichenketten jederzeit in Java Datumsangaben gewonnen werden können, werden auch innerhalb des Prototypen die Datumsangaben als Zeichenkette verarbeitet.

Das folgende Beispiel der Umsetzung der Klasse `Person` illustriert die erläuterte Vorgehensweise.

```
public class Person {
  public byte      Anrede;         // 0=Herr, 1=Frau, 2=Firma
  public String    Name;
  public String    Vorname;
  public String    Geburtsdatum;
  public byte      Geschlecht;     // 0=maennlich, 1=weiblich
  public String    Strasse;        // incl. Hausnr.
  public String    Ort;
  public String    Plz;
  public String    Telefon;
  public String    Fax;
  public String    Email;
}
```

Die Definition aller Klassen kann im Anhang A nachgesehen werden.

Im Vergleich zum Datenmodell (siehe Abbildung 3.1) wurde die Klasse `Person` um das Attribut `Geschlecht` erweitert. Es läßt sich eigentlich aus dem Attribut `Anrede` ableiten und dient nur der bequemeren Handhabung bei der Implementierung des Prototypen.

Aus diesem Grund ist auf in Übereinstimmung mit [Bas98] auch die Klasse `Tarif` um zwei Attribute erweitert worden. Die neuen Attribute der Klasse `Tarif` lauten `Eintrittsalter` und `Dynamik`. Das Attribut `Eintrittsalter` läßt sich eigentlich aus der Angabe des Geburtstages und des Versicherungsbeginnes berechnen und bezeichnet das Alter der zu versichernden Person zum Zeitpunkt des Versicherungsbeginnes. Das Attribut `Dynamik` soll anzeigen, ob eine Dynamik vereinbart wurde oder nicht. Es ließe sich auch aus dem Wert des Attributs `Erhoehung_um` ableiten - ist der Wert größer als Null, ist eine Dynamik gewünscht, beträgt der Wert Null, dann ist keine Dynamik gewünscht.

Zur Berechnung der Werte dieser zusätzlichen Attribute wird in den Klassendefinitionen eine Methode `AbleitbareAttribute()` deklariert (siehe Anhang A).

Beim Attribut `Preisklasse` repräsentieren die vier möglichen Buchstaben E, K, G und H die im Abschnitt 2.2.9 vorgestellten Preisklassen. Das Attribut `Gewinnsystem` nimmt die beiden Möglichkeiten der Überschußbeteiligung auf.

Wenn eine BUZ eingeschlossen wird, nimmt das Attribut `BUZ_Tarif` den gewählten Tarif (siehe Abschnitt 2.2.6) auf. Wenn eine BUZ leistungspflichtig werden sollte, hat der Versicherungsnehmer die Wahl, ob er auch weiterhin Beitrag bezahlen will oder nicht. Diese Entscheidung wird im Attribut `BUZ_Beitragsbefreiung` festgehalten. Das Attribut `BUZ_Ablaufalter` bestimmt, bis in welches Alter die BUZ leistungspflichtig sein soll. Dabei kann eine monatliche Rente vereinbart werden (Attribut `BUZ_Barrente`).

Die Attribute der Klasse `Kenngroessen` werden bis auf `Gueltig_seit` als eindimensionales Array definiert, um den kleinsten und größten der mögli-

chen Werte aufnehmen zu können. In der Beschreibung des Tarifemodells D4 wurde erklärt, daß z.B. der Anfangswert der TFS zwischen 10% und 200% der Versicherungssumme liegen darf. Daher soll das entsprechende Attribut folgende Werte enthalten:

```
TFS_Anfang[0] = 10
TFS_Anfang[1] = 200
```

Eine Instanz dieser Klasse muß dann mit den aktuell gültigen Werten in der Datenbank abgelegt werden, bevor die Applikation eingesetzt werden kann.

Auch die Klasse `Ergebnis` läßt sich ohne Probleme als Java-Klasse definieren. Dabei wird das Attribut `Tabelle` als zweidimensionales Array definiert.

4.1.2 Die Abbildung der Beziehungen

Beziehungen sind in Java nur über Komponentenobjekte darstellbar. Deshalb wurde eine weitere Klasse `Angebot` definiert, über die die Beziehungen zwischen den Klassen `Tarif` und `Person` bzw. `Ergebnis` umgesetzt werden können.

Ergänzt wird die Klasse um zwei Attribute: `Angebotsnummer` und `Datum`. Das Attribut `Angebotsnummer` soll eine laufend vergebene Nummer für interne Zwecke aufnehmen; das Attribut `Datum` beinhaltet das Datum der Angebotserstellung.

Damit die Kardinalitäten der Beziehung zwischen den Klassen `Person` und `Tarif` (siehe Abbildung 3.1) gewahrt werden, muß bei der Speicherung der Objekte in eine Datenbank zugesichert werden, daß gleiche Personen auch nur durch eine Instanz der Klasse `Person` dargestellt werden.

Da zu allen eingehenden Tarifdaten immer das Ergebnis neu berechnet wird, sind für die 1:1-Beziehung zwischen der Klasse `Ergebnis` und `Tarif` keine weiteren Randbedingungen zu beachten.

Durch die Funktionsweise der Methode, die das aktuelle `Kenngroessen`-Objekt aus der Datenbank laden soll, und des neuen Attributes `Datum` der Klasse `Angebot` kann auch die dritte Beziehung zwischen `Kenngroessen` und

`Tarif` verwirklicht werden. Eine weitere Voraussetzung ist die Garantie, daß sich alle Instanzen der Klasse `Kenngroessen` im Attribut `Gueltig_seit` voneinander unterscheiden. Damit kann mit der Implementierung der Methode `laden(aktuelles_Datum)` sichergestellt werden, daß genau ein (nämlich das aktuelle) Objekt geladen wird. Übergibt man dieser Methode das Attribut `Datum` der Klasse `Angebot`, kann jederzeit einer Instanz der Klasse `Tarif` das entsprechende `Kenngroessen`-Objekt zugeordnet werden. Auch die andere Richtung wäre möglich: Bestimmt man zu einem `Kenngroessen`-Objekt das nächstältere, können alle Instanzen der Klasse `Angebot` (und damit alle Instanzen der Klasse `Tarif`), deren Datum innerhalb des ermittelten Intervalles liegt, genau einem `Kenngroessen`-Objekt zugeordnet werden.

Um das Bestimmen des nächstälteren Objektes zu vereinfachen, könnte man ein zusätzliches Attribut in die Klasse `Kenngroessen` aufnehmen. Entweder fungiert das neue Attribut als Zähler (die Instanzen wären also durchgehend numeriert) oder das Attribut nimmt immer einen Verweis zum nachfolgenden (und damit nächstältestem) Objekt auf. Da diese Funktionalität nicht gefordert ist, wurde darauf bei der Umsetzung in Java verzichtet.

4.1.3 Die Umsetzung der Methoden

Die Methoden können in der jeweiligen Klasse gekapselt werden. Aber in Hinblick auf Erweiterungen oder Veränderung der Datenbankzugriffe wurde entschieden, eine eigene Klasse `Datenbank` zu definieren, die die Kommunikation mit dem DBMS übernimmt. Bei Änderungswünschen zu den Datenbankzugriffen muß nur die Klassendefinition der Klasse `Datenbank` geändert werden, die anderen Klassen können unverändert bleiben. Dadurch soll Pflege und Wartung erleichtert werden.

Es werden für den Prototypen zwei Methoden benötigt, eine Methode lädt die aktuelle Instanz der `Kenngroessen`, eine weitere speichert das komplette Angebot ab. Mit dem Speichern eines Objektes der Klasse `Angebot` werden auch jeweils die Komponenten-Objekte der Klassen `Person`, `Tarif` und `Ergebnis` gespeichert.

```
public class Datenbank {
   public void saveAngebot(Angebot angebot) {
```

```
    }
    public Kenngroessen getKenngroessen(String d) {
    Kenngroessen kenngroessen = new Kenngroessen();
    return (kenngroessen);
    }
}
```

Die Methoden sollen folgendes leisten:

- `public void saveAngebot(Angebot angebot)`
  Das übergebene `Angebot`-Objekt wird in die Datenbank gespeichert.

- `public Kenngroessen getKenngroessen(String d)`
  Der Parameter `d` enthält das aktuelle Datum, damit das gültige Objekt geladen und zurückgegeben werden kann. Der Verweis auf das Objekt ist leer, wenn ein Fehler aufgetreten ist.

Die Methodenrümpfe sind noch leer, sie werden dann für die jeweilige Datenbank implementiert. Wie die Implementierung der Methoden erfolgt, wird in späteren Abschnitten behandelt.

4.1.4 Die Schnittstelle als Java-Package

Java bietet zwar für Schnittstellen die Möglichkeit der Definition eines Interfaces, aber in einem Interface können nur Konstanten und abstrakte Methoden (nur Methodenköpfe, die Implementierung der Methode bleibt der Klasse vorbehalten, die dieses Interface implementiert) enthalten sein. Daher reichen die Möglichkeiten eines Interfaces für die Umsetzung der oben definierten Klassen nicht aus, Schnittstellen werden daher üblicherweise in einem sogenannten Package (Bündelung von Klassen) definiert. Das Interface ist vielmehr als Ersatz für die Mehrfach-Vererbung zu sehen, denn in Java ist es erlaubt, daß eine Klasse mehrere Interfaces implementiert. Damit muß diese Klasse die gesamte Funktionalität für die in den Interfaces definierten Methoden bereitstellen.

Um für die Applikation die Java-Klassen als Schnittstelle zur Verfügung zu stellen, wurden sie also in einem Package `DB_Zugriff` gebündelt. Dieses

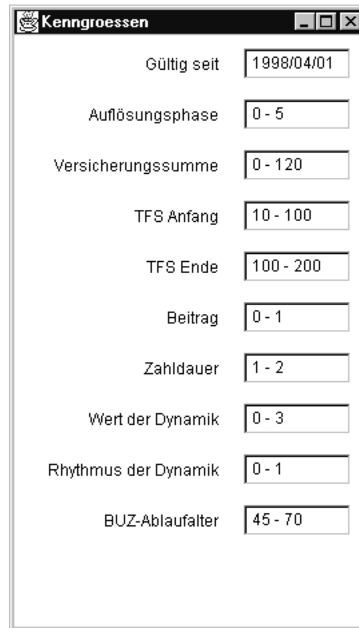

Abbildung 4.1: Ausgabefenster der Kenngrößen

Package muß nun nur bei der Implementierung der Applikation importiert werden,

```
import DB_Zugriff.*;
```

danach können die Klassen, so wie oben beschrieben, benutzt werden.

4.2 Die Test-Applets

Zur Evaluierung der Java-Schnittstelle wurden drei Applets zum Testen der beiden Zugriffsmethoden `getKenngroessen()` und `saveAngebot()` implementiert.

1. Laden der Kenngrößen
   Zum Laden der Kenngrößen wurde ein Applet `LadeKG` implementiert. Es fragt ein Datum ab und ermittelt aus der Datenbank die Instanz, die am Tag des übergebenen Datums aktuell war. Die Ausgabe der Daten erfolgt in einem eigenen Fenster, das in Abbildung 4.1 dargestellt ist.

Abbildung 4.2: Fenster zur Angebotserstellung

2. Speichern eines Angebotes
   Beim Start des Applets `Angebotserstellung` werden in einem Fenster (siehe Abbildung 4.2) alle benötigten Daten zur Angebotserstellung abgefragt. Bei einem Mausklick auf den Button „Berechne Angebot“ werden die Instanzen der Java-Klassen `Person`, `Tarif`, `Ergebnis` und `Angebot` anhand der getätigten Angaben erzeugt. Die Ergebnisdaten werden innerhalb dieses Test-Applets natürlich nicht berechnet, sondern willkürlich aufgefüllt. In einem neuen Fenster wird das Ergebnis präsentiert. In diesem Fenster kann dann mit einem Mausklick auf den Button „Angebot speichern“ das Angebot in die Datenbank gespeichert werden.

3. Laden eines Angebotes
   Nach dem Start des Applets `LadeAngebot` kann dem Fenster eine Angebotsnummer übergeben werden. Das entsprechende Angebot wird daraufhin aus der Datenbank geladen und in einem eigenen Fenster

präsentiert.

Während der Erzeugung der Instanzen und während der Zugriffe auf die Datenbank werden auf der Konsole entsprechende Mitteilungen ausgegeben.

Um die Applets auch in einem Browser ablaufen zu lassen, wurden entsprechende HTML-Seiten erzeugt.

4.3 Fazit

Unter Beachtung von einigen Bedingungen ist die Abbildung des Datenmodells in Java-Klassen ohne Informationsverlust erfolgt. Nach dieser Abbildung steht eine Java-Schnittstelle als Package zur Verfügung. Allerdings sind bisher noch keine Datenbankzugriffe möglich, die im Package deklarierten Methoden sind noch nicht implementiert. Dennoch steht zu diesem Zeitpunkt für die zu entwickelnde Applikation fest, welche Java-Klassen existieren und wie diese Java-Klassen benutzt werden. Diese Schnittstelle ist die Basis der zweiten Diplomarbeit [Bas98] innerhalb dieses Projektes, die sich vornehmlich mit der Implementierung der Applikation beschäftigt und ab diesem Zeitpunkt parallel und unabhängig erfolgen kann.

Die nächsten Arbeitsschritte werden die Implementierungen der beiden Methoden, die die nötigen Datenbankzugriffe realisieren, für die jeweiligen DBMS sein.

Kapitel 5

Die Schnittstelle für DB2

In diesem Kapitel wird erläutert, wie die Java-Schnittstelle für DB2 umgesetzt wird.

Nach einer kurzen Vorstellung von DB2 wird gezeigt, wie die Java-Klassen in das Datenmodell von DB2 überführt werden. Eine Möglichkeit, auf relationale DBMS aus Java zuzugreifen, ist die Verwendung von JDBC. Diese Spezifikation wird im Abschnitt 5.3 in ihren wesentlichen Punkten erläutert, da JDBC für die Zugriffe auf DB2 verwendet wird. Anschließend wird das Java-Binding (kurz JB) von DB2 vorgestellt.

5.1 Das relationale DBMS DB2

Für diese Arbeit wurde DB2 in der Version 5.0 verwendet.

DB2 von IBM ist ein relationales DBMS für die Plattformen Windows, Unix und OS/2. Die Anfragesprache ist SQL mit einigen Spracherweiterungen. Da die Möglichkeiten von SQL nicht immer ausreichend sind, können benutzerdefinierte Funktionen (kurz UDF - user-defined function) erstellt werden. Sie können u.a. in höheren Programmiersprachen implementiert werden und aus einer SQL-Anweisung aufgerufen werden. UDFs liefern einen einzelnen Wert oder eine Tabelle zurück.

Um den Netzverkehr zu verringern, können auch Prozeduren auf Verlangen des Clients direkt auf dem DB2-Server ausgeführt werden, der dann nur das Endergebnis an den Client übermittelt. Diese Prozeduren werden stored

procedures genannt.

Neben Constraints und Triggern kann der Anwender auch benutzerdefinierte Datentypen verwenden. Unter den möglichen Datentypen werden auch Large Objects (kurz LOB) unterstützt.

Das DB2 Call Level Interface (kurz DB2 CLI) ist ein C und C++ API zum Datenbankzugriff auf alle DB2-Server. Dynamische SQL-Anweisungen können ausgeführt werden, indem sie als Argumente bei Funktionsaufrufen übergeben werden. Somit stellt DB2 CLI eine Alternative zu Embedded SQL dar, benötigt aber keine Host-Variablen oder Precompiler.

5.2 Die Abbildung des Modells in das Relationenmodell

Da das Datenmodell objektorientiert entworfen wurde, muß für DB2 eine Abbildung des Datenmodells in ein relationales Modell erfolgen. Dabei wird gleich vom Java-Modell ausgegangen, weil dieses bereits um einige wichtige Attribute erweitert wurde. Die Attribute, die nur Zwischenwerte von Berechnungen enthalten (z.B. das Attribut `Eintrittsalter` der Klasse `Tarif`), werden nicht ins Relationenmodell übernommen.

5.2.1 Abbildung der Klassen

Die Klassen `Person`, `Tarif`, `Kenngroessen`, `Ergebnis` und `Angebot` werden auf Relationen abgebildet. Da das Relationenmodell über keine Methoden im objektorientierten Sinn verfügt, kann auf die Abbildung der Klasse `Datenbank` in eine Relation verzichtet werden.

Die Attributnamen werden genau übernommen.

5.2.2 Definition von Schlüsseln

Für die entstandenen Relationen müssen Schlüssel definiert werden. Die Attribute der Relationen `Person`, `Tarif` und `Ergebnis` eignen sich nicht für die Definition eines kurzen Schlüssels, denn bis auf die Relation `Person` würde der Schlüssel sich aus sämtlichen Attributen der Relation zusammensetzen.

Daher wird in die Relationen ein weiteres Attribut eingefügt, das jedes Tupel der Relation eindeutig kennzeichnet und somit die Funktion eines Schlüssels übernehmen kann. Dieser Vorgang ist eigentlich nur die Abbildung der Objektidentität in einen für die Relation einmalig vorkommenden Attributwert. Dabei kann das Attribut `Angebotsnummer`, das eindeutig ist und daher als Schlüssel in der Relation `Angebot` dienen kann, auch Schlüssel in den Relationen `Tarif` und `Ergebnis` sein.

Da sich eine Person mehrere Angebote erstellen lassen könnte, kann dieser Schlüssel in der Relation `Person` nicht verwendet werden. Daher erhält die Relation `Person` ein neues Attribut, das als Schlüssel fungiert. Die Bildung eines Schlüssels aus den vorhandenen Attributen wäre durchaus denkbar, z.B. könnten aus dem Namen, Vornamen, Geburtsdatum und eventuell aus der Adresse ein Schlüssel definiert werden. Das ist aber ungünstig, weil dieser Schlüssel auch Fremdschlüssel in der Relation `Angebot` ist und somit die Anzahl redundanter Daten erhöht wird.

In der Relation `Kenngroessen` bildet das Attribut `Gueltig_seit` den Schlüssel.

5.2.3 Abbildung der Objekt-Komponentenobjekt-Beziehungen

Die Klasse `Angebot` besitzt Komponentenobjekte (nämlich Instanzen der Klassen `Person`, `Tarif` und `Ergebnis`). Diese Beziehungen werden durch Fremdschlüssel abgebildet.

5.2.4 Abbildung der Typen

Die Abbildung der Typen zur Darstellung von Zeichenketten oder Werten gestaltet sich ohne Probleme.

Fast alle Attribute der Klasse `Kenngroessen` sowie das Attribut `Tabelle` der Klasse `Ergebnis` sind Arrays. DB2 verfügt allerdings über keine entsprechende Domäne, so daß noch entsprechende Umformungen nötig sind:

- Die Attribute der Klasse `Kenngroessen` sind eindimensionale Arrays, die aus zwei Werten bestehen. Deshalb werden die Attribute in jeweils

zwei neue Attribute umgewandelt, die die beiden Werte aufnehmen.

- Das Attribut `Tabelle` der Klasse `Ergebnis` ist ein zweidimensionales Array, das die Werte der Ergebnistabelle (siehe Tabelle 3.1) aufnimmt. Für dieses Array wird eine neue Relation gebildet. Dabei werden die Spaltenbezeichnungen als Attributnamen benutzt. Über den Fremdschlüssel `Angebotsnummer` ist die eindeutige Zuordnung zu einem Tupel der Relation `Ergebnis` möglich.

5.2.5 Das normalisierte Relationenmodell

Aus den vorherigen Schritten ergibt sich folgendes Relationenmodell:

```
Person = {ID_Person, Anrede, Name, Vorname, ...}
Angebot = {Angebotsnummer, ID_Person, Datum}
Tarif = {Angebotsnummer, Preisklasse, Laufzeit,
Eintrittsalter, ...}
Ergebnis = {Angebotsnummer, Tarif, BUZ_Aufschlag}
Ergebnistabelle = {Angebotsnummer, VD, Beitrag, EFS,
TFS, RKW}
Kenngroessen = {Gueltig_seit, Aufloesungsphase1,
Aufloesungsphase2, Versicherungssumme1,
Versicherungssumme2, ...}
```

Die genauen Definitionen der Relationen für DB2 sind dem Anhang B zu entnehmen.

Um einen guten Datenbankentwurf zu sichern, wird ein Relationenmodell in die vierte Normalform gebracht, es gibt dann nur noch die unvermeidbaren Redundanzen bei den Schlüsselwerten. Dabei ist zu beachten, daß die Abhängigkeits- und Verbundtreue beibehalten wird.

Das eben definierte Relationenmodell ist bereits normalisiert, also in der vierten Normalform:

1. Erste Normalform
   Bereits durch die Definition der Java-Klassen existieren aufgrund der

fehlenden Typkonstruktoren nur atomare Attribute. Arrays wurden ebenfalls in atomare Attribute umgewandelt.

2. Zweite Normalform
   Es bestehen keine funktionalen Abhängigkeiten zwischen Teilen eines Schlüssels und weiteren Attributen.
   Eine *funktionale Abhängigkeit* gilt zwischen zwei Attributen X und Y einer Relation, wenn in allen Tupeln der Wert des Attributes X den Attributwert von Y festlegt.

3. Dritte Normalform
   Es bestehen keine transitiven Abhängigkeiten zwischen den Schlüsseln und weiteren Attributen.

4. Vierte Normalform
   Es bestehen keine zwei mehrwertigen Abhängigkeiten zwischen Attributen.
   Unter einer *mehrwertigen Abhängigkeit* zwischen Attributen versteht man den Fall, daß der Wert eines Attributs mehrere Werte eines anderen Attributs festlegt, unabhängig von den restlichen Attributen der Relation.

Weitere Begriffserklärungen und eine umfassende Einführung in die Normalisierung können [Ull88] und [HS95] entnommen werden.

5.3 Die Java DataBase Connectivity (JDBC) Spezifikation

5.3.1 JDBC im Überblick

In diesem Abschnitt soll ein kurzer Überblick zu den wesentlichen Konzepten der JDBC-Spezifikation gegeben werden, weil die Datenbankzugriffe auf DB2 aus Java über JDBC erfolgen. Eine genauere Einführung in JDBC kann [Dic97, HCF97] entnommen werden.

Java eignet sich aufgrund der Plattformunabhängigkeit und seines Sicherheitskonzeptes hervorragend für die Programmierung von Clients für Datenbanken, auf die über das Netzwerk zugegriffen wird.

JDBC ist die Spezifikation einer Schnittstelle zwischen einer Client-Anwendung und einer SQL-Datenbank. Ein JDBC-Treiber übernimmt die gesamte Datenbankanbindung, so daß eine Java-Anwendung mit SQL-Syntax (SQL-92) auf alle Datenbanken, die einen JDBC-kompatiblem Treiber enthalten, zugreifen kann.

Damit ist es möglich, daß Java-Clients unterschiedliche Datenbanken nutzen können, ohne daß der Java-Code angepaßt werden muß.

Bei der Entwicklung von JDBC hat man sich an der ODBC-Spezifikation (Open Database Connectivity) von Microsoft orientiert. JDBC basiert auf dem X/Open Call Level Interface (kurz CLI). Dieses Interface stellt eine Definition der Implementierung von Client/Server-Aktionen für Datenbanksysteme dar.

Die JDBC-Spezifikation umfaßt unter anderem folgende Schwerpunkte:

- Datenbankverbindung
- Senden von SQL-Code an die Datenbank
- Aufnahme der Anfrageergebnisse
- Abbildung von SQL-Datentypen in Java-Typen und umgekehrt
- Transaktionskonzept

Traditionelle Client/Server-Systeme basieren auf der in Abbildung 5.1 dargestellten Zwei-Ebenen-Architektur. Die Java-Applikation kommuniziert direkt mit dem Datenbank-Server. Auf der Client-Seite muß ein JDBC-Treiber vorhanden sein, um dem Server SQL-Anfragen zu senden und die Ergebnisse aufzunehmen. Die Aufbereitung und Visualisierung der Daten erfolgt auf dem Client.

In der Drei-Ebenen-Architektur wird eine weitere Ebene zwischen Client und Server eingeführt: der Application-Server. Dieser sendet die SQL-Anfragen an den Datenbank-Server und nimmt die Ergebnisse auf, die dann an den Client weitergegeben werden.

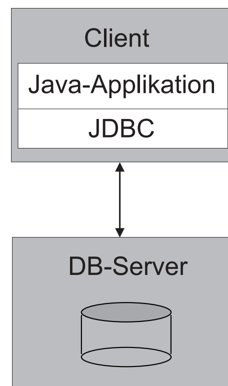

Abbildung 5.1: Zwei-Ebenen-Architektur

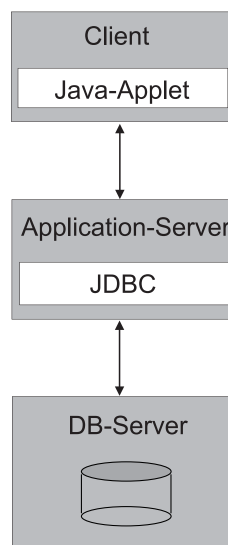

Abbildung 5.2: Drei-Ebenen-Architektur

Die Drei-Ebenen-Architektur besitzt gegenüber der klassischen Zwei-Ebenen-Architektur folgende Vorteile:

- Die Ergebnisse können auf dem Application-Server für die Visualisierung auf dem Client aufbereitet werden. Damit werden die Clients entlastet. Weiterhin muß der Java-Code zur Aufbereitung der Daten nicht mehr zum Client übertragen werden, somit sinkt auch der globale Netzverkehr.

- Der Web-Server muß sich nicht mehr auf dem gleichen Rechner wie der Datenbank-Server befinden.

- Das Applet auf dem Client kann sich aufgrund der Sicherheitsmechanismen des Java-Security-Managers keine native Software herunterladen. Damit können z.B. bestehende C-Programme oder Datenbankbibliotheken mit der Zwei-Ebenen-Architektur nicht eingebunden werden. Diese Software-Komponenten können aber auf dem Application-Server installiert werden, der dann als Java-Applikation auf beliebige Ressourcen des Datenbank-Servers zugreifen kann.

- Um die Performance des Systems (ausgenommen der Präsentation beim Client) zu steigern, müssen nicht sämtliche Clients, sondern nur der Application-Server aufgerüstet werden.

- Auf dem Application-Server bieten sich zusätzliche Möglichkeiten der Integration von weiteren Sicherheitsmechanismen.

Die Nachteile der Drei-Ebenen-Architektur liegen auf der Hand, dürften aber in den meisten Anwendungsbeispielen die Vorteile gegenüber der Zwei-Ebenen-Architektur nicht aufwiegen:

- Der Hardwarebedarf steigt.

- Durch die Entlastung der Clients (und damit Ermöglichung sogenannter thin clients) steigt die Serverlast.

- Der Entwicklungs- und Administrationsaufwand steigt erheblich.

5.3.2 Die JDBC-Architektur

Die JDBC-Treiber werden in vier Kategorien eingeteilt:

- JDBC-ODBC Bridge-Treiber
  JDBC-Aufrufe werden in ODBC-Anfragen auf der Client-Seite übersetzt, deshalb müssen ODBC-Treiber (und oft auch Datenbankbibliotheken) auf dem Client installiert sein. Daher ist die Benutzung der JDBC-ODBC-Bridge nur sinnvoll, wenn nicht direkt, sondern über einen Application-Server auf den Datenbank-Server zugegriffen wird.

- Native API Partial Java-Treiber
  JDBC-Aufrufe werden in native Datenbankmethoden unter Nutzung der herstellerspezifischen Datenbank-API umgesetzt. Auch in diesem Fall müssen einige Datenbank-Bibliotheken auf dem Client installiert sein.

- Net Protocol All-Java-Treiber
  Auf der Client-Seite sind keine weiteren Installationen von nativen Bibliotheken oder nativem Code nötig, der Client kommuniziert über einem Middleware-Server mit der Datenbank. Der Middleware-Server übersetzt die JDBC-Aufrufe in die entsprechenden nativen Datenbankmethoden.

- Native Protocol All-Java-Treiber
  Unter Nutzung der nativen Netzwerkprotokolle der Datenbank-Server können die Clients direkt mit dem Datenbank-Server kommunizieren.

Ab dem JDK 1.1 sind die JDBC-ODBC-Bridge und der JDBC-Treiber-Manager enthalten. Der JDBC-Treiber-Manager ist eine wesentliche Komponente der JDBC-Architektur, er sucht aus allen registrierten JDBC-Treibern denjenigen, mit dessen Hilfe das Applet mit der Datenbank kommunizieren kann.

5.3.3 JDBC in der praktischen Anwendung

In diesem Abschnitt werden einige wesentliche Klassen des JDBC-API vorgestellt. Es wird gezeigt, wie diese Klassen generell benutzt werden. Alle JDBC-

Komponenten sind im Java-Package `java.sql` enthalten.

Die JDBC-Benutzung besteht im Wesentlichen aus folgenden fünf Schritten:

1. Aufbau der Verbindung zur Datenbank
2. Erzeugung einer SQL-Anweisung
3. Ausführung der SQL-Anweisung
4. Auswertung der Anfrageergebnisse
5. Abbau der Verbindung zur Datenbank

Diese Schritte werden im Folgenden genauer erläutert.

Auf- und Abbau der Verbindung zur Datenbank

Zunächst werden mit der Methode `forName()` der Klasse `java.lang.Class` die JDBC-Treiber geladen. Danach kann eine Verbindung zur Datenbank aufgebaut werden.

Die Klasse `java.sql.DriverManager` stellt Mechanismen zur Verwaltung der JDBC-Treiber und Möglichkeiten des Managements von Datenbankverbindungen zur Verfügung. Mit der Methode `getConnection()` wird eine Verbindung hergestellt.

```
Connection connection = DriverManager.getConnection
(url, user, passwd);
```

Der Parameter `url` enthält ein Datenbank-URL (entspricht dem Internet-Standard-Uniform-Ressource-Locator), mit der die Datenbank spezifiziert wird. Ein Datenbank-URL setzt sich wie folgt zusammen:

```
jdbc:<Subprotokoll>:<Subname>.
```

Durch die Parameter `user` und `passwd` erfolgt mit dem Account-Namen des Benutzers und dessen Paßwort die Autorisierung.

Die Methode `close()` schließt die Verbindung.

Informationen zur Datenbank und zum Treiber können mit der Klasse `DatabaseMetaData` ausgelesen werden.

Sollen nur lesende Zugriffe auf die Datenbank vorgenommen werden, empfiehlt es sich, die Verbindung in den effizienteren Read-Only-Modus zu setzen (`Connection.setReadOnly()`).

Erzeugung einer SQL-Anweisung

Innerhalb einer Datenbankverbindung können beliebig viele Datenbank-Operationen ausgeführt werden. Die Anfragen werden in drei Formen unterschieden:

- Einfache SQL-Anweisungen
  Sie bilden das Grundmodell jeder Datenbank-Operation. Anfrageergebnisse werden in Form eines sogenannten Result Set (mehr dazu in einem folgendem Abschnitt) erhalten.

- Prepared Statements
  Gibt es SQL-Anweisungen, die öfter in der gleichen Form, aber mit unterschiedlichen Parametern ausgeführt werden, so bieten sich Prepared Statements an. Das sind bereits vorkompilierte SQL-Anweisungen, deren Parameter vor jeder Ausführung gefüllt werden. Prepared Statements werden erheblich schneller abgearbeitet als einfache SQL-Anweisungen.

- Stored Procedures
  Stored Procedures sind vorkompilierte SQL-Skripte, die in der Datenbank gespeichert sind. Sie müssen nicht aus nur einer Anweisung bestehen, sondern können sich aus relativ komplexen SQL-Prozeduren zusammensetzen. Auch die Definition von Ausgabeparametern ist möglich.

SQL-Anfragen werden mit der Methode `createStatement()` der Klasse `Connection` eingeleitet.

```
Statement stm = connection.createStatement();
stm.executeUpdate (''INSERT INTO table VALUES (x,
y)'');
```

Die Methode `executeUpdate()` sorgt dann für die Ausführung der als String übergebenen SQL-Anweisung; doch dazu im folgenden Abschnitt mehr.

Ausführung einer SQL-Anweisung

Die folgenden Methoden der Klasse `java.sql.Statement` bewirken die Ausführung einer SQL-Anweisung:

- `executeUpdate()`
  Eine Anweisung wird ausgeführt, die keine Ergebnismenge liefert. Neben DDL-Anweisungen (Data Definition Language) wie z.B. `CREATE TABLE` sind das auch `UPDATE`, `INSERT` oder `DELETE`-Anweisungen.

- `executeQuery()`
  Eine Anweisung wird ausgeführt, die ein Result Set (mehr dazu im folgendem Abschnitt) zurückliefert. Gewöhnlich ist das eine `SELECT FROM WHERE`-Anweisung.

- `execute()`
  Diese Methode wird in den seltenen Fällen benutzt, wenn das Statement mehrere Result Sets erzeugt.

Auswertung der Anfrageergebnisse

Die Klasse `java.sql.ResultSet` nimmt die Tupel des Anfrageergebnisses auf und bietet Mechanismen zur Weiterverarbeitung der Daten an.

Das Result Set kann ausschließlich sequentiell abgearbeitet werden. Die Methode `next()` zeigt beim ersten Aufruf auf das erste, bei jedem weiteren Aufruf auf das nächste Tupel der Ergebnismenge. Weiterhin gibt diese Methode eine Variable vom Typ `boolean` zurück, die den Zustand `true` einnimmt, falls noch weitere Tupel in der Ergebnismenge enthalten sind. Wird auf das letzte Tupel gezeigt, wird `false` zurückgeliefert. Eine Rückwärtsbewegung innerhalb der Ergebnismenge ist nicht möglich.

Auf die einzelnen Attribute des aktuellen Tupel kann mit typabhängigen Methoden zugegriffen werden.

```
Statement stm = connection.createStatement();
ResultSet rs = stm.executeQuery(``SELECT .. FROM ...`` );
boolean mehr = rs.next();
while (mehr){
```

```
    String beispiel = rs.getString(2);
    System.out.println(beispiel);
    mehr = rs.next();
}
```

Das Result Set wird gleich beim Aufruf der SQL-Anweisung mitdefiniert. Innerhalb der `while`-Schleife werden die Tupel sequentiell bearbeitet; in diesem Beispiel wird jeweils das zweite Attribut der Tupel des Result Sets ausgelesen und ausgegeben.

Verarbeitung von Warnungen und Ausnahmen

Da der Parser keine Überprüfung der SQL-Syntax vornimmt, kann es zu Laufzeitfehlern kommen, die eine SQL-Ausnahme auslösen.

Datenbankoperationen müssen daher stets innerhalb eines Try-Catch-Blockes ausgeführt werden. In einer Try-Anweisung können mehrere Anweisungen gebündelt werden. Löst eine der Anweisungen zur Laufzeit eine Ausnahme aus, wird die Ausführung der restlichen Anweisungen unterbrochen und die Catch-Anweisung gesucht, die für den Typ der ausgelösten Ausnahme eine Fehlerbehandlungsroutine beinhaltet.

Es empfiehlt sich, vom Treiber-Manager eine Log-Datei führen zu lassen, in der die Fehlermeldungen genau protokolliert werden, denn jede SQL-Ausnahme beinhaltet folgende Informationen:

- einen String, der die Beschreibung des aufgetretenen Fehlers beinhaltet,
- einen Status,
- einen herstellerspezifischen Fehlercode und
- eine Verknüpfung zu einer weiteren Ausnahme, die mit der aktuell ausgelösten Ausnahme zusammenhängt.

Die selten auftretenden Warnungen können wie die Ausnahmen protokolliert und verarbeitet werden, sie führen allerdings nicht zu einem Programmabbruch.

Das Transaktionskonzept

Jede Datenbank-Anweisung wird wie eine eigene Transaktion behandelt, es ist kein expliziter Aufruf der Methode `commit()` nötig, da per Default die AutoCommit-Funktion eingeschaltet ist. Sollen mehrere Datenbank-Anweisungen eine Transaktion bilden, müssen vorher die AutoCommit-Funktion ausgeschaltet und alle Anweisungen in einem Try-Catch-Block mit anschließendem `commit()` zusammengefaßt werden.

JDBC unterstützt zur Zeit kein Zwei-Phasen-Commit und eignet sich daher für verteilte Datenbanken kaum.

JDBC 2.0

Zur Zeit steht die Spezifikation von JDBC 2.0 öffentlich zur Diskussion, die Ergebnisse sollen diesen Sommer eingefroren werden.

Nachteilig bei der Implementierung der Datenbankzugriffe mittels JDBC erwies sich die bestehende Form der Auswertung der Anfrageergebnisse. So können die Datenfelder des Resul Set zur Zeit nur einmalig und sequentiell (vorwärts) ausgewertet werden. Daher mußten für einzelne Datenfelder, die mehrfach referenziert werden sollten, Variablen definiert werden. Gegebenenfalls mußten ganze Objekte zwischengespeichert werden. Mit JDBC 2.0 werden dann die Anfrageergebnisse flexibler auswertbar sein. Dazu wird das Konzept der „scrollable cursors“ verwendet, das bereits aus ODBC bekannt ist.

Zu erwarten ist weiterhin eine Unterstützung von komplexen SQL-3 Typen wie Binary Large Objects (kurz BLOB) und strukturierten Datentypen.

Ein weiteres Feature wird der CORBA-kompatible Object Request Broker (kurz ORB) sein, der die Anbindung von Java-Clients an ORB-fähige Server erlauben wird. Damit können auch Java-Clients in verteilte Anwendungen einbezogen werden. So wird ein transparenter Zugriff auf Java-Objekte im Netz möglich sein.

Generell sind in verschiedenen Teilbereichen von JDBC Performance-Steigerungen zu erwarten.

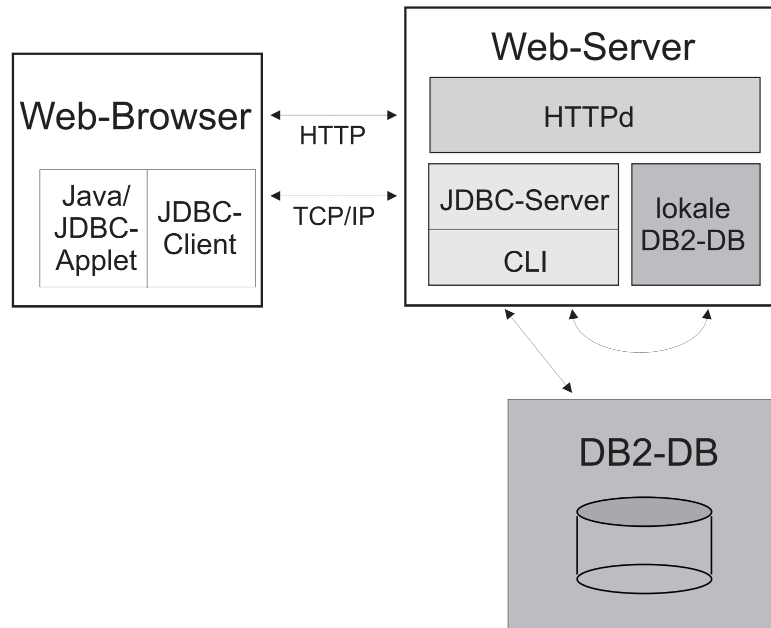

Abbildung 5.3: JDBC-Applet für DB2

5.4 Java-Binding für DB2

Java kann auf Daten einer DB2-Datenbank über die JDBC-API des JDK 1.1 zugreifen. Die DB2-Unterstützung für Java besteht aus zwei unabhängigen Komponenten:

- Java-Applikationen und -Applets auf der Client-Seite greifen über JDBC auf DB2 zu.
- UDFs und stored procedures können auf Server-Seite in Java geschrieben werden.

5.4.1 JDBC-Applikationen und -Applets in DB2

Die Abbildung 5.3 soll die Arbeitsweise eines JDBC-Treibers eines Applets verdeutlichen.

Der Treiber besteht aus einem JDBC-Client und -Server. Wenn eine Verbindung zur Datenbank benötigt wird, lädt der Web-Browser bzw. das Java-Applet den JDBC-Client, der über TCP/IP den JDBC-Server anspricht. Der

JDBC-Server veranlaßt mit den entsprechenden CLI (ODBC)-Befehlen die Datenbank, die geforderte Aufgabe abzuarbeiten. Die entstehenden Ergebnisse nimmt der JDBC-Server auf und sendet sie über die bestehende TCP/IP-Verbindung an den JDBC-Client zurück. DB2 muß dazu auf dem Web-Server installiert sein.

Eine Applikation hingegen benötigt eine Installation der CAE-Komponente (Client Application Enabler) auf der Client-Seite. Diese CAE-Komponente steuert die Kommunikation zwischen der Applikation und der Datenbank. Vorher müssen aber die JDBC-Anweisungen in DB2 CLI-Anweisungen durch native Methoden von Java übersetzt werden.

5.4.2 UDFs und stored procedures

Wenn die in Java implementierten UDFs und stored procedures registriert sind, können sie von jedem Programm (egal in welcher Sprache implementiert) aufgerufen werden. DB2 lädt den Java-Interpreter beim ersten Aufruf einer Java-UDF oder stored procedure.

DB2 verfügt allerdings nicht über einen eigenen Java-Interpreter, vorher muß erst das JDK nebst Interpreter installiert und konfiguriert werden.

5.5 Fazit

Um den Java-Klassen in DB2 Persistenz zu verleihen, wurden diese in das Relationenmodell abgebildet. Das Ergebnis ist ein Schema, das aus sechs Relationen besteht und in der vierten Normalform vorliegt. Es traten dabei Schwierigkeiten mit dem Java-Datentyp `Array` auf, die mit zusätzlichen Attributen und einer weiteren Relation gelöst wurden. Dadurch wird die Anfragestellung komplizierter. Die Bildung des Schlüssels der Relation `Person` muß innerhalb der Java-Methode `saveAngebot()` erfolgen.

Bereits bei der Abbildung der Java-Klassen in ein relationales Modell offenbaren sich einige Schwächen des Relationenmodells gegenüber dem objektorienterten Modell, doch dieses Thema wird zu einem späteren Zeitpunkt (Kapitel 8) ausführlicher betrachtet werden.

Nachdem die Relationen in DB2 angelegt wurden, kann der Zugriff des

Prototypen über JDBC erfolgen. JDBC stellt eine einfach zu benutzende Schnittstelle für Java-Zugriffe auf verschiedene Datenbanken dar. Durch die Anlehnung an ODBC fällt es vielen Benutzern, die bereits Erfahrungen mit ODBC hatten, leicht, auch mit JDBC umzugehen. Die meisten Datenbank-Hersteller haben mittlerweile JDBC für ihre Produkte umgesetzt.

Kapitel 6

Die Schnittstelle für O_2

In diesem Kapitel wird das Java-Binding (kurz JB) von O_2 vorgestellt. Es wird gezeigt, daß der Java-Entwickler dank des komfortablen JB von O_2 auf eine Abbildung seiner Java-Klassen in das Datenmodell von O_2 verzichten kann. Daher soll auf eine kurze Beschreibung des Datenmodells von O_2 verzichtet werden.

6.1 Das Java-Binding von O_2

Das in den folgenden Abschnitten beschriebene Java-Binding entspricht der Version 5.0 des Java-Bindings von O_2-Technology.

Mit dem Java-Binding von O_2 ist es möglich, Java-Klassen Persistenz in verschiedenen Datenbanken (neben O_2 auch Sybase und Oracle sowie alle Datenbanken mit einem JDBC-Treiber) zu verleihen und aus Java-Anwendungen auf die Datenbank zuzugreifen.

Neben dem JB-API (das notwendige Klassen zur Datenbankverbindung und Transaktionsverwaltung bereitstellt) gehört auch eine Werkzeugsammlung zum Produkt. Diese Werkzeuge ermöglichen es dem Entwickler, ohne Kenntnisse über die verwendete Datenbank aus seinen Java-Klassen ein Datenbankschema und die nötigen Datenbankoperationen zu generieren.

Diese Möglichkeit kommt gerade Programmierern entgegen, die sich nicht groß um Datenbank-Angelegenheiten kümmern, sondern nur die sogenannte JB-Applikation entwickeln wollen.

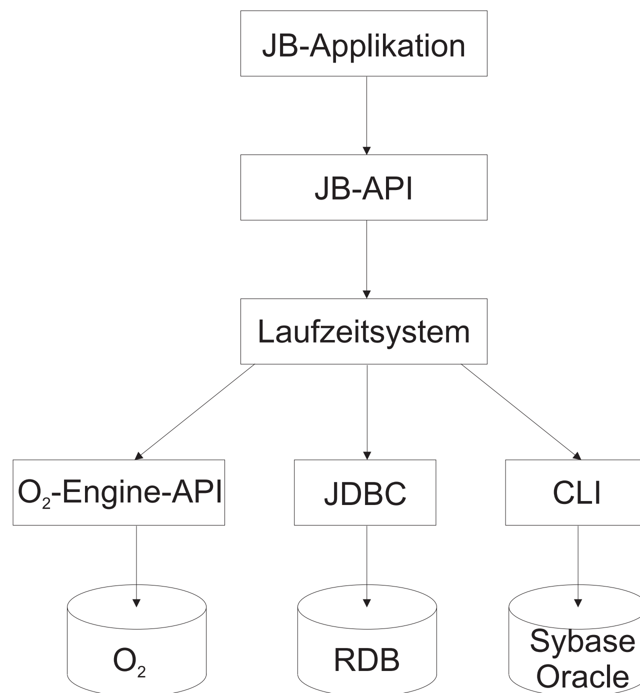

Abbildung 6.1: Allgemeine JB-Architektur von O_2

Die Abbildung 6.1 soll die allgemeine JB-Architektur veranschaulichen.

Wie dieser Abbildung zu entnehmen ist, existieren drei Implementationen des JB-API:

- Die nativen Methoden von O_2 werden genutzt, so kann direkt über die O_2-Engine-API auf eine O_2-Datenbank zugegriffen werden.
- Mittels JDBC kann auf Datenbanken zugegriffen werden, die über einem JDBC-kompatiblen Treiber verfügen.
- Über CLI kann direkt auf Oracle oder Sybase zugegriffen werden.

Die Entwicklung einer JB-Applikation besteht allgemein aus folgenden drei Schritten:

1. Vorbereitung des Datenbank-Servers
2. Import der Java-Klassen in die Datenbank
3. Implementierung der Datenbankzugriffe unter Nutzung des JB-API

Beim Import der Java-Klassen wird deren Struktur als entsprechendes Datenbank-Schema generiert. Weiterhin werden die Java-Klassen um die nötigen Datenbankmethoden erweitert.

6.1.1 Allgemeine Konzepte des JB von O_2

Vorbereitung des Datenbank-Servers

Da verschiedene DBMS verwendet werden können, und diese eine unterschiedliche Funktionalität besitzen, muß dem API und den Werkzeugen mit Hilfe eines URL mitgeteilt werden, welche Datenbank verwendet wird. Der Aufbau des URL ist bereits im Abschnitt 5.3.3 beschrieben worden.

Mit den entsprechenden Werkzeugen kann nun eine Datenbank angelegt (`o2jb_create_schema`, `o2jb_create_base`) und initialisiert (`o2jb_init_base`) werden.

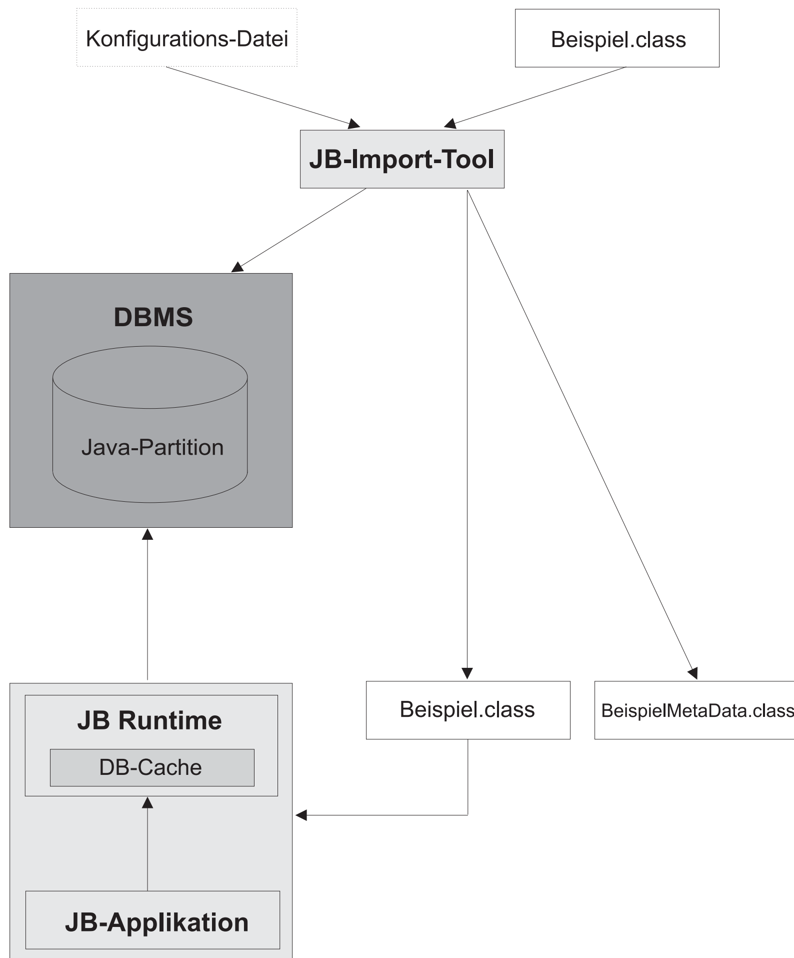

Abbildung 6.2: Importieren einer Java-Klasse in O_2

Das Importieren von Klassen

Mit dem Importierwerkzeug (`o2jb_import`) werden Klassen in die Datenbank importiert, denn nur die Instanzen von importierten Klassen können persistent gemacht werden.

Die Abbildung 6.2 veranschaulicht die verschiedenen Schritte des Imports einer Klasse.

Die Klasse `Beispiel` soll importiert werden. Das Werkzeug legt in der Datenbank die notwendigen Strukturen entsprechend der Klassendefinition der zu importierenden Klasse an und generiert die Implementierung der

Methoden, die für die Zugriffe auf die Datenbank nötig sind, direkt in die `Beispiel.class`-Datei. Weiterhin wird eine `BeispielMetaData.class`-Datei generiert, die die Struktur der importierten Klasse und deren Nutzung zur Laufzeit beschreibt.

Mit einer Konfigurations-Datei kann der Anwender auf die Arbeitsweise des Importwerkzeuges Einfluß nehmen.

- Bestimmte Klassenvariablen können als transient markiert werden. Das bedeutet, daß diese Variablen beim Speichern eines Objektes in die Datenbank nicht abgespeichert werden. Das ist sinnvoll bei Variablen, die z.B. nur Zwischenwerte von Berechnungen enthalten.
- Namenskonflikte können durch Umbenennungen aufgehoben werden.
- Die Zeichenkettenlänge der Java-Strings läßt sich auf eine begrenzte Länge reduzieren, die dann in die Datenbank abgelegt wird.

Nach dem Importieren einer Klassen können Instanzen dieser Klasse persistent in der Datenbank gehalten werden.

Implementierung der JB-Applikation

Das JB-API stellt unter anderem die Klassen `Database` und `Transaction` bereit, die in einer JB-Applikation von großer Bedeutung sind.

Mit einer Instanz der Klasse `Database` kann die JB-Applikation die Verbindung zum Datenbank-Server herstellen und eine neue Verbindung mit einem Datenbankschema eröffnen. Dazu werden die Methoden `connect()` und `open(Datenbank)` benutzt. Die Verbindung zur Datenbank kann mit der Methode `close()`, zum Server mit `disconnect()` geschlossen werden.

Innerhalb einer Transaktion können Zugriffe auf und Änderungen an persistenten Objekten durchgeführt werden. Am Ende der Transaktion (`transaction.commit()`) werden die Änderungen durchgeschrieben, bei Fehlern werden sämtliche bereits innerhalb der Transaktion getätigten Änderungen zurückgenommen. Die Methode `validate()` beendet wie `commit()` die Transaktion, allerdings wird der JB-Cache, in dem zunächst die persistenten Objekte gehalten werden, nicht gelöscht. Die Methode `abort()` bricht

die Transaktion ab und macht alle Änderungen an persistenten Objekten rückgängig.

Somit haben JB-Programme im wesentlichen folgenden Aufbau.

```
database = new Database(...);       //DB-Instanz anlegen
database.connect();                 //Verbindung zum DB-Server
   database.open(base1);
     transaction = new Transaction();
     transaction.begin();
       //Erstellen, Verändern und Löschen von Objekten
     transaction.commit();          //Durchschreiben der Objekte
   database.close();                //Schließen der aktuellen DB
database.disconnect();
```

Handhabung der persistenten Objekte

Die importierten Klassen wurden durch das Importier-Werkzeug um alle benötigten Methoden für Datenbankzugriffe erweitert. Somit kann der Zugriff auf die Datenbank weiterhin in Java geschehen, es müssen nicht wie bei JDBC SQL-Anweisungen bzw. die Anfragesprache des jeweiligen DBMS benutzt werden.

Alle Objekte innerhalb einer JB-Applikation haben einen der beiden möglichen Zustände:

- transient
  Ein transientes Objekt stammt nicht aus der Datenbank und existiert deshalb nur im Heap des Programmes und ist nicht im JB-Cache. Es besitzt kein korrespondierendes Objekt in der Datenbank.

- persistent
  Ein persistentes Objekt kommt aus der Datenbank oder wurde innerhalb einer Transaktion als persistent markiert. Es befindet sich im JB-Cache und besitzt ein korrespondierendes Objekt in der Datenbank.

Ein transientes Java-Objekt, dessen Klasse importiert wurde, kann mit Aufruf der Methode `persist(Objekt)` der Klasse `Database` persistent werden.

Über den Klassenextent, der alle Objekte einer Klasse enthält, die in die Datenbank geschrieben wurden, können Objekte aus der Datenbank geladen werden. Der Extent kann mit einem Prädikat gefiltert werden, die Syntax ähnelt der `SELECT FROM WHERE`-Anfrage von OQL.

```
Extent personExtent;
personExtent = Extent.all(''Person'').where(''this.name
= 'John' '');
```

Änderungen an den persistenten Objekten innerhalb der Transaktion werden beim Transaktionsende in die Datenbank durchgeschrieben.

Die Methode `delete(Objekt)` der Klasse `Database` löscht ein persistentes Objekt aus der Datenbank.

Arrays müssen explizit persistent gemacht werden.

Weitere Ausführungen können der von O_2 -Technology mitgelieferten Dokumentation [Man01] entnommen werden.

6.2 Fazit

Das JB von O_2 ist sehr komfortabel und entlastet den Java-Programmierer von einigen Arbeitsschritten, die bei der Implementierung der Java-Schnittstelle für DB2 nötig waren; z.B. entfällt die Abbildung der Java-Klassen in das Datenmodell der benutzten Datenbank und die Simulierung der Arrays.

Anfragen werden über Prädikate des Klassenextentes in Java gestellt.

Mit der Konfigurations-Datei können die Attribute der Java-Klassen, die nur Zwischenwerte der Berechnungen enthalten, als transient markiert werden (siehe Anhang C). Sie werden beim Importieren der Klasse nicht in das Datenbankschema übernommen.

Kapitel 7

Die Workflow-Komponente

Die eingehenden Angebote sollen nicht nur in einer Datenbank gehalten werden, sondern auch automatisch an Mitarbeiter der Gothaer Versicherungen a.G. weitergeleitet werden. Welche Mechanismen sich die AVSG vorstellt, wird im Abschnitt 7.1 beschrieben.

Danach werden die Realisierungsmöglichkeiten für die beiden Datenbanksysteme DB2 und O_2 im Abschnitt 7.2 diskutiert.

7.1 Beschreibung der erwarteten Leistung

Die automatische Weiterverarbeitung der eingegangenen Angebote soll sich wie folgt gestalten:
In periodischen Zeitabständen oder auf Anforderung sollen die Angebote

1. nach vorgebbaren Merkmalen sortiert werden.
   Sinnvolle Merkmale wären z.B. das Geschlecht oder die Postleitzahl des Anwenders.

2. an die zuständigen Sachbearbeiter verschickt werden.
   Die zuständigen Sachbearbeiter können über die Postleitzahl bestimmt werden. Für die Versendung der Daten bieten sich folgende Varianten an:

   - Per E-Mail wird die (eindeutige) Angebotsnummer übermittelt. Das komplette Angebot kann sich der zuständige Sachbearbeiter

dann aus der Datenbank holen.

- Das komplette Angebot wird im ASCII-Format per E-Mail versendet.
- Das komplette Angebot wird in einem für DB2 weiterverarbeitbaren Format per E-Mail versendet.

Der Begriff Workflow ist für dieses Beispiel sicherlich etwas hochgegriffen, auch wenn diese Komponente prozeßorientiert angesehen werden kann, wird unter Workflow in [JBS97] doch wesentlich mehr verstanden: Die formale Beschreibung und Modellierung wesentlich komplexerer Geschäftsprozesse. Eine Klassifikation von Workflow-Management-Systemen ist z.B. in [Rui97] gegeben.

7.2 Realisierungsmöglichkeiten

7.2.1 Realisierung durch aktive Komponenten

Die Workflow-Komponente läßt sich eigentlich mit den Konzepten von aktiven DBMS realisieren. In [DG96] wird ein aktives DBMS wie folgt definiert:

> Ein Datenbanksystem heißt *aktiv*, wenn es zusätzlich zu den üblichen Datenbanksystem-Fähigkeiten in der Lage ist, definierbare Situationen in der Datenbank (und wenn möglich darüber hinaus) zu erkennen und als Folge davon bestimmte (ebenfalls definierbare) Reaktionen auszulösen.

Eine Situation in diesem Sinne wird allgemein durch ein Paar, bestehend aus einem Ereignis und einer Bedingung, gekennzeichnet.

Ein *Ereignis* ist ein punktuelles Geschehnis, dem ein Zeitpunkt zugeordnet werden kann. Somit besteht ein Ereignis aus einer Ereignisbeschreibung und dem Ereigniszeitpunkt. Man unterscheidet primitive und zusammengesetzte Ereignisse.

Die *Bedingung* ist ein Prädikat über der Datenbank und kann mit der jeweiligen Anfragesprache formuliert werden.

Die Reaktion wird üblicherweise als *Aktion* bezeichnet und ist allgemein ein beliebiges ausführbares Programmstück, das Operationen, die nicht nur auf die Datenbank wirken müssen, enthalten kann.

Diese drei Bestandteile werden mit ECA-Regeln (Event - Condition - Action) beschrieben. In manchen Systemen werden sie auch Trigger genannt. Somit könnte eine ECA-Regel wie folgt aussehen:

```
DEFINE RULE variante_1
ON EVERY DAY 0:00
DO {
SQL-Anweisung;
EXEC Send_email;
}
```

Diese ECA-Regel soll verdeutlichen, wie die in Abschnitt 7.1 beschriebene erste Variante einer Realisierungsmöglichkeit (per E-Mail wird die Angebotsnummer an die zuständigen Sachbearbeiter verschickt) definiert werden kann. In der ersten Zeile wird die ECA-Regel mit dem Namen `variante_1` deklariert. Sie wird jeden Tag um 0:00 Uhr ausgeführt, eine Bedingung entfällt. Die Aktion gliedert sich in zwei Schritte: Die SQL-Anweisung bestimmt alle am vergangenen Tag erstellten Angebote, projeziert sie auf die Postleitzahl und Angebotsnummer und sortiert sie nach der Postleitzahl. Danach wird die Prozedur `Send_email` gestartet, die aus dem Anfrageergebnis des 1. Schrittes die erforderlichen E-Mails generiert.

Mit diesem Beispiel zeigt sich schon, daß alle in Abschnitt 7.1 vorgestellten Varianten mit ECA-Regeln realisierbar sind:

- Das **Ereignis** tritt ein, wenn
  - A. periodisch ein bestimmter Zeitpunkt erreicht wird oder
  - B. eine explizite Anforderung gestellt wird.
- Die **Bedingung** kann für alle Varianten entfallen.
- Die **Aktion** gliedert sich in zwei Schritte:

1. Mit der jeweiligen Anfragesprache werden die Angebote entsprechend aufbereitet. Mit einer Selektion werden die neuen Angebote bestimmt, eine Projektion verringert die Attribute auf diejenigen, die dem Sachbearbeiter gesendet werden sollen. Danach kann eine Sortierung nach der Postleitzahl erfolgen.

2. Eine in einer höheren Programmiersprache implementierte Prozedur generiert die erforderlichen E-Mails. Das Anfrageergebnis wird sequentiell abgearbeitet. Als erstes wird über die Postleitzahl der zuständige Sachbearbeiter bestimmt. Dann werden die Daten in das zum Senden benutzte Format gebracht. Anschließend wird dem Sachbearbeiter eine E-Mail gesendet, die die Angebote enthält.

In den folgenden Abschnitten wird überprüft, ob auch DB2 und O_2 über solche Konzepte verfügen und wie die verschiedenen Varianten umgesetzt werden können.

Möglichkeiten in DB2

Für DB2 können Trigger definiert werden, die nach oder vor der Ausführung einer SQL-Anweisung `CREATE`, `INSERT` oder `UPDATE` auf einer bestimmten Tabelle aktiviert werden. Die Definition geschieht mit der SQL-Anweisung `CREATE TRIGGER`. Ein aktiver Trigger sorgt für die Ausführung von möglicherweise mehreren Operationen. Diese können SQL-Anweisungen oder UDFs sein. Eine Bedingung kann z.B. in Form einer `WHEN`-Klausel optional angegeben werden.

Am Ende des Abschnittes 7.2.1 wurde zusammengefaßt, welche Voraussetzungen die aktiven Komponenten des jeweiligen DBMS erfüllen müssen, um die Workflow-Komponente zu realisieren. Die Workflow-Komponente kann in DB2 also wie folgt realisiert werden:

- Ereignis
  DB2-Trigger können periodisch auftretende Ereignisse nicht definieren. Ereignis A muß also „von außen“, z.B. durch ein Shell-Skript, das eine SQL-Anweisung periodisch ausführt, umgesetzt werden. Ereignis B ist

mit einem Trigger definierbar. Für beide Ereignisse muß eine neue Tabelle angelegt werden, auf die die aktionsauslösende SQL-Anweisung zugreift.

- Bedingung
  Die Bedingung kann entfallen.

- Aktion
  Der erste Schritt der Aktion (das Aufbereiten der Daten) kann mit SQL-Anweisungen vollzogen werden. Der zweite Schritt (das Generieren der E-Mails) kann nicht realisiert werden, da keine externen Programme aufgerufen werden können.

Die Workflow-Komponente kann also nicht mit einem Trigger in DB2 umgesetzt werden. Daher ist eine Anwendung zu implementieren, die entweder periodisch durch ein Shell-Skript oder auf Wunsch direkt gestartet wird und beide Schritte der Aktion ausführt.

Möglichkeiten in O_2

Die aktive Komponente von O_2 weicht ihrem Funktionsprinzip stark von den üblichen ECA-Regeln oder den Triggern, wie sie z.B. in DB2 existieren, ab. Daher wird sie in einem eigenen Abschnitt genauer vorgestellt. Im sich daran anschließendem Abschnitt werden die Realisierungsmöglichkeiten der Workflow-Komponente in O_2 mit aktiven Komponenten untersucht.

Der Notification Service. In O_2 steht als aktive Komponente der sogenannte *Notification Service* zur Verfügung, der es einem Client erlaubt, andere Clients, die mit dem gleichen Server verbunden sind, zu informieren, daß ein Ereignis eingetreten ist. Alle Clients können beim Notification Service registrieren lassen, über welche eintretenden Ereignisse sie informiert werden möchten.

Abbildung 7.1 veranschaulicht das Wirkungsprinzip des Notification Service.

Client 1 löst das Ereignis E1 aus. Der O_2-Server erhält die Nachricht, daß Ereignis E1 eingetreten ist und informiert darüber alle Clients, die für

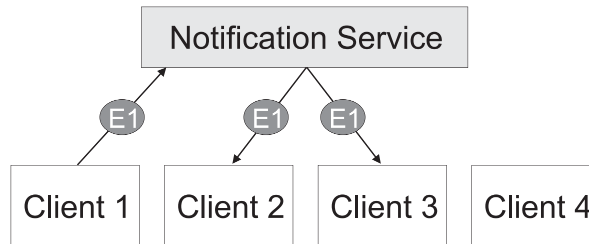

Abbildung 7.1: Der Notification Service von O_2

das Ereignis E1 registriert sind. Client 4 hat sich für das Ereignis E1 nicht registrieren lassen und erhält folglich auch keine Nachricht.

Die Nachricht kann neben der genauen Bezeichnung des eingetretenen Ereignisses auch eine Referenz eines persistenten Objektes enthalten. Eintreffende Nachrichten werden beim Client in einer extra dafür vorgesehenen Speicherstruktur, die nach dem FIFO-Prinzip (First In First Out) arbeitet, gesammelt.

Die Ereignisse werden in folgende Kategorien eingeteilt:

- Objekt-Ereignis
  Ein Objekt-Ereignis tritt ein, wenn ein Objekt, das als *notifiable object* registriert wurde, gelöscht oder geändert wurde. Jeder Client kann beim Notification Service ein Objekt als notifiable object registrieren lassen.

- Anwender-Ereignis
  Jeder Client kann ein beliebiges Anwender-Ereignis definieren und muß es dann beim Notification Service registrieren. Soll ein Anwender-Ereignis eintreten, informiert der Client den Notification Service über den Eintritt des Anwender-Ereignisses mit einer speziell dafür vorgesehenen Nachricht, denn das Anwender-Ereignis wird nicht wie die anderen Ereignisse vom System automatisch erkannt.

- Client-Ereignis
  Ein Client-Ereignis kann ein Connect- oder Disconnect-Ereignis sein, je nachdem ein Client die Verbindung zum Server auf- oder abbaut.

Die Clients können, wie in Abbildung 7.1 deutlich wird, Empfänger oder Sender (oder beides) sein. Ein Empfänger kann bei der Registrierung einen

Filter setzen, ob er über das Eintreten eines bestimmten Objekt-Ereignisses (bzw. Client-Ereignisses) oder aller Objekt-Ereignisse (bzw. Client-Ereignisse) informiert werden möchte. [Man02]

Umsetzung der Workflow-Komponente mit dem Notification Service. Zeitereignisse müssen als Anwender-Ereignisse simuliert werden. Um die geforderten Schritte der Aktion zu realisieren, muß eine Anwendung geschrieben werden, die ständig die Nachrichtenschlange abfragt, ob das Ereignis eingetreten ist. Danach müssen die Angebote geholt und aufgearbeitet werden, bevor sie per E-Mail versendet werden können. Somit übernimmt diese Anwendung fast sämtliche Schritte der Workflow-Komponente. Sie kann deshalb auch gleich durch ein Shell-Skript gestartet werden.

Deshalb ist der Notification Service für die Realisierung der Workflow-Komponente unbrauchbar.

7.2.2 Realisierung durch eine Java-Anwendung

In den vorangegangenen Abschnitten wurde deutlich, daß mit den aktiven Komponenten der beiden DBMS eine Umsetzung der Workflow-Komponente unmöglich war. Die aktiven Komponenten sind nicht in der Lage, ein externes Programm zu starten, das z.B. die Generation der E-Mails übernimmt. Beide Systeme sind auch nicht in der Lage, Zeitereignisse zu definieren und zu erkennen. Dieses Problem hätte z.B. mit einen Shell-Skript, das periodisch auf ein bestimmtes Objekt bzw. auf eine bestimmte Relation zugreift, gelöst werden können.

Dieses Shell-Skript könnte aber eine Anwendung starten, die sich die notwendigen Daten aus der Datenbank holt und die E-Mails generiert. Diese Anwendung kann dann auf jedem beliebigen Host laufen, der eine Verbindung zum DBMS herstellen kann.

Weil DB2 bei der Gothaer Versicherungen a.G. zur praktischen Anwendung kommt und O_2 in dieser Arbeit nur als Vergleich zum Erkennen von Alternativen, Vorteilen und Mängeln dient, wird die weitere Betrachtung der Workflow-Komponente speziell auf DB2 orientiert sein. Die Implementierung einer Anwendung für O_2 ist jedoch in fast allen Schritten identisch. Allerdings

verfügt O_2 über keine Importmöglichkeit.

Theoretisch kann diese Anwendung auf jedem Betriebssystem mit verschiedenen Programmiersprachen implementiert werden. Naheliegend ist aber eine Implementierung in Java aus folgenden Gründen:

- Plattformunabhängigkeit.
  Die Java-Applikation kann auf allen Betriebsystemen ausgeführt werden. Damit könnte sie auf jedem beliebigen Host innerhalb des Intra- oder gar Internets laufen.

- Nutzung der bereits vorhandenen Java-Schnittstelle.
  Die Schnittstelle wurde zu Testzwecken (siehe Abschnitt 4.2) um eine Methode erweitert, die Angebote aus der Datenbank holt. Darauf aufbauend kann eine Methode implementiert werden, die alle neuen Angebote (eventuell schon entsprechend aufbereitet) der Java-Applikation zur Verfügung stellt. Für die Implementierung für DB2 ist z.B. nur die gesendete SQL-Anfrage der vorhandenen Methode entsprechend zu verändern.

- Frei verfügbare E-Mail-Clients für Java.
  Im Internet sind bereits mehrere, sehr komfortable E-Mail-Clients frei verfügbar. Im folgenden wird noch eine Java-Klasse vorgestellt, die E-Mails auf sehr einfache Weise versenden kann. Somit müssen keine Fremd-Werkzeuge in die Applikation eingebunden werden.

- Nutzung nativer Methoden anderer Programmiersprachen.
  In Java kann auch Code anderer Programmiersprachen eingebunden werden. In dem C-API von DB2 ist z.B. eine Funktion vorhanden, mit der man Daten aus DB2 in verschiedene Dateiformate exportieren kann. Diese Funktion kann in C geschrieben und in die Java-Anwendung eingebunden werden. Wie das möglich ist, ist z.B. in [New97] und [Mor97] beschrieben.

- Zukunftssicherheit.
  In [FEY98] ist dargelegt, warum die Anzahl der Implementierungen

mit Java in Zukunft zunehmen wird. Auch Performance-Einbußen gegenüber Sprachen wie C++ können dank JIT-Compiler (Just In Time) und schnellerer VMs (Virtual Machine) vermieden werden.

Ein Implementierung in C ist ebenfalls eine bequeme Alternative, allerdings ist dann auf die Plattformunabhängigkeit zu verzichten.

Die Implementierung der Anwendung in Java gliedert sich in folgende Etappen:

- Die Java-Schnittstelle wird um eine neue Methode ergänzt. Diese holt alle neuen Angebote sortiert aus der Datenbank. Die Angebote können dann in jedes beliebige Format gebracht werden.
- In Abhängigkeit der Organisation der Informationen über die Sachbearbeiter (Datei oder Datenbank) wird die Ermittlung der zuständigen Sachbearbeiter implementiert.
- Für den Export der Daten eröffnen sich zwei Möglichkeiten:
  - Die Export-Funktion von DB2 wird benutzt. Die Funktion wird in C geschrieben und als C-Code in Java eingebunden. Welche Dateiformate möglich sind, wird in Abschnitt 7.2.3 beschrieben.
  - Die Daten werden über JDBC aus DB2 geholt. Die Java-Objekte können dann in jede beliebige Form gebracht werden, wenn die entsprechende Methode implementiert wird.
- Mit Hilfe eines Java-E-Mail-Clients kann die Erstellung und Versendung der E-Mails an die Sachbearbeiter realisiert werden.

Eine einfach zu verwendende Klasse zum Versenden von E-Mails ist `jSendEMail` von Johnathan Mark Smith. Diese Klasse ist unter

`http://www.statenislandonline.com/java/jSendEMail/`
`index.html`

inklusive Dokumentation verfügbar. Im Anhang E befindet sich eine kurze Übersicht zu den Methoden und ein kleines Beispiel zur Anwendung dieser Klasse.

Unter der Adresse

`http://home.pi.net/~hverbeek/Java.html`

zeigt der Autor Harm Verbeek in zwei Java-Dateien, wie mit eigenen Applets E-Mails [WWW04] und binäre Dateien per Attachement in einer E-Mail [WWW05] verschickt werden können. [WWW05] ist auszugsweise im Anhang E enthalten.

7.2.3 Export der Daten aus DB2

Innerhalb des C-API von DB2 wird eine Funktion `sqluexpr()` bereitgestellt, mit der Daten aus DB2 in folgende Dateiformate exportiert werden können:

- DEL
  Beschränktes ASCII-Format, zum Austausch mit dBase, BASIC, Programmen der IBM Personal Decision Serie und vielen Datenbank- und Dateimanagern.

- WSF
  Worksheet-Format zum Austausch mit Lotus Symphony und 1-2-3 Programmen.

- IXF
  PC-Version des Integrated Exchange Fromat. Häufig verbreitetes Format zum Exportieren von Daten aus einer Tabelle, die später wieder in die gleiche oder in andere Tabellen anderer Systeme geladen werden können.

Sollen die Angebote im ASCII-Format versendet werden, ist das DEL-Format zu verwenden.

Werden versendete Daten wieder in eine DB2-Datenbank eingelesen werden, bietet sich das informationsreichere IXF-Format an. In diesem Format können sogar die Index-Definitionen einer Relation erhalten werden, wenn die SQL-Operation, die die Daten selektiert, keine Sicht darstellt oder einen Verbund enthält. Hatte die SQL-Anfrage die Form von `SELECT * FROM tablename`, werden auch die Definitionen der Default- und `NOT NULL`-Werte gespeichert.

Allerdings werden zum Laden der kompletten Angebote Verbunde und `WHERE`-Klauseln benötigt, so daß diese zusätzlichen Informationen auch im IXF-Format verloren gehen.

DB2 kann neben den drei vorgestellten Formaten auch unbeschränktes ASCII importieren. Damit ergeben sich vier Dateiformate, die von DB2 weiterverarbeitbar sind.

7.3 Fazit

Theoretisch ist die Workflow-Komponente mit aktiven Komponenten eines DBMS realisierbar. Leider bieten die aktiven Komponenten von DB2 und O_2 keine Möglichkeit, externe Programme zu starten, die das Versenden der E-Mails übernehmen. Deshalb ist eine Anwendung zu implementieren, die sämtliche Schritte der Workflow-Komponente übernimmt. Als Implementierungssprache bietet sich (neben C) Java an.

Zum Laden der neuen Angebote ist nur eine sehr geringfügige Änderung an der bereits bestehenden Java-Schnittstelle nötig.

Der Export der Daten kann auf zwei unterschiedlichen Wegen erfolgen.

1. Die Instanzen der Java-Klasse `Angebot` können direkt im ASCII-Format gespeichert werden.

2. Die Export-Funktion der C-API von DB2 kann Daten in drei Dateiformate (DEL, IFX, WSF) bringen. Zusätzliche Datenbank-Informationen, die im IFX-Format gespeichert werden könnten, gehen aufgrund der SQL-Anweisung verloren, so daß hier nicht mehr Informationen als im beschränkten ASCII-Format DEL enthalten sind.

Das Senden der generierten E-Mails kann mit frei verfügbaren E-Mail-Clients erfolgen. Dabei können die Angebote in den verschiedenen Formaten als Attachement der E-Mail beigefügt werden

Kapitel 8

Vergleich der beiden Datenbanken

In diesem Kapitel sollen die beiden verwendeten DBMS hinsichtlich ihrer Eignung für den Prototypen verglichen werden. Dabei wird aber teilweise auch auf Stärken bzw. Schwächen eingegangen, die für den Prototypen noch nicht ausschlaggebend sind, bei komplexeren Anwendungen aber gravierend werden können. Beim Vergleich der beiden DBMS liegt das Augenmerk auf folgenden Schwerpunkten:

- Datenmodellierung
- Datenbankentwurf
- Anfragesprache
- Java-Binding
- Export der Daten

Die Möglichkeiten der aktiven Komponenten beider DBMS sind im Kapitel 7 bereits beschrieben und diskutiert worden. Auf sie wird innerhalb dieses Kapitels nur zusammenfassend im Abschnitt 8.7 eingegangen.

In einem kurzen Abschnitt werden auf Probleme bei der praktischen Nutzung beider Systeme hingewiesen.

Es soll nicht Aufgabe dieser Arbeit sein, grundlegende Begriffe und Konzepte aus der Welt der Datenbanken zu definieren und zu erläutern (das

würde den Rahmen dieser Arbeit sprengen). Deshalb soll an dieser Stelle auf zahlreiche Literatur zu Grundlagen wie etwa [Ull88], [HS95] und [Heu97] verwiesen werden. In [Heu97] ist ein ausführlicher Vergleich des relationalen und objektorientierten Modells enthalten. Dieser Vergleich bildet damit eine wesentliche Bezugsquelle der Ausführungen zu den Schwerpunkten Datenmodellierung, Datenbankentwurf und Anfragesprache in den folgenden Abschnitten.

Vor- und Nachteile von DB2 und O_2 beruhen fast immer auf den generellen Konzepten des Relationenmodells bzw. eines objektorientierten Modells, so daß in den folgenden Abschnitten teilweise mehr eine Abwägung hinsichtlich objektorientiertes DBMS gegen relationales DBMS als O_2 gegen DB2 erfolgt.

8.1 Datenmodellierung

8.1.1 DB2

Eine große Stärke des Relationenmodells liegt sicherlich in der einfachen und einheitlichen Beschreibung der Anwendungsdaten. Allgemein stehen zur Datenmodellierung die Konzepte Attribut, Relationenschema, Integritätsbedingung, Schlüssel und Fremdschlüssel zur Verfügung.

Mögliche Datentypen für die Attribute sind in DB2 neben den üblichen Standard-Datentypen (Zeichenketten, Numerische Werte, Datum und Zeit etc.) auch sogenannte LOBs (Large Objects), die Zeichenketten bis zu einer Größe von 2 Gigabytes aufnehmen können. Mit diesem Datentyp soll es ermöglicht werden, komplette Texte, Bilder, Audio- oder Video-Dateien in der Datenbank abzuspeichern.

Grundsätzlich sind im Relationenmodell keine Typkonstruktoren vorhanden, so daß wie auch in allen anderen relationalen Datenbanken komplexe Attribute nicht direkt dargestellt werden können. Komlexe Attribute bestehen aus Wertemengen (z.B. kann eine Person mehrere Hobbys haben) oder aus mehreren Komponenten (Beispiel folgt). Um Redundanz bei Attributen mit Wertemengen zu vermeiden, müssen mehrere Relationen definiert werden. Mehrere Tabellen aber erhöhen den Aufwand bei der Formulierung von Anfragen und senken die Überschaubarkeit. Man möge sich vorstellen, was

passiert, wenn ein Wert einer Wertemenge wiederum eine Menge von Werten ist.

Komplexe Attribute, die aus mehreren Komponenten bestehen, müssen in einzelne Attribute zerlegt werden. So wäre für die Modellierung der Tarifdaten des D4 denkbar, ein einziges Attribut `BUZ` zu definieren, das sich aus den Komponenten `Beitragsbefreiung`, `Tarif`, `Ablaufalter` und `Barrente` zusammensetzt. Da dies nicht möglich ist, erhöht sich die Anzahl der Attribute innerhalb einer Relation.

Schlüssel sind sichtbar und veränderlich. Damit besitzen Tupel keine eigene „Identität“. Nun ist es allerdings vorstellbar, daß in der realen Welt Objekte durch äußere Eigenschaften nicht zu unterscheiden sind, sie aber eine eigene Identität besitzen. Um diese mit den Mitteln des Relationenmodells zu beschreiben, müßten weitere Kriterien betrachtet werden, die aber für die Anwendungsentwicklung unerheblich sein könnten.

Beziehungen werden im Relationenmodell in einer Relation mit Fremdschlüsseln dargestellt. Dabei ist es nicht möglich, die verschiedenen Semantiken der Beziehungen (z.B. Is-a-Beziehung und Objekt-Komponentenobjekt-Beziehung) zu unterscheiden. Das muß bei der Anfrageformulierung berücksichtigt werden.

Funktionen oder Prozeduren können im eigentlichen Modell in DB2 nicht mitdefiniert werden. Sie können nur von außen (z.B. in einer Anwendung, die die Datenbank benutzt) nachträglich aufgesetzt werden.

8.1.2 O_2

Mit den objektorientierten Konzepten wie etwa Klassen, Einkapselung und Vererbung ist es in erster Linie auch rein intuitiv besser möglich, Sachverhalte der realen Welt möglichst getreu in ein Modell abzubilden. In diesem Abschnitt soll nun aber vorwiegend auf die angesprochenen Probleme des Relationenmodells eingegangen werden und gezeigt werden, daß diese Probleme mit objektorientierten Konzepten nicht eintreten.

In O_2 ist es erlaubt, Typkonstruktoren wiederholt anzuwenden. Damit ist es möglich, komplexe Attribute darzustellen. So könnte das Attribut `BUZ` der Klasse `Tarif`, um das Beispiel des vorangegangenen Abschnittes aufzugrei-

fen, in O_2 wie folgt definiert werden:

```
CLASS Tarif
   TYPE TUPLE ( Preisklasse: BYTE,
                ...
                BUZ: TUPLE ( Beitragsbefreiung: BOOLEAN,
                             Tarif: BYTE,
                             Ablaufalter: BYTE,
                             Barrente: INTEGER ))
```

Neben der besseren Übersichtlichkeit und Handhabung ergibt sich ein weiterer Vorteil: Gehört eine BUZ zu den Tarifdaten, sind die entsprechenden Komponenten des komplexen Attributes mit sinnvollen Werten gefüllt. Wird keine BUZ abgeschlossen, so ist bereits `BUZ` mit einem Nullwert belegt. Ohne Typkonstruktoren kann die Entscheidung, ob die einzelnen Komponenten sinnvolle Werte enthalten oder nicht, gar nicht so einfach getroffen werden. Aus diesem Grund mußte ein zusätzliches Attribut `BUZ_Abschliessen` (siehe Klassendefinition auf Seite 114) eingeführt werden, das lediglich als Flag benutzt wird.

Objektorientierte Systeme verwalten die Objekte über die Objektidentität, die von außen in der Regel unsichtbar ist und stets unveränderlich bleibt. Selbst wenn sich alle Eigenschaften eines Objektes ändern, bleibt es das gleiche Objekt. Ändern sich Teile des Schlüssels eines Tupels im Relationenmodell, so wird auch die Identität gewechselt.

Beziehungen können in objektorientieren Systemen im allgemeinen über Komponentenobjeke dargestellt werden. Eine Is-a-Beziehung wird über Vererbung erreicht. Deshalb ist in O_2 eine semantische Unterscheidung der Beziehungen möglich. Manche objektorientierte DBMS verfügen sogar über einen eigenen Objekttyp für Beziehungen, mit dem man analog zu ER-Modellen die teilnehmenden Klassen und deren Kardinalitäten angeben kann. Liegt der Datenbankentwurf als ER-Modell vor, dann ist die Abbildung in das Datenbankmodell noch einfacher. In O_2 wäre es möglich, einen solchen Objekttyp selbst zu definieren. Ob das allerdings sehr sinnvoll ist, muß am konkreten Anwendungszweck untersucht werden.

Methoden können ebenfalls im Modell definiert werden. Innerhalb der Klassendeklaration (oder auch später) werden die Signaturen der Methoden definiert. Die Methodenrümpfe werden immer außerhalb der Klassendeklaration definiert.

8.1.3 Fazit

In der Datenmodellierung haben relationale DBMS erhebliche Nachteile gegenüber objektorientierten DBMS. Neben den bereits angesprochenen Nachteilen des Relationenmodells, die in der Tabelle 8.1 zusammengefaßt sind, besitzt das objektorientierte Modell weitere Stärken für die Datenmodellierung, die an dieser Stelle nicht weiter ausgeführt, sondern nur genannt werden sollen:

- Klassen- und Typhierarchie
- Überladen und dynamisches Binden von Methoden

Nun besitzen einige Programmiersprachen, so auch Java, keine Typkonstruktoren, so daß deshalb die Schwäche in der Darstellung von komplexen Attributen in DB2 bei der Entwicklung des Prototypen auch kein direkter Nachteil gegenüber O_2 darstellt. Weiterhin spielen für den Prototypen die Objektidentität und die semantische Unterscheidung der Beziehungen keine Rolle.

Methoden können nur im objektorientierten Modell definiert werden, aber auch das hatte keinen Einfluß auf die Entwicklung des Prototypen.

8.2 Datenbankentwurf

Mit einem Datenbankentwurf wird eine gegebene Anwendungsstruktur in ein Datenbankschema überführt. Dabei sollten strukturelle und semantische Informationen möglichst erhalten bleiben.

Solche Informationen äußern sich z.B. in den Abhängigkeiten zwischen Attributen. Zwei wichtige Abhängigkeiten, die funktionale und mehrwertige Abhängigkeit, wurden bereits im Abschnitt 5.2 auf Seite 47 definiert. Die Entwurfstechniken des Relationenmodells können unter Umständen große

Schwächen von DB2 gegenüber O_2	Wirkung auf den Prototypen
Darstellung komplexer Attribute nur simulierbar	auch in Java nur simulierbar
keine Objektidentität	Definition von Schlüsseln notwendig
verschiedene Beziehungen darstellbar, aber nicht unterscheidbar	nicht nötig
keine Methoden im Modell definierbar	keine Auswirkung

Tabelle 8.1: Schwächen von DB2 in der Datenmodellierung

Probleme bei der Darstellung dieser Abhängigkeiten haben. Ausführliche Beispiele können [Heu97] entnommen werden.

Zur Normalisierung eines Relationenschemas (siehe Abschnitt 5.2) werden Dekompositions- und Synthesealgorithmen benutzt. Durch Dekomposition können Attribute eines Objekttyps auf mehrere Relationen verteilt werden. Die Synthese hingegen kombiniert unter Umständen Attribute verschiedener Objekttypen in eine Relation. Solche Schwierigkeiten existieren in objektorientierten Modellen nicht, der vollständige Objekttyp gehört stets zu einer Klasse.

Daher ist die Gefahr, strukturelle und semantische Informationen beim Datenbankentwurf zu verlieren, in relationalen DBMS wesentlich größer als in objektorientierten DBMS.

Die angesprochenen Probleme relationaler DBMS ergaben sich allerdings nicht bei der Überführung der Java-Schnittstelle in das Relationenmodell. So lag der Grund der Darstellung der Java-Klasse `Ergebnis` durch zwei Relationen ausschließlich an der fehlenden Domäne für Arrays bzw. ist ein Array zweidimensional und kann deshalb nicht in mehrere Attribute aufgeteilt werden.

Dennoch ist das entstandene Datenbankschema von DB2 nur mit mehr Aufwand zu verwenden als das von O_2. Das wurde besonders deutlich in der Implementierung der Methoden der Klasse `Datenbank` für die Datenbankzu-

griffe (siehe auch Anhang D).

8.3 Anfragesprache

In diesem Abschnitt sollen die Anfragesprachen hinsichtlich ihrer praktischen Verwendbarkeit kurz beleuchtet werden.

In DB2 wird SQL92, in O_2 wird OQL (nach ODMG-93 Standard) verwendet.

8.3.1 DB2

In [Heu97] werden folgende drei schwerwiegenden Probleme relationaler Anfragesprachen genannt:

- Strukturmangel im Ergebnis.
  Das Ergebnis einer Anfrage ist eine Relation. Die ursprüngliche Struktur der Anwendungsobjekte muß explizit rekonstruiert werden.

- Fehlende Unterstützung komplexer Strukturen in der Anfrageformulierung.

- Notwendigkeit expliziter Verbundoperationen.
  Um Informationen aus verschiedenen Relationen zu nutzen, muß bei der Anfrageformulierung ein Verbund vorgenommen werden. Das betrifft z.B. aber auch die Relationen, die eigentlich einen Objekttypen repräsentieren, der aber aufgrund der Dekomposition durch mehrere Relationen dargestellt wird.
  Treten Änderungen an solchen Objekttypen auf, müssen die entsprechenden Update-Operationen nacheinander auf jede einzelne Relation angewendet werden. Das wirkt sich vor allem durch größeren Aufwand auf die Implementierung der Datenbankzugriffe aus.

8.3.2 O_2

OQL ist eine SQL-artige Anfragesprache. Je nach Anfrage liefert OQL Werte, komplexe Attribute, Objekte oder Mengen von Objekten zurück; Verbunde

sind dabei möglich. OQL besitzt gegenüber SQL zwei wesentliche Vorteile:

- Komplexe Strukturen werden unterstützt.
- Mit Anfrageergebnissen lassen sich neue Anwendungstypen definieren.

8.3.3 Fazit

Viele Anfragen, die in einer informalen Sprache einfach zu formulieren sind, können in SQL oft gar nicht oder nur sehr umständlich formuliert werden. Daher bietet DB2 teilweise auch Spracherweiterungen an, um solche Probleme zu vermeiden. Da allerdings solche Probleme bei der Implementierung des Prototypen kaum auftraten, soll zu einer genaueren Analyse der Schwächen von SQL auf [Dat89] und [DD97] verwiesen werden.

8.4 Java-Binding

Das JB beider DBMS wurde in den Abschnitten 5.4 und 6.1 beschrieben. Dabei zeigten sich große Unterschiede.

8.4.1 Erforderliche Arbeitsschritte

Um Instanzen von Java-Klassen in einer Datenbank Persistenz zu verleihen, sind folgende drei Arbeitsschritte nötig.

1. Abbildung der Java-Klassen in das Datenmodell des DBMS

    - DB2
      Der Implementierer muß aus den Java-Klassen einen relationalen Datenbankentwurf durchführen. Dabei können Probleme entstehen, die bereits in den Abschnitten 5.2, 8.1.1 und 8.2 diskutiert wurden.
    - O_2
      Das Importier-Werkzeug übernimmt die Abbildung der Java-Klassen.

2. Anlegen des Datenbankschemas

   - DB2
     Der Implementierer sorgt selber für das Anlegen der Relationen nach dem Datenbankentwurf. Integritätsbedingungen müssen teilweise mit Constraints und Triggern modelliert werden.
   - O_2
     Das Importier-Werkzeug legt automatisch das Datenbankschema an, der Zugriff erfolgt dann transparent. Die Beziehungen zwischen den Java-Klassen bleiben erhalten.

3. Implementierung der Zugriffe auf die Daten der Datenbank

   - DB2
     Der Zugriff wird über JDBC realisiert. Der Implementierer muß die Anfragen als SQL-Anweisungen formulieren. Fehler in der Anfrageformulierung werden erst zur Laufzeit aufgedeckt. Auf Mängel von SQL wurde teilweise im Abschnitt 8.3.1 hingewiesen. Die JDBC-Komponenten sind im JDK enthalten.
   - O_2
     Der Zugriff wird komplett in Java mit Hilfe der Klassen des JB-API realisiert. Anfragen werden über den entsprechenden Klassenextent, der mit Prädikaten gefiltert werden kann, gestellt.

8.4.2 Fazit

Das JB von O_2 ist wesentlich komfortabler. Die Werkzeuge nehmen dem Implementierer viele Arbeitsschritte ab, der sich so nicht mehr um Datenbank-Angelegenheiten kümmern muß.

Große Aufmerksamkeit muß der Typkonvertierung in DB2 geschenkt werden. Dieser Aufwand wird dem Implementierer für O_2 durch das Import-Werkzeug abgenommen. Veränderungen am Datenmodell ziehen in DB2 enorme Änderungsarbeiten an den Zugriffsmethoden nach sich.

Ein weiterer Vorteil des JB von O_2 liegt in der Möglichkeit, Arrays in die Datenbank zu speichern. In DB2 war das nicht möglich.

Aufgrund von Softwareproblemen (siehe auch Abschnitt 8.6) war es zeitlich leider nicht möglich, das JB von O_2 hinsichtlich des Imports der Java-Schnittstelle in ein relationales DBMS zu testen. So konnten auch weitere angedachte Vergleiche zwischen den JB beider DBMS nicht erfolgen.

8.5 Export der Daten

Die Exportmöglichkeiten von DB2 wurden bereits im Abschnitt 7.2.3 untersucht. An dieser Stelle soll aber noch einmal erwähnt werden, daß Daten aus DB2 mittels einer Export-Funktion einiger Programmier-APIs in drei verschiedene Datei-Formate gebracht werden können. Mit dem Werkzeug „Command Center“ ist der Export der Daten ebenfalls möglich.

O_2 bietet weder in den Programmier-Schnittstellen noch mit den Werkzeugen eine Möglichkeit, Daten zu exportieren. Der Anwender muß also eine Exportfunktion selber innerhalb einer Anwendung implementieren.

8.6 Stabilität der Systeme

Die Implementierung der Datenbankzugriffe für DB2 erfolgte unter dem Betriebssystem Windows NT. DB2 und die Kommunikation über JDBC funktionierten auf Anhieb. Leider glückte keine Installation unter Solaris.

Die JB-Werkzeuge von O_2 gibt es zur Zeit nur für die Unix-Plattform. Durch die irrtümliche Verwendung von O_2-Komponenten unterschiedlicher Versionen lief O_2 einige Zeit sehr instabil. Weiterhin stellt O_2 auch an die Version von Solaris Anforderungen, so mußte eine neue Version des Betriebssystemes installiert werden, damit O_2 verläßlich arbeitete. Jedoch scheinen die JB-Werkzeuge noch einige Fehler zu enthalten. So lassen sich selbst die mitgelieferten Beispielklassen nicht immer importieren. Auch die Java-Schnittstelle des Prototypen ließ sich nur zeitweise importieren (in dieser Phase wurde die Methode zum Laden der Kenngrößen erfolgreich implementiert und getestet). Zur Zeit bricht das Import-Werkzeug den Importvorgang mit

	DB2	O_2	Wirkung auf Prototyp
Datenmodellierung	$\ominus$	$\oplus$	keine
Datenbankentwurf	$\ominus$	$\oplus$	mehr Aufwand bei Implementierung der Datenbankzugriffe für DB2
Anfragesprache	$\ominus$	$\oplus$	keine
Java-Binding	$\ominus$	$\oplus$	mehr Aufwand bei Implementierung der Datenbankzugriffe für DB2
Export der Daten	$\oplus$	$\ominus$	Exportfunktion für O_2 muß selber implementiert werden
aktive Komponente	$\ominus$	$\ominus$	Workflow-Komponente muß als eigene Anwendung realisiert werden

Tabelle 8.2: Zusammenfassung der Stärken und Schwächen beider DBMS

einem Java-Fehler ab. Auch der Support von O_2-Technology konnte bisher die Ursache dieses Fehlers nicht finden. Daher konnte die Implementierung zum Speichern eines Angebotes nicht getestet werden.

8.7 Fazit

Die Stärke des Relationenmodells liegt in der einheitlichen und einfachen Beschreibung der Anwendungsdaten. Dem Relationenmodell liegt ein exaktes mathematisches Modell zugrunde. Dennoch bestehen im Verhältnis zum objektorientierten Modell viele Defizite in der Datenmodellierung. Auch im Datenbankentwurf und bei der Anfragesprache bestehen einige Nachteile. Diese Defizite erlangten allerdings am Beispiel des Prototypen kaum an Bedeutung.

Die größten Unterschiede ergeben sich beim Java-Binding. Das JB von O_2 ist wesentlich komfortabler und senkt den Implementierungsaufwand für die Datenbankzugriffe erheblich. Außerdem gestaltet sich der Zugriff auf die Datenbank für den Implementierer wesentlich transparenter, er muß weder einen Datenbankentwurf, noch das Datenbankschema anlegen. Auch um die Konvertierung der Java-Typen in die OQL-Typen und umgekehrt muß sich der Implementierer nicht kümmern.

O_2 bietet keine Möglichkeit, Daten zu exportieren. Daten aus DB2 können in drei verschiedene Dateiformate exportiert werden. Damit sind auch die Möglichkeiten für die Realisierung der Workflow-Komponente für O_2 beschränkter bzw. muß eine Exportfunktion durch den Implementierer geschaffen werden.

Große Unterschiede ergaben sich auch beim Vergleich der aktiven Komponenten (siehe auch Abschnitt 7.2.1). In DB2 werden z.B. nur SQL-Operationen auf vorhandenen Relationen als Ereignis erkannt. In O_2 ist es möglich, den Vorgang des Verbindungsaufbaues bzw. -abbaues eines Clients zum Server als Ereignis zu definieren. Weiterhin können Ereignisse jeglicher Art definiert werden, die dann allerdings nicht vom System selbständig erkannt werden. In DB2 können Aktionen definiert werden, die nach dem Eintreten eines Ereignisses und dem Überprüfen einer Bedingung ausgelöst werden. Als Aktionen sind allerdings nur SQL-Anweisungen möglich. In O_2 werden nach dem Erkennen eines eintretenden Ereignisses lediglich Nachrichten verschickt, die Clients über das Eintreten des Ereignisses informieren. Der Client muß deshalb selber dafür sorgen, daß eine bestimmte Aktion ausgeführt wird.

Die Schwerpunkte, die in diesem Kapitel untersucht und diskutiert wurden, sind nochmals in Tabelle 8.2 zusammengefaßt.

Auch wenn bei der Entwicklung des Prototypen nicht alle Vorteile des objektorientierten Modells gegenüber dem Relationenmodell zur Geltung kamen, gestaltete sich der Implementierungsaufwand der Datenbankanbindung von O_2 wesentlich einfacher als der von DB2. Es soll nochmals betont werden, daß sich die Schwächen des Relationenmodells in puncto Datenmodellierung, Datenbankentwurf und Anfragesprache bei komplexeren Anwendungsbeispielen sehr nachteilig gegenüber der Verwendung eines objektorientierten DBMS auswirken können.

Kapitel 9

Generieren von HTML-Seiten aus einer Datenbank

In diesem Kapitel sollen einige Möglichkeiten der dynamischen Generierung von HTML-Seiten kurz erörtert werden. Diese Seiten sind nicht statisch auf einem Web-Server gespeichert, sondern werden von einem Programm auf dem Server bei jedem Zugriff durch den Browser neu erstellt. Dabei steht die Abwägung, inwieweit diese Techniken alternativ beim Entwurf des Prototypen genutzt werden können, im Vordergrund.

9.1 CGI

Das Common Gateway Interface (kurz CGI) ist der Versuch der Standardisierung der Kommunikation eines Web-Servers mit einer SQL-Datenbank.

CGI-Programme können:

- Daten aus einem HTML-Formular in eine Datenbank speichern,
- Daten aus einer Datenbank abrufen und damit
- HTML-Seiten dynamisch erzeugen.

CGI-Programme können mit Skriptsprachen (z.B. Perl) oder mit Programmiersprachen entwickelt werden. Bei der Gestaltung der Formulare ist man jedoch auf die Möglichkeiten von HTML beschränkt.

Der Ablauf der Verwendung eines CGI-Programmes gestaltet sich wie folgt:

1. Der Anwender klickt in seinem Browser auf ein URL, der ein Programm anstatt einer HTML-Seite repräsentiert.
2. Der HTTP-Server, auf den der URL verweist, ruft das Programm mit den durch den URL übergebenen Parametern auf.
3. Das Programm speichert eventuell Daten in eine Datenbank.
4. Das Programm erzeugt eine HTML-Seite, die an den Web-Server gesendet wird.
5. Der Web-Server sendet die neu erstellte HTML-Seite zum Browser.

Der Nachteil des Einsatzes von CGI-Programmen liegt in der zusätzlichen Belastung des Servers durch den Verbrauch von Prozessorzeit und Speicher. Bei Verwendung von Perl kann laut [HvS97] von einem Speicherverbrauch von mindestens 1 MByte ausgegangen werden.

In [Blu97] werden weitere Nachteile genannt. So bietet CGI eine mangelnde Unterstützung für viele externe Programmentwicklungs-Tools, Betriebssysteme und Umgebungen. Die Datenübertragungsmethode ist weder robust noch effizient. Für allgemein gebräuchliche Aufgaben stehen nur Schnittstellen auf einer unnötig tiefen Ebene zur Verfügung.

Wegen dieser Nachteile von CGI gibt es auch verschiedene Lösungsansätze (z.B. der ISAPI-Standard von Microsoft, Livewire von Netscape oder Servlet-Technologie von Sun Microsystems), die diese Probleme vermeiden oder den Programmieraufwand auf Seiten des Web-Administrators kleiner halten wollen. Bei der Verwendung dieser Lösungen wird allerdings die Plattformunabhängigkeit verloren.

9.2 O_2 Web

Für den Zugriff auf O_2 direkt aus dem WWW bietet O_2-Technology das Produkt O_2 Web an [Man03]. Wenn ein Web-Server auf O_2 Web aufgebaut

ist, kann eine OQL-Anfrage, die der Client mit dem URL übergeben kann, zum O_2-Server geleitet werden. Die Anfrageergebnisse werden durch automatisch generierte HTML-Seiten dargestellt, die dann anschließend zum Client gesendet werden.

Dem Anwender stehen drei Niveauebenen von Einflußmöglichkeiten auf die HTML-Generierung zur Verfügung:

1. Der Anwender übt keinerlei Einfluß auf die HTML-Generierung aus. Anfrageergebnisse werden objektweise dargestellt. Jedes Objekt besitzt seine eigene HTML-Seite, auf der zeilenweise die Attributnamen und -werte gelistet werden. Über Links kann sich der Anwender durch alle Objekte navigieren.

2. Die Objekte werden wie auf der 1. Niveauebene dargestellt. Zusätzlich kann aber Text am Anfang und Ende der HTML-Seite eingefügt werden. Weiterhin kann auf Fehler oder Ereignisse (wie z.B. Verbindungsauf- und abbau zum Server) reagiert werden.

3. Der Anwender erhält die totale Kontrolle über die HTML-Generierung. HTML-Code, der die Darstellung der Objekte beschreibt, kann in die Klassendefinition eingebettet werden, daher kann sogar die Darstellung der Objekte in Abhängigkeit von Attributwerten beeinflußt werden. Natürlich kann auch auf dieser Niveauebene zusätzlicher Text am Anfang und Ende der HTML-Seite eingefügt werden.

9.3 Fazit

Die Verwendung von CGI-Programmen oder O_2 Web wäre für die Implementierung des Prototypen ebenso möglich gewesen.

Allerdings ist man beim Layout der Formulare und Anfrageergebnisse auf die Möglichkeiten von HTML beschränkt. Ein weiterer Nachteil ist die größere Serverlast durch das Generieren von HTML-Seiten. Gerade bei Verwendung von CGI sollen laut [Blu97] Probleme auftreten, wenn zeitgleich mehrere Anwender auf die Datenbank zugreifen. Allerdings bietet CGI als einzige der genannten Möglichkeiten den Erhalt der Plattformunabhängigkeit.

Ferner ist zu bedenken, daß viele Skriptsprachen nicht die Mächtigkeit von Programmiersprachen besitzen.

Der große Vorteil solcher Lösungen besteht sicherlich in der automatischen Präsentation unterschiedlicher Anfrageergebnisse durch HTML-Seiten. Der Prototyp ruft allerdings lediglich die aktuelle Instanz der Klasse `Kenngroessen` aus der Datenbank ab, die dem Anwender nicht präsentiert werden soll, sondern nur zur Parameterüberprüfung für die Applikation nötig ist. Dem Anwender wird nur das Ergebnis seines Angebotes präsentiert, das zu diesem Zeitpunkt noch nicht persistent ist. Erst zu einem späteren Zeitpunkt wird das Ergebnis innerhalb des kompletten Angebotes (sie dazu Kapitel 4) in der Datenbank gespeichert.

Die Darstellung der kompletten Angebote für die Sachbearbeiter gehört nicht zum geforderten Aufgabenbereich des Prototypen.

Kapitel 10

Schlußbetrachtung

10.1 Die Architektur der Applikation

Um den Überblick zum Prototypen zu vervollständigen, wird in diesem Abschnitt die Architektur der Applikation erläutert. Genauere Informationen zur Applikation können [Bas98] entnommen werden.

Der Aufbau der Applikation wird in der Abbildung 10.1 dargestellt.

Der interessierte Kunde startet auf seinem lokalen Rechner über einen Web-Browser ein Java-Applet, daß alle nötigen Parameter aufnimmt und diese der laufenden Java-Applikation auf dem Web-Server mitteilt. Auf dem Web-Server wird für jeden Client eine eigene Task generiert. Diese Task läßt die Berechnungskomponente mit den übergebenen Parametern die Ergebniswerte berechnen und liest oder schreibt die entsprechenden Daten in oder aus der Datenbank über eine Java-Schnittstelle.

Die Berechnungskomponente ist ein bereits in Nutzung befindliches MS-Excel-Modul. Möglichkeiten des Datenaustausches sind in [Bas98] beschrieben.

Eine Workflow-Komponente generiert automatisch E-Mails und informiert so die entsprechenden Sachbearbeiter über eingegangene Angebote.

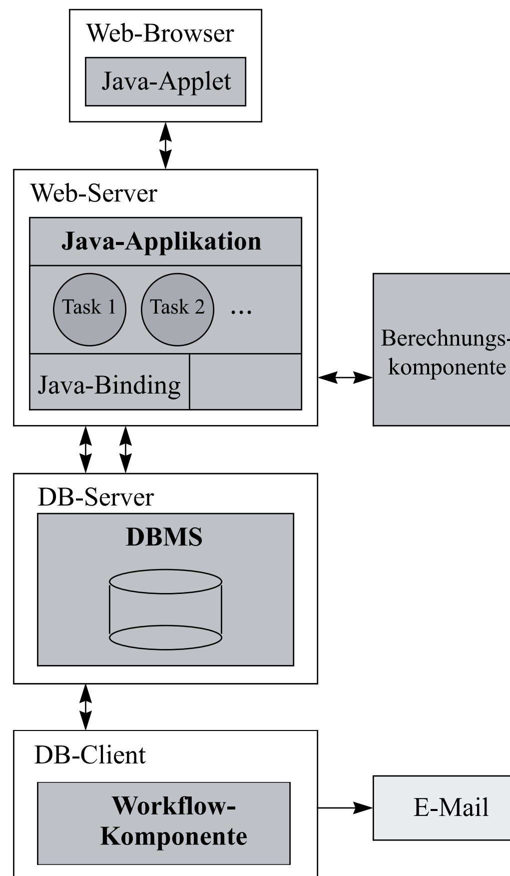

Abbildung 10.1: Die Applikation

10.2 Zusammenfassung

Ziel dieser Arbeit und der Diplomarbeit von M. Basler [Bas98] war die Konzeption und Implementierung eines Prototypen, der eine Angebotserstellung einer Kapital- und Lebensversicherung per WWW ermöglicht. Als Implementierungssprache wurde Java wegen der Plattformunabhängigkeit und des Sicherheitskonzeptes gewählt.

Nach einer Analyse des konkreten Tarifmodells wurden dessen relevante Informationen bestimmt und in einem allgemeingültigen Datenmodell dargestellt. Aus diesem Datenmodell wurde eine Java-Schnittstelle definiert, die einen einheitlichen und tranparenten Zugriff auf das relationale DBMS DB2 und das objektotientierte DBMS O_2 erlaubt. Aufgrund des komfortableren JB von O_2 ist der Implementierungsaufwand der Methoden für die Datenbankzugriffe für O_2 wesentlich geringer als der für DB2.

Es erfolgte ein bewertender Vergleich beider DBMS bezüglich ihrer Eignung für den Prototypen. Nicht alle Stärken und Vorteile eines objektorientierten gegenüber eines relationalen DBMS kamen für den Protoypen zum Tragen.

Es wurden Möglichkeiten der Umsetzung der Workflow-Komponente diskutiert. Die aktiven Komponenten beider DBMS reichen leider nicht aus, die Workflow-Komponente zu realisieren. Alternativ wurde eine Java-Anwendung beschrieben, die auf die Schnittstelle aufbauend und mit Verwendung von frei verfügbaren E-Mail-Clients alle Anforderungen an die Workflow-Komponente erfüllt.

10.3 Ausblicke

Hohe Priorität lag bei der Darstellung des Machbaren und Diskussion der Eignung vorhandener Techniken. Dabei kristallisierten sich folgende Kernpunkte heraus.

- Als Programmiersprache wird Java immer bedeutender werden.
  Mit den stetigen Verbesserungen und neuen Möglichkeiten von Java (siehe [FEY98, Sch98, Tie98]) ist Java längst über den Status als Mittel zur Erstellung von „netten Effekten“ auf Homepages hinausgewachsen.

Deshalb haben bereits einige Softwarehäuser angekündigt, daß sie auch große Projekte in Zukunft mit Java realisieren werden.

- Objektorientierte DBMS besitzen viele Vorteile gegenüber relationalen DBMS. Daher zeichnet sich ein Trend ab, relationale DBMS um objektorientierte Konzepten zu erweitern (Objektrelationale Datenbanken). Einige Nachteile von DB2 (z.B. das Fehlen einer Objektidentität) gegenüber O_2 könnten dadurch aufgehoben werden.

- Die Modellierung komplexerer Geschäftsprozesse als das Versenden von E-Mails an die Sachbearbeiter sind denkbar (siehe auch [Rui97]).

- Es wäre äußerst wünschenswert, daß die aktiven Komponenten der DBMS bezüglich Ereignisdefinition und -erkennung bzw. der Definition der Aktion komfortabler und mächtiger werden. Sie könnten dann auch dazu beitragen, wesentlich komplexere Geschäftsprozesse zu modellieren.

- Um die Stellung von O_2 als Alternative zu relationalen DBMS auszubauen, ist die Robustheit des gesamten Systems zu heben. Das JB sollte in Zukunft OMG-konform gestaltet werden, noch gibt es einige Unterschiede (siehe [CB98]).

Abkürzungen

API	Application Programming Interface
AVSG	Allgemeine Versicherungs-Software GmbH
BLOB	Binary Large Object
BUZ	Berufsunfähigkeits-Zusatzversicherung
CGI	Common Gateway Interface
CLI	(X/Open) Call Level Interface
CORBA	Common Object Request Broker Architecture
DB	Datenbank
DB2 CLI	DB2 Call Level Interface
DBMS	Datenbankmanagementsystem
DDL	Data Definition Language
ECA-Regel	Event-Condition-Action - Regel
EFS	Erlebensfallsumme
FIFO	First In First Out
HTML	Hypertext Markup Language
JAF	JavaBeans Activation Framework
JB	Java-Binding
JDBC	Java Database Connectivity
JDK	Java Development Kit
JIT	Just In Time
LOB	Large Object
ODMG	Object Database Management Group
OQL	Object Query Language
ORB	Object Request Broker
PDA	Personal Digital Assistent
RDB	Relationale Datenbank

SQL	Structured Query Language
TFS	Todesfallsumme
UDF	User Defined Function
URL	Uniform Resource Locator
VM	Virtual Machine
WWW	World Wide Web

Abbildungsverzeichnis

3.1 Das Modell des D4 30

4.1 Ausgabefenster der Kenngrößen 39
4.2 Fenster zur Angebotserstellung 40

5.1 Zwei-Ebenen-Architektur 49
5.2 Drei-Ebenen-Architektur 49
5.3 JDBC-Applet für DB2 57

6.1 Allgemeine JB-Architektur von O_2 62
6.2 Importieren einer Java-Klasse in O_2 64

7.1 Der Notification Service von O_2 74

10.1 Die Applikation 98

Tabellenverzeichnis

3.1 Beispiel für die Ergebnistabelle eines Angebotes 28

8.1 Schwächen von DB2 in der Datenmodellierung 86
8.2 Zusammenfassung der Stärken und Schwächen beider DBMS . 91

Literaturverzeichnis

[Bas98] Matthias Basler: **Konzeption und Implementierung eines Prototypen zur Präsentation von Versicherungsdiensten im WWW**; Diplomarbeit, TU Ilmenau, 1998

[Blu97] Adam Blum: **ActiveX**; 1. Auflage, Sybex-Verlag GmbH, 1997

[Dat89] C. Date: **A Guide to the SQL-Standard**; 2. Auflage, Addison-Wesley, 1989

[Dic97] Hans Dicken: **JDBC. Internet-Datenbankanbindung mit Java**; 1. Auflage, International Thomson Publishing, 1997

[DD97] C. Date, H. Darwen: **A Guide to the SQL-Standard**; 4. Auflage, Addison-Wesley, 1997

[DG96] Klaus R. Dittrich, Stella Gatziu: **Aktive Datenbanksysteme. Konzepte und Mechanismen.**; 1. Auflage, International Thomson Publishing, TAT13, 1996

[FEY98] Jürgen Fey: **Sortenvielfalt**; c't 10/98, S. 234, Verlag Heinz Heise GmbH & Co KG, 1998

[HCF97] G. Hamilton, R. Cattell, M. Fisher: **JDBC Database Access with Java**; 1. Auflage, Addison-Wesley, 1997

[Heu97] Andreas Heuer: **Objektorientierte Datenbanken**; 2. Auflage, Addison-Wesley, 1997

[HS95] Andreas Heuer, Gunter Saake: **Datenbanken: Konzepte und Sprachen.**; 1. Auflage, International Thomson Publishing, 1995

[HvS97] Jan Hermelink, Georg von Stein: **Von SQL nach HTML**; Gateway 11/97, Verlag Heinz Heise GmbH & Co KG, 1998

[Int97] **Der Tarif D4**, Interne Papiere der Gothaer Versicherungen a.G.

[JBS97] Stefan Jablonski, Markus Böhm, Wolfgang Schulze (Hrsg.): **Workflow-Management. Entwicklung von Anwendungen und Systemen**; 1. Auflage, dpunkt.verlag, 1997

[Mor97] Michael Morrison: **Java 1.1 für Insider**; 1. Auflage, SAMS Verlag, 1997

[New97] Alexander Newman u.a.: **Java: Referenz und Anwendung**; 1. Auflage, Que, 1997

[CB98] R. Cattell, D.K. Barry: **The Object Database Standard ODMG 2.0**; Morgan Kaufmann Publishers, Inc., 1998

[Rui97] Tania Ruiz: **Integration von Workflow-Management-Funktionalität in eine CORBA-basierte CSCW-Umgebung**; Diplomarbeit, Universität Rostock, 1997

[Sch98] Hajo Schulz: **Orakelsprüche**; c't 10/98, S. 44, Verlag Heinz Heise GmbH & Co KG, 1998

[Tie98] Eric Teiling: **Gezähmte Monster**; c't 10/98, S. 226, Verlag Heinz Heise GmbH & Co KG, 1998

[Ull88] J. D. Ullmann: **Principles of Database and Knowledge-Base Systems**; Band 1, Computer Science Press, 1988

WWW-Verweise

[WWW01] `http://www.javasoft.com/docs/index.html`

[WWW02] `http://www.software.ibm.com/data/db2/udb/`

[WWW03] `http://www.o2tech.fr/`

[WWW04] `http://home.pi.net/~hverbeek/SendMail.java`

[WWW05] `http://home.pi.net/~hverbeek/SendMIME.java`

O_2-Manuals

[Man01] Ardent Java Bindings for O_2 and Relational Databases
`(lokal)/doc/PDF/o2javabd.pdf`

[Man02] O_2 Notification User Manual
`(lokal)/doc/PDF/o2notif.pdf`

[Man03] O_2 Web User Manual
`(lokal)/doc/PDF/o2webo2c.pdf`

DB2-Manuals

[ManDB2] `(local)/docs/HTML/index.htm`

Verwendete Software

Objektorientierter Modellentwurf:

- Object Domain 1.19a von Object Domain Systems

Implementierung der Applets:

- JDK 1.1.5
- Standardeditoren der jeweiligen Betriebssysteme

Implementierung der HTML-Dokumente:

- Standardeditoren der jeweiligen Betriebssysteme, getestet mit
  - Netscape Navigator 4.0 (mit Java 1.1 Plug-In)
  - Internet-Explorer 4.01

Anhang A

Die Java-Schnittstelle

```
public class Person {
   public byte         Anrede;            // 0=Herr, 1=Frau, 2=Firma
   public String       Name;
   public String       Vorname;
   public String       Geburtsdatum;
   public byte         Geschlecht;        // 0=maennlich, 1=weiblich
   public String       Strasse;           // incl. Hausnr.
   public String       Ort;
   public String       Plz;
   public String       Telefon;
   public String       Fax;
   public String       Email;

   public Person() {};
}


public class Tarif {
   public byte         Preisklasse;
                       // Code: 0=E, 1=K, 2=G, 3=H
   public byte         Laufzeit;
   public byte         Eintrittsalter;
   public String       Versicherungsbeginn;
   public boolean      Aufloesungsoption;
   public byte         Dauer_Aufloesungsphase;
   public byte         Zahlweise_Beitrag;
                       // Code: 0=monatlich, 1=viertel-, 2=halb-,
                       //  3=jährlich, 4=Depot
   public byte         Berechnungsart;    // Code: 0=Beitrag, 1=VS
   public int          Vorgabewert;
   public boolean      Abgekuerzte_Zahlung;
   public byte         Beitragszahlungsdauer;
   public byte         Gewinnsystem;      // Code: 0=BE, 1=BR
   // Leistungsverlauf
```

```
  public byte        Leistungsverlauf;
  public int         TFS_Beginn;
  public int         TFS_Ende;
  public byte        TFS_konstant;
  // Dynamik
  public boolean     Dynamik;            // true=ja, false=nein
  public byte        Kennzeichen_Dynamik;
                     // Code: 0=Tod und Erleben, 1=Erleben
  public byte        Rhythmus;
                     // Code: 0=ein-, 1=zwei-, 2=dreijährig
  public byte        Erhoehung_um;
  // Berufsunfähigkeits-Zusatzversicherung (BUZ)
  public boolean     BUZ_Abschliessen;   // true=ja, false=nein
  public boolean     BUZ_Beitragsbefreiung;
                     // true=ja, false=nein
  public byte        BUZ_Tarif;          // Code: 0=BJ, 1=BJr
  public byte        BUZ_Ablaufalter;
  public int         BUZ_Barrente;

  public Tarif() {};
}


public class Kenngroessen {
  public String      Gueltig_seit;
  public byte        Aufloesungsphase [];
  public int         Versicherungssumme [];
  public int         TFS_Anfang [];
  public int         TFS_Ende [];
  public int         Beitrag [];
  public byte        Zahldauer [];
  public byte        Dynamik_Wert [];
  public byte        Dynamik_Rhythmus [];
  public byte        BUZ_Ablaufalter [];

  public Kenngroessen() {};
}


public class Ergebnis {
  public String      Tarif;
                     // abschließende Tarifbezeichnung
  public int         BUZ_Aufschlag;
  public int         Tabelle[][];
                     // Tabelle mit den Rückkaufswerten etc.

  public Ergebnis() {};
}
```

```
public class Angebot {
   public Person       Personendaten;      // Angaben zur Person
   public Tarif        Tarifdaten;         // Tarifdaten
   public Ergebnis     Ergebnisdaten;      // Ergebnis der Berechnung
   // Verwaltungsdaten:
   public int          Angebotsnummer;
   public String       Datum;
                       // Datum der Angebotserstellung

   public Angebot() {};

   public void AbleitbareAttribute(){

      //Wenn der Versicherungsbeginn vor dem Geburtstag in diesem Jahr
      //liegt, dann muß das Eintrittsalter zusätzlich zu Vers.Beginn -
      //Geburtsjahr um 1 Jahr verkleinert werden.
      Tarifdaten.Eintrittsalter = (byte)(new Integer(
        Tarifdaten.Versicherungsbeginn.substring(0,4)).intValue() -
        new Integer(Personendaten.Geburtsdatum.substring(0,4)).intValue());
      if ((Personendaten.Geburtsdatum.substring(5,7).compareTo(
        Tarifdaten.Versicherungsbeginn.substring(5,7))>0) ||
        ((Personendaten.Geburtsdatum.substring(5,7) ==
        Tarifdaten.Versicherungsbeginn.substring(5,7)) &&
        (Personendaten.Geburtsdatum.substring(8,10).compareTo(
        Tarifdaten.Versicherungsbeginn.substring(8,10))>0)))
      { Tarifdaten.Eintrittsalter -=1;}

      if (Tarifdaten.Erhoehung_um > 0) {Tarifdaten.Dynamik = true;}
      else Tarifdaten.Dynamik = false;

      if (Personendaten.Anrede < 2)
        {Personendaten.Geschlecht = Personendaten.Anrede;}
   }
}


public class Datenbank {
   public void saveAngebot(Angebot angebot) {}

   public Kenngroessen getKenngroessen(String d) {
   Kenngroessen kenngroessen = new Kenngroessen();
   return (kenngroessen);
   }
   public Datenbank() {};
}
```

Anhang B

Definitionen der Relationen für DB2

```
CREATE TABLE Kenngroessen (
      Gueltig_seit VARCHAR(8) NOT NULL,
      Aufloesungsphase1 VARCHAR (3),
      Aufloesungsphase2 VARCHAR (3),
      VS1 VARCHAR (3),
      VS2 VARCHAR (3),
      TFS_Anfang1 VARCHAR (3),
      TFS_Anfang2 VARCHAR (3),
      TFS_Ende1 VARCHAR (3),
      TFS_Ende2 VARCHAR (3),
      Beitrag1 VARCHAR (3),
      Beitrag2 VARCHAR (3),
      Zahldauer1 VARCHAR (3),
      Zahldauer2 VARCHAR (3),
      Dyn_Wert1 VARCHAR (3),
      Dyn_Wert2 VARCHAR (3),
      Dyn_Rhythmus1 VARCHAR (3),
      Dyn_Rhythmus2 VARCHAR (3),
      BUZ_Ablaufalter1 VARCHAR (3),
      BUZ_Ablaufalter2 VARCHAR (3),
      PRIMARY KEY (Gueltig_seit))

CREATE TABLE Person (
      ID_Person INTEGER NOT NULL,
      Anrede SMALLINT,
      Name VARCHAR (20),
      Vorname VARCHAR (20),
```

```
      Geburtsdatum VARCHAR (8),
      Strasse VARCHAR (20),
      Ort VARCHAR (20),
      PLZ SMALLINT,
      Telefon VARCHAR (15),
      Fax VARCHAR (15),
      Email VARCHAR (30),
      PRIMARY KEY (ID_Person),
      CONSTRAINT ID_UNIQ UNIQUE (ID_Person))

CREATE TABLE Angebot (
      Angebotsnummer Integer NOT NULL,
      Datum VARCHAR(8),
      ID_Person INTEGER NOT NULL,
      PRIMARY KEY (Angebotsnummer),
      CONSTRAINT ANR_UNIQ UNIQUE (Angebotsnummer),
      CONSTRAINT FK_IDP FOREIGN KEY (ID_Person)
      REFERENCES Person (ID_Person) ON DELETE CASCADE )

CREATE TABLE Tarif (
      Angebotsnummer Integer NOT NULL,
      Preisklasse SMALLINT,
      Laufzeit SMALLINT,
      VSbeginn VARCHAR (8),
      Aufloesungsoption VARCHAR (1),
      Dauer_AP SMALLINT,
      Zahlweise_Beitrag SMALLINT,
      Berechnungsart SMALLINT,
      Vorgabewert INTEGER,
      Abg_Zahlung VARCHAR (1),
      Beitragszahldauer SMALLINT,
      Gewinnsystem SMALLINT,
      Leistungsverlauf SMALLINT,
      TFS_Beginn INTEGER,
      TFS_Ende INTEGER,
      TFS_konstant SMALLINT,
      Kennzeichen_D SMALLINT,
      Rhythmus SMALLINT,
      Erhoehung_um VARCHAR (5),
      BUZ_Abschliessen VARCHAR(1),
      BUZ_Beitragsbefr VARCHAR (1),
      BUZ_Tarif SMALLINT,
      BUZ_Ablaufalter SMALLINT,
```

```
      BUT_Barrente SMALLINT,
      CONSTRAINT FK_ANR FOREIGN KEY (Angebotsnummer)
      REFERENCES Angebot (Angebotsnummer) ON DELETE CASCADE )

CREATE TABLE Ergebnis (
      Angebotsnummer Integer NOT NULL,
      Tarif VARCHAR (10),
      BUZ_Aufschlag INTEGER,
      CONSTRAINT ANR_UNIQ UNIQUE (Angebotsnummer),
      CONSTRAINT FK_ANR FOREIGN KEY (Angebotsnummer)
      REFERENCES Angebot (Angebotsnummer) ON DELETE CASCADE )

CREATE TABLE Ergebnistabelle (
      Angebotsnummer Integer NOT NULL,
      VD SMALLINT,
      Beitrag INTEGER,
      EFS INTEGER,
      TFS INTEGER,
      RKW INTEGER,
      CONSTRAINT FK_ANR FOREIGN KEY (Angebotsnummer)
      REFERENCES Ergebnis (Angebotsnummer) ON DELETE CASCADE )
```

Wenn ein Angebot gelöscht wird, muß überprüft werden, ob die Person auch noch weitere Angebote erstellt hat. Ansonsten kann auch die Person gelöscht werden. Dazu wurde folgender Trigger definiert.

```
CREATE TRIGGER Delete_Person
AFTER DELETE ON Angebot
FOR EACH ROW MODE DB2SQL
DELETE FROM Person WHERE ID_Person NOT IN
      (SELECT ID_Person FROM Angebot GROUP BY ID_Person)
```

Anhang C

Die Konfigurations-Datei für den Import der Java-Klassen in O_2

Das Import-Werkzeuges `o2jb_import` muß mit dem zusätzlichen Parameter `-config` aufgerufen werden, um die folgende Konfigurations-Datei zu verwenden.

```
# transiente Felder
Transient Person: Geschlecht;
Transient Tarif: Eintrittsalter, Dynamik;
```

Sollen Java-Strings unbegrenzter Länge auf eine Zeichenkette mit bestimmter Länge gekürzt werden, dann muß z.B. folgende Zeile in die Konfigurations-Datei übernommen werden:

```
VARCHAR 20 Person: Name;
```

Sollen in der Datenbank Zeichenketten unbegrenzter Länge gespeichert werden, so kann das mit folgender Zeile erreicht werden:

```
TEXT Person: Name;
```

Anhang D

Die Implementierung der Methoden für die Datenbankzugriffe

D.1 DB2

```
package DB_Zugriff;

import java.lang.*;
import java.util.*;
import java.net.URL;
import java.sql.*;
import java.io.*;

public class Datenbank {

  String url = ''jdbc:db2:AVSG1'';
  String user = ''???'';
  String password = ''???'';

  public void saveAngebot(Angebot angebot) {

    try {
      OutputStream outFile = new FileOutputStream (''jdbc.out'');
      PrintStream outStream = new PrintStream (outFile, true);
      DriverManager.setLogStream (outStream);
    }
    catch (java.io.IOException ioex){ioex.printStackTrace();}

    try {
      System.out.println (''Lade JDBC-Treiber...'');
      Class.forName (''COM.ibm.db2.jdbc.app.DB2Driver'');
```

```
System.out.println (''JDBC-Treiber geladen.'');
System.out.println (''Baue Verbindung zur Datenbank auf.'');
Connection con = DriverManager.getConnection (url, user, password);
System.out.println (''Verbindung steht. Sende Anfrage.'');

con.setAutoCommit(false);
Statement PersonVorhanden = con.createStatement();
Statement AngebotInsert = con.createStatement();
int ID_Person;

ResultSet rs = PersonVorhanden.executeQuery(
    ''SELECT ID_Person FROM Person '' +
    ''WHERE Name = ''‘'' + angebot.Personendaten.Name + ''‘'' +
    ''; AND Vorname = ''‘,'' +
      angebot.Personendaten.Vorname + ''‘'' +
    ''; AND Geburtsdatum = ''‘'' +
      angebot.Personendaten.Geburtsdatum + ''‘'' +
    ''; AND Plz = '' + angebot.Personendaten.Plz +
    ''; AND Strasse = ''‘'' +
      angebot.Personendaten.Strasse + ''‘'');
boolean mehr = rs.next();
if (mehr) {ID_Person = rs.getInt(1);
  System.out.println(''Person schon vorhanden.'');
}
else {
  ID_Person = (int)System.currentTimeMillis();
  AngebotInsert.executeUpdate (''INSERT INTO Person '' +
      ''VALUES ('' + ID_Person + '', '' +
      angebot.Personendaten.Anrede + '', '' +
      ''‘'' + angebot.Personendaten.Name + ''‘, '' +
      ''‘'' + angebot.Personendaten.Vorname + ''‘, '' +
      ''‘'' + angebot.Personendaten.Geburtsdatum + ''‘, '' +
      ''‘'' + angebot.Personendaten.Strasse + ''‘, '' +
      ''‘'' + angebot.Personendaten.Ort + ''‘, '' +
      new Integer(angebot.Personendaten.Plz) + '', '' +
      ''‘'' + angebot.Personendaten.Telefon + ''‘,'', '' +
      ''‘'' + angebot.Personendaten.Fax + ''‘, '' +
      ''‘'' + angebot.Personendaten.Email + ''‘'')'');
  System.out.println(''Person eingefügt.'');
}
PersonVorhanden.close();
rs.close();

AngebotInsert.executeUpdate (''INSERT INTO Angebot '' +
    ''VALUES ('' + angebot.Angebotsnummer + '', '' +
    ''‘'' + angebot.Datum + ''‘, '' +
    ID_Person + '')'');
System.out.println(''Angebot'');
```

```
String Aufloesungsoption = new String();
String Abgek_Z = new String();
String BUZ_Abschliessen = new String();
String BUZ_Beitragsbefreiung = new String();
if (angebot.Tarifdaten.Aufloesungsoption)
{Aufloesungsoption = ''1'';}
else{Aufloesungsoption = ''0'';}
if (angebot.Tarifdaten.Abgekuerzte_Zahlung){Abgek_Z = ''1'';}
else{Abgek_Z = ''0'';}
if (angebot.Tarifdaten.BUZ_Abschliessen){BUZ_Abschliessen = ''1'';}
else{BUZ_Abschliessen = ''0'';}
if (angebot.Tarifdaten.BUZ_Beitragsbefreiung)
{BUZ_Beitragsbefreiung = ''1'';}
else{BUZ_Beitragsbefreiung = ''0'';}
AngebotInsert.executeUpdate (''INSERT INTO Tarif '' +
    ''VALUES ('' + angebot.Angebotsnummer + '', '' +
    angebot.Tarifdaten.Preisklasse + '', '' +
    angebot.Tarifdaten.Laufzeit + '', '' +
    ''`'' + angebot.Tarifdaten.Versicherungsbeginn + ''`, '' +
    ''`'' + Aufloesungsoption + ''`, '' +
    angebot.Tarifdaten.Dauer_Aufloesungsphase + '', '' +
    angebot.Tarifdaten.Zahlweise_Beitrag + '', '' +
    angebot.Tarifdaten.Berechnungsart + '', '' +
    angebot.Tarifdaten.Vorgabewert + '', '' +
    ''`'' + Abgek_Z + ''`, '' +
    angebot.Tarifdaten.Beitragszahlungsdauer + '', '' +
    angebot.Tarifdaten.Gewinnsystem + '', '' +
    angebot.Tarifdaten.Leistungsverlauf + '', '' +
    angebot.Tarifdaten.TFS_Beginn + '', '' +
    angebot.Tarifdaten.TFS_Ende + '', '' +
    angebot.Tarifdaten.TFS_konstant + '', '' +
    angebot.Tarifdaten.Kennzeichen_Dynamik + '', '' +
    angebot.Tarifdaten.Rhythmus + '', '' +
    ''`'' + String.valueOf(
        angebot.Tarifdaten.Erhoehung_um) + ''`, '' +
    ''`'' + BUZ_Abschliessen + ''`, '' +
    ''`'' + BUZ_Beitragsbefreiung + ''`, '' +
    angebot.Tarifdaten.BUZ_Tarif + '', '' +
    angebot.Tarifdaten.BUZ_Ablaufalter + '', '' +
    angebot.Tarifdaten.BUZ_Barrente + '')'');
System.out.println(''Tarif'');

AngebotInsert.executeUpdate (''INSERT INTO Ergebnis '' +
    ''VALUES ('' + angebot.Angebotsnummer + '', '' +
    ''`'' + angebot.Ergebnisdaten.Tarif + ''`, '' +
    angebot.Ergebnisdaten.BUZ_Aufschlag + '')'');
System.out.println(''Ergebnis'');

byte i;
```

```
    for ( i = 0; i < angebot.Tarifdaten.Laufzeit; i++)
    {
      AngebotInsert.executeUpdate (
        ''INSERT INTO Ergebnistabelle '' +
            ''VALUES ('' + angebot.Angebotsnummer + '', '' +
            angebot.Ergebnisdaten.Tabelle[0][i] + '', '' +
            angebot.Ergebnisdaten.Tabelle[1][i] + '', '' +
            angebot.Ergebnisdaten.Tabelle[2][i] + '', '' +
            angebot.Ergebnisdaten.Tabelle[3][i] + '', '' +
            angebot.Ergebnisdaten.Tabelle[4][i] + '')'');
    }
    System.out.println(''Ergebnistabelle '');

    AngebotInsert.close();
    con.commit();
    con.close();
    System.out.println (''Verbindung geschlossen.'');
  }
  catch (SQLException ex){
    System.out.println (''***SQLException***'');
    while (ex != null){
      System.out.println (''SQL-Status: '' +
      ex.getSQLState());
      System.out.println (''Meldung: '' +
      ex.getMessage());
      System.out.println (''Hersteller! '' +
      ex.getErrorCode());
      ex = ex.getNextException();
      System.out.println ('''');
    }
  }
  catch (java.lang.Exception ex){
    ex.printStackTrace();
  }
}

//************************************************************
public Kenngroessen getKenngroessen(String d) {

  Kenngroessen kenngroessen = new Kenngroessen();
  java.util.Date aktuelles_datum = new java.util.Date(d);
  try{
    OutputStream outFile = new FileOutputStream (''jdbc.out'');
    PrintStream outStream = new PrintStream (outFile, true);
    DriverManager.setLogStream (outStream);
      }
      catch (java.io.IOException ioex){
        ioex.printStackTrace();}
      try {
```

```
System.out.println (''Lade JDBC-Treiber...'');
Class.forName (''COM.ibm.db2.jdbc.app.DB2Driver'');
System.out.println (''JDBC-Treiber geladen.'');
System.out.println (''Baue Verbindung zur Datenbank auf.'');
Connection con = DriverManager.getConnection(url,user,password);
System.out.println (''Verbindung steht. Sende Anfrage.'');
con.setReadOnly(true);
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
    ''SELECT * FROM Kenngroessen'');
System.out.println (''Anfrage-Ergebnis erhalten.'');
boolean mehr = rs.next();
//Uebergabe des 1. Tupels an Kenngroessen-Objekt
kenngroessen.Gueltig_seit = rs.getString(1);
kenngroessen.Aufloesungsphase[0] = rs.getByte(2);
kenngroessen.Aufloesungsphase[1] = rs.getByte(3);
kenngroessen.Versicherungssumme[0] = rs.getInt(4);
kenngroessen.Versicherungssumme[1] = rs.getInt(5);
kenngroessen.TFS_Anfang[0] = rs.getInt(6);
kenngroessen.TFS_Anfang[1] = rs.getInt(7);
kenngroessen.TFS_Ende[0] = rs.getInt(8);
kenngroessen.TFS_Ende[1] = rs.getInt(9);
kenngroessen.Beitrag[0] = rs.getInt(10);
kenngroessen.Beitrag[1] = rs.getInt(11);
kenngroessen.Zahldauer[0] = rs.getByte(12);
kenngroessen.Zahldauer[1] = rs.getByte(13);
kenngroessen.Dynamik_Wert[0] = rs.getByte(14);
kenngroessen.Dynamik_Wert[1] = rs.getByte(15);
kenngroessen.Dynamik_Rhythmus[0] = rs.getByte(16);
kenngroessen.Dynamik_Rhythmus[1] = rs.getByte(17);
kenngroessen.BUZ_Ablaufalter[0] = rs.getByte(18);
kenngroessen.BUZ_Ablaufalter[1] = rs.getByte(19);

java.util.Date kenngroessen_datum =
    new java.util.Date(kenngroessen.Gueltig_seit);
Kenngroessen dummy = new Kenngroessen();
while (mehr){
  dummy.Gueltig_seit = rs.getString(1);
  dummy.Aufloesungsphase[0] = rs.getByte(2);
  dummy.Aufloesungsphase[1] = rs.getByte(3);
  dummy.Versicherungssumme[0] = rs.getInt(4);
  dummy.Versicherungssumme[1] = rs.getInt(5);
  dummy.TFS_Anfang[0] = rs.getInt(6);
  dummy.TFS_Anfang[1] = rs.getInt(7);
  dummy.TFS_Ende[0] = rs.getInt(8);
  dummy.TFS_Ende[1] = rs.getInt(9);
  dummy.Beitrag[0] = rs.getInt(10);
  dummy.Beitrag[1] = rs.getInt(11);
  dummy.Zahldauer[0] = rs.getByte(12);
```

```
      dummy.Zahldauer[1] = rs.getByte(13);
      dummy.Dynamik_Wert[0] = rs.getByte(14);
      dummy.Dynamik_Wert[1] = rs.getByte(15);
      dummy.Dynamik_Rhythmus[0] = rs.getByte(16);
      dummy.Dynamik_Rhythmus[1] = rs.getByte(17);
      dummy.BUZ_Ablaufalter[0] = rs.getByte(18);
      dummy.BUZ_Ablaufalter[1] = rs.getByte(19);
      java.util.Date tupel_datum =
        new java.util.Date(dummy.Gueltig_seit);
      if ((tupel_datum.after(kenngroessen_datum))
          && (!(tupel_datum.after(aktuelles_datum))))
      {
        kenngroessen.Gueltig_seit = dummy.Gueltig_seit;
        kenngroessen.Aufloesungsphase[0] = dummy.Aufloesungsphase[0];
        kenngroessen.Aufloesungsphase[1] = dummy.Aufloesungsphase[1];
        kenngroessen.Versicherungssumme[0] =
          dummy.Versicherungssumme[0];
        kenngroessen.Versicherungssumme[1] =
          dummy.Versicherungssumme[1];
        kenngroessen.TFS_Anfang[0] = dummy.TFS_Anfang[0];
        kenngroessen.TFS_Anfang[1] = dummy.TFS_Anfang[1];
        kenngroessen.TFS_Ende[0] = dummy.TFS_Ende[0];
        kenngroessen.TFS_Ende[1] = dummy.TFS_Ende[1];
        kenngroessen.Beitrag[0] = dummy.Beitrag[0];
        kenngroessen.Beitrag[1] = dummy.Beitrag[1];
        kenngroessen.Zahldauer[0] = dummy.Zahldauer[0];
        kenngroessen.Zahldauer[1] = dummy.Zahldauer[1];
        kenngroessen.Dynamik_Wert[0] = dummy.Dynamik_Wert[0];
        kenngroessen.Dynamik_Wert[1] = dummy.Dynamik_Wert[1];
        kenngroessen.Dynamik_Rhythmus[0] = dummy.Dynamik_Rhythmus[0];
        kenngroessen.Dynamik_Rhythmus[1] = dummy.Dynamik_Rhythmus[1];
        kenngroessen.BUZ_Ablaufalter[0] = dummy.BUZ_Ablaufalter[0];
        kenngroessen.BUZ_Ablaufalter[1] = dummy.BUZ_Ablaufalter[1];
        java.util.Date kenngroessen_Datum =
            new java.util.Date(kenngroessen.Gueltig_seit);
      }
      mehr = rs.next();
    }
    rs.close();
    stmt.close();
    con.close();
    System.out.println (''Verbindung geschlossen.'');
  }

  catch (SQLException ex){
    System.out.println (''***SQLException***'');
    while (ex != null){
      System.out.println (''SQL-Status: '' +
      ex.getSQLState());
```

```
        System.out.println (''Meldung: '' +
        ex.getMessage());
        System.out.println (''Hersteller! '' +
        ex.getErrorCode());
        ex = ex.getNextException();
        System.out.println ('''');
      }
    }

    catch (java.lang.Exception ex){
      ex.printStackTrace();
    }

    return (kenngroessen);
  }

  //**************************************************************
  public Angebot getAngebot(String d) {

    Angebot angebot = new Angebot();
    angebot.Tarifdaten = new Tarif();
    angebot.Personendaten = new Person();
    angebot.Ergebnisdaten = new Ergebnis();
    angebot.Ergebnisdaten.Tabelle = new int[5][40];
    String ID_Person;

    try{
      OutputStream outFile = new FileOutputStream (''jdbc.out'');
      PrintStream outStream = new PrintStream (outFile, true);
      DriverManager.setLogStream (outStream);
    }
    catch (java.io.IOException ioex){
      ioex.printStackTrace();}

    try {
      System.out.println (''Lade JDBC-Treiber...'');
      Class.forName (''COM.ibm.db2.jdbc.app.DB2Driver'');
      System.out.println (''JDBC-Treiber geladen.'');
      System.out.println (''Baue Verbindung zur Datenbank auf.'');
      Connection con = DriverManager.getConnection(url,user,password);
      System.out.println (''Verbindung steht. Sende Anfrage.'');
      con.setReadOnly(true);

      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(
          ''SELECT * FROM Tarif WHERE Angebotsnummer = '' + d);
      rs.next();
        angebot.Tarifdaten.Preisklasse = rs.getByte(2);
        angebot.Tarifdaten.Laufzeit = rs.getByte(3);
```

```
    angebot.Tarifdaten.Versicherungsbeginn = rs.getString(4);
    angebot.Tarifdaten.Aufloesungsoption = rs.getBoolean(5);
    angebot.Tarifdaten.Dauer_Aufloesungsphase = rs.getByte(6);
    angebot.Tarifdaten.Zahlweise_Beitrag = rs.getByte(7);
    angebot.Tarifdaten.Berechnungsart = rs.getByte(8);
    angebot.Tarifdaten.Vorgabewert = rs.getInt(9);
    angebot.Tarifdaten.Abgekuerzte_Zahlung = rs.getBoolean(10);
    angebot.Tarifdaten.Beitragszahlungsdauer = rs.getByte(11);
    angebot.Tarifdaten.Gewinnsystem = rs.getByte(12);
    angebot.Tarifdaten.Leistungsverlauf = rs.getByte(13);
    angebot.Tarifdaten.TFS_Beginn = rs.getInt(14);
    angebot.Tarifdaten.TFS_Ende = rs.getInt(15);
    angebot.Tarifdaten.TFS_konstant = rs.getByte(16);
    angebot.Tarifdaten.Kennzeichen_Dynamik = rs.getByte(17);
    angebot.Tarifdaten.Rhythmus = rs.getByte(18);
    angebot.Tarifdaten.Erhoehung_um = rs.getByte(19);
    angebot.Tarifdaten.BUZ_Abschliessen = rs.getBoolean(20);
    angebot.Tarifdaten.BUZ_Beitragsbefreiung = rs.getBoolean(21);
    angebot.Tarifdaten.BUZ_Tarif = rs.getByte(22);
    angebot.Tarifdaten.BUZ_Ablaufalter = rs.getByte(23);
    angebot.Tarifdaten.BUZ_Barrente = rs.getInt(24);
    System.out.println (''Tarifdaten erhalten.'');
  rs.close();

  rs = stmt.executeQuery(''SELECT ID_Person FROM Angebot
        WHERE Angebotsnummer = '' + d);
  rs.next();
    ID_Person = rs.getString(1);
  rs.close();
  System.out.println (''Person bestimmt.'');

  rs = stmt.executeQuery(''SELECT * FROM Person
        WHERE ID_Person = '' + ID_Person);
  rs.next();
    angebot.Personendaten.Anrede = rs.getByte(2);
    angebot.Personendaten.Name = rs.getString(3);
    angebot.Personendaten.Vorname = rs.getString(4);
    angebot.Personendaten.Geburtsdatum = rs.getString(5);
    angebot.Personendaten.Strasse = rs.getString(6);
    angebot.Personendaten.Ort = rs.getString(7);
    angebot.Personendaten.Plz = rs.getString(8);
    angebot.Personendaten.Telefon = rs.getString(9);
    angebot.Personendaten.Fax = rs.getString(10);
    angebot.Personendaten.Email = rs.getString(11);
  rs.close();
  System.out.println (''Persondaten erhalten.'');

  rs = stmt.executeQuery(''SELECT BUZ_Aufschlag FROM Ergebnis
        WHERE Angebotsnummer = '' + d);
```

```
      rs.next();
        angebot.Ergebnisdaten.BUZ_Aufschlag = rs.getInt(1);
      rs.close();
      System.out.println (''BUZ_Aufschlag bestimmt.'');

      rs = stmt.executeQuery(''SELECT * FROM Ergebnistabelle
            WHERE Angebotsnummer = '' + d);
      boolean mehr = rs.next();
        byte i;
        for ( i = 0; mehr; i++) {
          angebot.Ergebnisdaten.Tabelle[0][i] = rs.getByte(2);
          angebot.Ergebnisdaten.Tabelle[1][i] = rs.getInt(3);
          angebot.Ergebnisdaten.Tabelle[2][i] = rs.getInt(4);
          angebot.Ergebnisdaten.Tabelle[3][i] = rs.getInt(5);
          angebot.Ergebnisdaten.Tabelle[4][i] = rs.getInt(6);
          mehr = rs.next();
        }
      rs.close();
      System.out.println (''Ergebnistabelle erhalten.'');

      stmt.close();
      System.out.println (''Angebot komplett.'');
      con.close();
      System.out.println (''Verbindung geschlossen.'');
    }

    catch (SQLException ex){
      System.out.println (''***SQLException***'');
      while (ex != null){
        System.out.println (''SQL-Status: '' + ex.getSQLState());
        System.out.println (''Meldung: '' + ex.getMessage());
        System.out.println (''Hersteller! '' + ex.getErrorCode());
        ex = ex.getNextException();
        System.out.println ('''');
      }
    }

    catch (java.lang.Exception ex){
      ex.printStackTrace();
    }

    return (angebot);
  }
}
```

D.2 O_2

```
package DB_Zugriff;

import com.ardentsoftware.jb.api.*;
import java.lang.*;
import java.util.*;

public class Datenbank {

  public void saveAngebot(Angebot angebot) {

  //Verbindung zum Server
  Database database =
      new Database(''o2:santorin:demo'','''','''');
  try {database.connect();
    System.out.println(''Connected!'');}
  catch (DBRuntimeException e){System.out.println(e);};

  //Datenbank öffnen
  try {database.open(''avsg_base'');
    System.out.println(''Base open!'');}
  catch (DBRuntimeException e){System.out.println(e);};

  //da Objekte in die DB geschrieben werden sollen, müssen
  //Operationen in einer Transaktion ausgeführt werden
  Transaction transaction = new Transaction();
  try {transaction.begin();
  //Hat Person schon mal Angebot erstellt?
    Extent personExtent = Extent.all(''Person'').where(
      this.Name = angebot.Personendaten.Name and
      this.Vorname = angebot.Personendaten.and
      this.Geburtsdatum = angebot.Personendaten.Geburtsdatum and
      this.Ort = angebot.Personendaten.Ort);
    Enumeration enum = personExtent.elements();
    if (enum.hasMoreElements())
      //Person schon vorhanden, OID umsetzen
      {angebot.Personendaten = (Person)enum.nextElement();}
      //wenn Person nicht vorhanden, dann Personendaten speichern
      else {database.persist(angebot.Personendaten);}
    //Speichern der Tarif-, Ergebnis- und Angebotsdaten
    database.persist(angebot.Tarifdaten);
    database.persist(angebot.Ergebnisdaten);
    //Array explizit speichern
    database.persist(angebot.Ergebnisdaten.Tabelle);
    database.persist(angebot);
  transaction.commit(); }
  catch (ActiveTransactionExeption e){System.out.println(e);};
  catch (DBRuntimeException e){System.out.println(e);};
```

```
//Datenbank schließen
try {database.close();
  System.out.println(''Base closed!'');}
catch(InvalidCallException e){System.out.println(e);}
catch(TransactionAbortedException e){System.out.println(e);};

//Verbindung zum Server abbauen
try {database.disconnect();
  System.out.println(''Disconnected!'');}
catch(InvalidCallException e){System.out.println(e);}
catch(TransactionAbortedException e){System.out.println(e);};
}



public Kenngroessen getKenngroessen(String d) {
  Kenngroessen kenngroessen = new Kenngroessen();
  //Dummy-Instanz von Kenngroessen,
  //zum Bestimmen der aktuellen Kenngrößen
  Kenngroessen dummy_kg = new Kenngroessen();
  //aus String d Datum machen
  java.util.Date aktuelles_datum = new java.util.Date(d);

  //Verbindung zum Server
  Database database =
      new Database(''o2:santorin:demo'','''','''');
  try {database.connect();
    System.out.println(''Connected!'');}
  catch (DBRuntimeException e){System.out.println(e);};

  //Datenbank öffnen
  try {database.open(''avsg_base'');
    System.out.println(''Base open!'');}
  catch (DBRuntimeException e){System.out.println(e);};

  //wenn keine Änderungen an persistenten Objekten, dann keine
  //Extra-Transaktion nötig, nun effizienterer ReadOnly-Modus
  Extent kgExtent;
  try {kgExtent = Extent.all(''DB_Zugriff.Kenngroessen'');
    //Übergabe des Anfragergebnisses an Enumeration-Objekt
    Enumeration enum = kgExtent.elements();
    //erstes Enumeration-Objekt an Kenngroessen-Instanz übergeben
    kenngroessen = (Kenngroessen)enum.nextElement();
    //Datum der Kenngroessen-Instanz ermitteln
    java.util.Date kenngroessen_datum =
        new java.util.Date(kenngroessen.Gueltig_seit);
    //alle Elemente des Enumeration-Objektes durchgehen und
    //aktuellste Kenngroesse finden
```

```
  while (enum.hasMoreElements()){
    dummy_kg = (Kenngroessen)enum.nextElement();
    java.util.Date dummy_datum =
        new java.util.Date(dummy_kg.Gueltig_seit);
    if ((dummy_datum.after(kenngroessen_datum)) &&
        (!(dummy_datum.after(aktuelles_datum))))
      {kenngroessen.Gueltig_seit = dummy.Gueltig_seit;
      kenngroessen.Aufloesungsphase[0] =
          dummy.Aufloesungsphase[0];
      kenngroessen.Aufloesungsphase[1] =
          dummy.Aufloesungsphase[1];
      kenngroessen.Versicherungssumme[0] =
          dummy.Versicherungssumme[0];
      kenngroessen.Versicherungssumme[1] =
          dummy.Versicherungssumme[1];
      kenngroessen.TFS_Anfang[0] = dummy.TFS_Anfang[0];
      kenngroessen.TFS_Anfang[1] = dummy.TFS_Anfang[1];
      kenngroessen.TFS_Ende[0] = dummy.TFS_Ende[0];
      kenngroessen.TFS_Ende[1] = dummy.TFS_Ende[1];
      kenngroessen.Beitrag[0] = dummy.Beitrag[0];
      kenngroessen.Beitrag[1] = dummy.Beitrag[1];
      kenngroessen.Zahldauer[0] = dummy.Zahldauer[0];
      kenngroessen.Zahldauer[1] = dummy.Zahldauer[1];
      kenngroessen.Dynamik_Wert[0] = dummy.Dynamik_Wert[0];
      kenngroessen.Dynamik_Wert[1] = dummy.Dynamik_Wert[1];
      kenngroessen.Dynamik_Rhythmus[0] =
          dummy.Dynamik_Rhythmus[0];
      kenngroessen.Dynamik_Rhythmus[1] =
          dummy.Dynamik_Rhythmus[1];
      kenngroessen.BUZ_Ablaufalter[0] =
          dummy.BUZ_Ablaufalter[0];
      kenngroessen.BUZ_Ablaufalter[1] =
          dummy.BUZ_Ablaufalter[1];
      java.util.Date kenngroessen_Datum =
          new java.util.Date(kenngroessen.Gueltig_seit);
    }
  }
catch (DBRuntimeException e){System.out.println(e);};

//Datenbank schließen
try {database.close();
  System.out.println(''Base closed!'');}
catch(InvalidCallException e){System.out.println(e);}
catch(TransactionAbortedException e){System.out.println(e);};

//Verbindung zum Server abbauen
try {database.disconnect();
  System.out.println(''Disconnected!'');}
catch(InvalidCallException e){System.out.println(e);}
```

```
    catch(TransactionAbortedException e){System.out.println(e);};

    return (kenngroessen);
  }
}
```

Anhang E

Java E-Mail-Clients

E.1 Die Klasse `jSendEMail`

Version: 1.1 9/18/97
Author: Johnathan Mark Smith
smithj@statenislandonline.com

E.1.1 Methodenübersicht

- {`getMessage()` }
  liefert die Nachricht, die versendet werden soll
- {`getRecipient()` }
  liefert den Namen des Empfängers
- {`getSender()` }
  liefert den Namen des Senders
- {`getSMTPHost()` }
  liefert den SMTP-Host
- {`getSubject()` }
  liefert das Subject der E-Mail
- {`sendEMail()` }
  sendet die E-Mail an den SMTP-Server
- {`setMessage(String)` }
  setzt den Text, der als E-Mail verschickt werden soll
- {`setRecipient(String)` }

setzt den Informationen zum Empfänger für den SMTP-Server

- {`setSender(String)` }
  setzt den Informationen zum Sender für den SMTP-Server
- {`setSMTPHost(String)` }
  setzt den SMTP-Host zum Senden der E-Mail
- {`setSubject(String)` }
  setzt den Text, der als Subjekt der E-Mail verwendet wird

E.1.2 Beispiel für die Verwendung

```
jSendEMail jsm = new jSendEMail(''myhost'');
jsm.setSender(''jsmith@myhost'');
jsm.setRecipient(''joe@foo.com'');
jsm.setSubject(''this is the subject line'');
jsm.setMessage(''This is the message'');
jsm.sendEMail();
```

E.2 E-Mail mit Attachement versenden

Auszug aus `SendMIME.java`

```
/**
Copyright (c)1996 Harm Verbeek, All Rights Reserved.

Description: Use MIME-extensions to send E-mail with attached
binary file from within Java applet;
won't work with applets in WWW-browser
(security violation !!!).

The code is not optimized, it just shows how you
can use MIME-attachments in Java applications.

Version : 0.3
Date : 24-Feb-96

Permission to use, copy, modify, and distribute this software
and its documentation for NON-COMMERCIAL or COMMERCIAL purposes and
without fee is hereby granted.

*/

import java.lang.*;
import java.io.*;
import java.net.Socket;
import java.net.*;
import java.applet.*;
import java.awt.*;

public class SendMIME extends Applet

{
/* name of local host, mailhost, sender & receiver */
TextField my_machine, mailhost, sender, receiver;

  static int statusLines = 0;
  static int width = 600, height = 360;
  static Frame frame;
  static TextField tf_File;
  static TextArea tf_Msg, tf_Log;
  static Button btn1;

  static FileDialog openDialog;
  static int SMTPport = 25;
  static Socket socket;
  static DataInputStream din;
  static PrintStream prout;
```

```
static char BaseTable[] = {
  'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P',
  'Q','R','S','T','U','V','W','X','Y','Z','a','b','c','d','e','f',
  'g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v',
  'w','x','y','z','0','1','2','3','4','5','6','7','8','9','+','/'
  };

/* Name of file to be sent. */
String FileName = '''';

/* All \r\n-combinations are essential ! */
String MsgStart0 = ''\r\n\r\n'';
String MsgStart1 = ''\r\n\r\n--======\r\n\r\n'';
String MsgBody = ''This is where you put your message.'';
String MIME_ver = ''MIME-Version: 1.0\r\n'';
String Content_Typ0 = ''Content-Type: TEXT/PLAIN; CHARSET=us-ascii'';
String Content_Typ1 = ''Content-Type:
    multipart/mixed; boundary='' + '\'' ' + ''======'' + '\'' ' + ''\r\n'';
String Content_Dsp = ''\r\n\r\n--======\r\n'' +
    ''Content-Disposition: inline;filename='';

/* you can change the Content-Type of the attached binary file
to 'image/gif', 'image/jpeg' or whatever is appropriate. */
String Content_Typ2 = ''Content-Type:application/octet-stream\r\n'';
String Content_Enc = ''Content-Transfer-Encoding: base64\r\n\r\n'';
String MsgEnd0 = ''\r\n.\r\n'';
String MsgEnd1 = ''\r\n--=======--\r\n.\r\n'';

/* if everything went OK, mailhost returns code
starting with either ''2'' or ''3''.
The program uses the complete return codes, if they
don't work with your mailhost just use the first
character of each return code (''2'' instead of ''211'', etc.)  */
String SystemStatus = ''211'';
String ClosingCommand = ''221'';
String HelpMessage = ''214'';
String MailSystemReady = ''220'';
String CommandCompleted = ''250'';
String StartMailMessage = ''354'';

/* Open binary file (application, image, data, whatever)
and do base64 encoding */
void Encode_and_Send_File(String filename)
{
  byte buf[] = new byte[4];
  try {
    FileInputStream fin = new FileInputStream(filename);
    File f = new File(filename);
    byte bytes[] = new byte[(int)(f.length())];
```

```
int n = fin.read(bytes);
if (n<1) return;
int n3byt = n / 3;
int k = n3byt * 3;
int nrest = n int linelength = 0;
int i = 0;
while ( i < k ) {
  buf[0] = (byte)(( bytes[i] & 0xFC) >> 2);
  buf[1] = (byte)(((bytes[i] & 0x03) << 4) |
    ((bytes[i+1] & 0xF0) >> 4));
  buf[2] = (byte)(((bytes[i+1] & 0x0F) << 2) |
    ((bytes[i+2] & 0xC0) >> 6));
  buf[3] = (byte)( bytes[i+2] & 0x3F);
  prout.print(BaseTable[buf[0]]);
  prout.print(BaseTable[buf[1]]);
  prout.print(BaseTable[buf[2]]);
  prout.print(BaseTable[buf[3]]);
  if ((linelength += 4) >= 76) {
    prout.print(''\n'');
    linelength = 0;}
  i += 3;
}
int npad = n3byt * 4;
if (nrest==2) {
  buf[0] = (byte)(( bytes[k] & 0xFC) >> 2);
  buf[1] = (byte)(((bytes[k] & 0x03) << 4) |
    ((bytes[k+1] & 0xF0) >> 4));
  buf[2] = (byte)(( bytes[k+1] & 0x0F) << 2);
  prout.print(BaseTable[buf[0]]);
  prout.print(BaseTable[buf[1]]);
  prout.print(BaseTable[buf[2]]);
  npad +=3;
}
else if (nrest==1) {
  buf[0] = (byte)((bytes[k] & 0xFC) >> 2);
  buf[1] = (byte)((bytes[k] & 0x03) << 4);
  prout.print(BaseTable[buf[0]]);
  prout.print(BaseTable[buf[1]]);
  npad +=2;
}
npad if (npad==0) {return;}
npad = 3-npad;
i = 0;

while ( (i++) < npad ) {
  prout.print(''='');
  if ((linelength++) >= 76) {
    prout.print(''\n'');
    linelength = 0;
```

```
        }
      }
      prout.flush();
    }
    catch (NullPointerException e) {
      showStatus(''Client> NullPointerException'');}
    catch (java.io.FileNotFoundException e) {
      showStatus(''Client> FileNotFounfException'');}
    catch (java.io.IOException e) {
      showStatus(''Client> IOException'');}

  } /* Encode_and_Send_File */
  public void showStatus(String s) {
    if (statusLines > 60) {
      tf_Log.setText('''');
      statusLines = 0;
    }
    statusLines++;
    tf_Log.appendText(''\r\n''+s);
  }

  /* read return codes after you've send a command
  to the mailhost */
  void expect(String expected, String msg) throws Exception
  {
    String lastline;
    showStatus(''Client> '' + msg);
    lastline = din.readLine();
    if (!lastline.startsWith(expected))
      throw new Exception(lastline);
      showStatus(''Host > '' + lastline);
      while (lastline.startsWith(expected + ''-'')) {
        lastline = din.readLine();
        showStatus(''Host > '' + lastline);
      }
    }

    /* send mail message with attachment */
    public boolean send_mail() {
      boolean FileOK = false;
      Socket socket = null;
      try {
        // connect to the mail host...
        showStatus(''Client> Connecting to '' +
          mailhost.getText() + ''...'');
        socket = new Socket(mailhost.getText(), SMTPport);
        din = new DataInputStream(socket.getInputStream());
        prout = new PrintStream(socket.getOutputStream());
        expect(MailSystemReady, ''CONNECT'');
```

```
// OK, we're connected, let's be friendly
and say hello to the mail server...
// we use ''EHLO'' instead of ''HELO''
to enable MIME extensions
prout.print(''EHLO '' + my_machine.getText() + ''\r\n'');
// get return message...
expect(CommandCompleted, ''EHLO'');
// For starters, find out the commands that are available...
prout.print(''HELP\r\n'');
expect(HelpMessage, ''HELP'');
// OK, now let server know WHO wants to send mail...
prout.print(''MAIL FROM:<'' + sender.getText() + ''>\r\n'');
expect(CommandCompleted, ''MAIL FROM'');
// let server know WHOM you're gonna send mail to...
prout.print(''RCPT TO:<'' + receiver.getText() + ''>\r\n'');
expect(CommandCompleted, ''RCPT TO'');
// let server know you're now gonna send the message contents...
prout.print(''DATA\r\n'');
expect(StartMailMessage, ''DATA'');
// MIME extensions start here
if ( (new File(FileName)).exists() ) {
  FileOK = true;
}
// MIME Version...
prout.print(MIME_ver);
// To...
prout.print(''To: Receiver <'' + receiver + ''>\r\n'');
// Subject...
prout.print(''Subject: MIME Attachment test\r\n'');
// Send your message...
if (FileOK) {
  // Type of message...
  prout.print(Content_Typ1);
  prout.print(MsgStart1);}
else {
  prout.print(Content_Typ0);
  prout.print(MsgStart0);
}
DataInputStream msg_stream = new DataInputStream(
    new StringBufferInputStream(tf_Msg.getText()));
try {
  while (msg_stream.available() > 0) {
  String ln = msg_stream.readLine();
  if (ln.equals(''.''))
    ln = ''..'';
    prout.println(ln);
  }
  prout.flush();}
catch (IOException e) {
```

```
    showStatus(''Client> Error sending message body.'');
  }
  if (FileOK) {
    // send attachment info...
    prout.print(Content_Dsp);
    prout.print('\'' ' + (new File(FileName)).getName() +
      '\'' ' + ''\r\n'');
    prout.print(Content_Typ2);
    prout.print(Content_Enc);
    // Send attached binary file base64 encoded...
    Encode_and_Send_File(FileName);
    // send the closing part of the message...
    prout.print(MsgEnd1);
  }
  else {
    prout.print(MsgEnd0);
  }
  expect(CommandCompleted, ''END OF MESSAGE'');
  // Say Bye Bye...
  prout.print(''QUIT\r\n'');
  expect(ClosingCommand, ''QUIT'');
}
catch (Exception e) {
  showStatus(''Client> '' + e.getMessage());
  return false;
}

finally {
  try {
  if (socket != null)
    socket.close();
  }
  catch (Exception e)
    showStatus(''Client> '' + e.getMessage());
  }
  return true;
}
```

Selbständigkeitserklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Rostock, den 15.Juli 1998 Thoralf Bojak

Thesen

I. Java ist aufgrund des Sicherheitskonzeptes und der Plattformunabhängigkeit die zur Zeit am besten geeignete Programmiersprache zur Implementierung von Anwendungen für das WWW.

II. Alternativ wäre die Verwendung von CGI-Programmen oder O_2 Web möglich gewesen. Diese Lösungen weisen aber einige Nachteile gegenüber Java auf.

III. Der Versicherungstarif D4 läßt sich relativ einfach mit Java-Klassen beschreiben. Wünschenswert wäre aber die Möglichkeit, Tupelkonstruktoren wiederholt in Java anzuwenden.

IV. Um der Anwendung einen transparenten Datenbankzugriff auf die verwendeten DBMS zu erlauben, kann eine Schnittstelle als Java-Package definiert werden.

V. Die Java-Schnittstelle läßt sich ohne Informationsverlust in das Datenmodell von DB2 überführen. Die Überführung in das Datenmodell von O_2 wird durch ein Werkzeug übernommen.

VI. Das DBMS O_2 besitzt nicht nur starke Vorteile (Datenmodellierung, Datenbankentwurf etc.) gegenüber DB2, es besitzt auch das bessere Java-Binding. Die Implementierungskosten für das Erreichen von Persistenz für Java-Klassen sind für O_2 wesentlich geringer. Daher eignet sich O_2 besser für den Prototypen als DB2.

VII. Die aktive Komponenten von DB2 und O_2 sind in der Definition von Ereignissen und Aktionen nicht mächtig genug für die Realisierung der

Workflow-Komponente. Diese muß als eigene Anwendung implementiert werden.