Projektgruppe 290

FOKIS
Föderiertes objektorientiertes Krankenhausinformationssystem

Teilnehmer:
Andreas Dinsch, Mario Ellebrecht, Xi Gao,
Sven Gerding, Betül Ilikli, Djamel Kheldoun,
Patrick Koehne, Mischa Lohweber, Ulf Radmacher,
K.-H. Schulte, Dilber Yavuz, Peter Ziesche
Betreuer:
Dr. Willi Hasselbring, Klaus Alfert

16. September 1997

Inhaltsverzeichnis

Eine Übersicht

Einleitung

Verfasser: Patrick Koehne

Die PG290 hat sich im September 1996 mit Ihren 12 Teilnehmern zusammengefunden, um etwas ganz neues zu entwickeln: Ein föderiertes objektorientiertes Krankenhausinformationssystem.

Ziel der Projektgruppe ist es, verschiedene Komponenten zu integrieren und somit ein förderiertes System zu schaffen. Die Projektgruppe untersucht, ob dies mit Hilfe von *CORBA*, einem *OODBMS* und einem *RDBMS* möglich ist. *CORBA* ist für die Interprozesskommunikation zuständig und basiert auf einer plattformunabhängigen Standardisierung. Dies ist gerade in Hinsicht auf die Portabilität von Vorteil. O_2 dient als *OODBMS* und *Oracle* als *RDBMS*. Es wurde absichtlich eine objektorientierte und eine relationale Datenbank verwendet, um eine solche Förderierung zu testen. Als Beispielanwendungen sollen ein Abrechnungssystem implementiert werden und ein Eingabesystem aus der Angiographie für das Vorhaben angepaßt werden.

Zu Beginn der Projektgruppe steht eine Einführungsaufgabe. Hier sollen alle benutzten Tools getestet werden und Erfahrungen mit diesen Tools gesammelt werden. Parallel dazu wird bereits das Abrechnungssystem entworfen und implementiert. Die Einführungsaufgabe hat sich als sehr sinnvoll herausgestellt. Die Teilnehmer lernen in einem kleinen Anwendungsbeispiel die verschiedenen Tücken der Software und die Probleme während der Umsetzung kennen. Wenn es dann im späteren Verlauf der Projektgruppe zur Umsetzung der eigentlichen Hauptanwendung kommt, ist man bereits in der Lage Probleme vorzeitig zu erkennen und entsprechend zu behandeln – und es gab genug Probleme!

Der hier vorliegende Endbericht enthält Einführungen in die jeweiligen Thematiken, stellt die Zeitplanung für das Projekt vor und gibt den letztendlich herausgekommenen Zeitverlauf wieder, wird die benutzten Tools in Hinsicht auf diesen Anwendungsfall bewerten, den Entwurf erläutern und den Verlauf des Projekts schildern.

Viel Spaß beim Lesen...

Teil I

Die Seminare

Kapitel 1

Die Nordhelle-Seminare

1.1 Einleitung

Verfasser: Patrick Koehne

Die Seminare, die in diesem Kapitel folgen, sind im Rahmen einer Seminarfahrt gehalten worden. Am 14. Oktober 1996, also zum Beginn des Semesters, trafen sich alle Teilnehmer der PG und fuhren geschlossen nach Nordhelle, im Sauerland. In der dortigen Tagungsstätte wurden an insgesamt drei Tagen von jedem PG-Teilnehmer eines der Seminare gehalten. Die einzelnen Seminare dienten dem Einstieg in die Thematik und die gesamte Fahrt hat sich sehr postiv auf die Gruppendynamik ausgewirkt.

1.2 Datenbank–Schema–Integration

Verfasser: Patrick Koehne

Quelle für diesen Vortrag: [BL86]

1.2.1 Einführung

> Eins der grundlegensten Prinzipien des Datenbankansatzes ist es, daß man alle gespeicherten Daten nichtredundant und einheitlich repräsentieren kann. Dies wird nur möglich, wenn Methoden zur Verfügung stehen, die einen Integrationsprozeß über Organisations– und Anwendungsgrenzen hinweg ermöglichen.

Datenbankmanagementsysteme (DBMS) sind schon seit über 20 Jahren in den verschiedensten Formen in der Entwicklung und Anwendung. Die Datenbankintegration ist allerdings ein relativ neues Problem, welches verstärkt im Zusammenhang mit verteilten Datenbanken auftritt.

Die häufigste Vorgehensweise beim Entwurf von Datenbanken ist es, kleine, unabhängige Schemas zu entwerfen, die Teile der Gesamtanwendung darstellen, und diese anschließend zu einem großen, einheitlichen Schema zusammenzufügen. Die Gründe dafür liegen auf der Hand:

- Die Strukturen von Datenbanken für große Anwendungen werden so groß, daß sie zu komplex sind, als daß sie von einer einzigen Person entworfen werden könnten.
- Verschiedene Benutzergruppen arbeiten typischer Weise unabhängig voneinander und haben somit auch unterschiedliche Anforderungen an die Datenbank.

Der Prozeß der Datenbankintegration ist nun genau der des Zusammenführens von bereits bestehenden Datenbankschemas in ein globales, einheitliches Schema.

1.2.2 Methoden der Schemaintegration

Schwierigkeiten bei der Integration und deren Ursachen

Die grundlegenden Probleme, mit welchen man während der Integration zu tun hat, sind die strukturellen sowie semantischen Unterschiede zwischen den zu mischenden Schemas. Im folgenden werden zunächst die Unterschiede klassifiziert:

Verschiedene Benutzersichten: Während des Entwurfprozesses werden von unterschiedlichen Benutzergruppen unterschiedliche Sichten für dasselbe Objekt in den Entwurf eingebracht. Dies kann nur der triviale Unterschied von verschiedenen Namen für dasselbe Objekt sein, kann aber auch eine unterschiedliche Modellierung von Verbindungen bedeuten.

Äquivalente Entwürfe: Oft kommt es vor, daß dieselben Anwendungen durch unterschiedliche Konstrukte modelliert werden, sprich dasselbe ausdrücken sollen und somit äquivalent sind.

Inkompatibilität: Wenn fälschlicher Weise eine inkorrekte Wahl bezüglich von Namen, Typen oder gar Integritätsbedingungen gemacht wurde, so kann dies zu falschen Eingaben für den Integrationsprozeß führen und zu einem Scheitern der Integration führen. Eine gute Integrationsmethode sollte solche Fehler aufdecken.

Bislang ging es um den gemeinsamen Teil – verschiedene Schemas beinhalten unterschiedliche Konzepte, um dasselbe auszudrücken. Es ist aber auch möglich, daß verschiedene Konzepte in verschiedenen Schemas eine gewisse semantische Verbindung aufweisen. Diese Art von Gemeinsamkeiten werden als *Interschema Properties* bezeichnet.

Man kann die gemeinsamen Konzepte auch etwas formaler fassen: Dasselbe Konzept wird durch unterschiedliche Repräsentationen R_1 und R_2 verwirklicht. Die Unterschiede oder Beziehungen zwischen diesen Repräsentationen können wie folgt klassifiziert werden:

1. identisch: R_1 und R_2 sind genau gleich
2. äquivalent: R_1 und R_2 sind nicht genau gleich, doch wurden äquivalente Konstrukte angewandt. Es gibt die folgenden drei Definitionen von äquivalent:
   (a) vom Verhalten: $R_1 \equiv R_2$, wenn es für jede Instanz von R_1 eine entsprechende Instanz für R_2 gibt, die dieselben Antworten für die Anfrage liefert und umgekehrt.
   (b) abbildbar: $R_1 \equiv R_2$, wenn ihre Instanzen in eine 1:1 - Abbildung gebracht werden können.
   (c) überführbar: $R_1 \equiv R_2$, wenn R_2 aus R_1 durch eine Anzahl von atomaren Transformationen überführt werden kann.
3. kompatibel: R_1 und R_2 sind weder identisch noch equivalent. Trotzdem sind die verwendeten Konstrukte und Abhängigkeiten nicht gegensätzlich.
4. inkompatibel: R_1 und R_2 sind auf Grund ihrer Spezifikation gegensätzlich.

Die Situation 2, 3 und 4 können als Konflikte bei der Integration interpretiert werden.

Bislang wurden also die Ursachen der Schemakonflikte analysiert und diese klassifiziert.

Schritte und Ziele der Integrationsprozesse

Jede Integrationsmethode verfolgt bei der Auflösung der gefundenen Konflikte ihre eigenen Ansätze. Trotzdem sollte sich jede Methode als eine Zusammensetzung aus folgenden Aktivitäten darstellen lassen:

Präintegration: Hier wird eine Analyse der Schemas vor der eigentlichen Integration vorgenommen. Aus ihr sollte hervorgehen, welche Schemas in welcher Reihenfolge integriert werden und welche Belegungen (Attribute, Abhängigkeiten) eventuell noch zu einem Schema hinzugefügt werden müssen, um es zu vervollständigen. Ebenso sollte entschieden werden, wieviele Schemas in einem Schritt integriert werden und welche zusätzlichen Informationen eventuell noch gebraucht werden.

Vergleich der Schemas: Schemas werden miteinander verglichen, um Übereinstimmungen und mögliche Konflikte zwischen ihnen zu finden. Ebenso sollten *Interschema Properties* gefunden werden.

Anpassen der Schemas: Nach Feststellung der Konflikte ist es nun die Aufgabe, diese zu beheben, um das Zusammenfügen der Schemas zu ermöglichen. Eine automatische Konfliktlösung ist meistens nicht möglich. Eine enge Zusammenarbeit zwischen Entwicklern und Benutzern ist hier erforderlich, bevor eventuelle Kompromisse erreicht werden können.

Zusammenfügen und Neustrukturieren: Nun können Schemas zusammengeführt werden, eventuelle Zwischenergebnisse neu analysiert und eventuell neu strukturiert werden, um gewisse Qualitäten zu erzielen.

Ein globales Schema sollte auf folgendes getestet werden:

Vollständigkeit und Korrektheit: Alles, was in den Einzelschemas möglich ist, muß in dem globalen Schema ebenfalls möglich sein.

Minimalität: Wenn ein Konzept mehrmals in den Einzelschemas auftaucht, darf es nur einmal in dem globalen Schema erscheinen: Vermeidung von Redundanzen.

Verständlichkeit: Das integrierte Schema sollte sowohl für den Entwickler als auch für den Benutzer leicht verständlich sein. Dies heißt, daß aus der Vielzahl der Repräsentationsmöglichkeiten, die ein Modell erlaubt, das einfachste ausgewählt wird.

1.2.3 Vergleich der Methoden

Wann sind die Methoden anwendbar?

In der Literatur werden 12 verschiedene Methoden vorgestellt, deren namentliche Nennung nur im Einzelfall sinnvoll ist. Die meisten Methoden fallen in die Kategorie der Sichtintegration, die an verschiedenen Stellen des Datenbankentwurfs angewendet werden kann.
Die offensichtlich bevorzugteste Phase für die Anwendung ist während des Konzeptentwurfes, da es dort die wenigsten Probleme gibt. Zwischen der Anforderungsanalyse und dem Konzeptentwurf ist die Benutzeranforderung meist noch zu schlecht strukturiert, um sinnvoll damit arbeiten zu können. Während des Implementationsentwurfes ist der Gebrauch der Abstraktion nicht mehr möglich.
Allgemein sollte man sich an die folgenden zwei Regeln halten:

1. Integration sollte so früh wie möglich angewendet werden, weil sonst die Gefahr ansteigt, falsche oder inkonsistente Daten durch den "Life – Cycle" des Datenbankentwurfes mitzuschleifen.
2. Integration sollte nur dann angewendet werden, wenn vollständige, korrekte, minimale und eindeutige Entwürfe vorliegen.

Der Integrationsprozeß als Blackbox

Einfach gesagt, kann man die Eingaben als eine Anzahl von Schemakomponenten und die Ausgabe als ein integriertes Schema betrachten. Eine etwas feinere Aufteilung ist bei der näheren Betrachtung der vorgestellten Methoden allerdings schon zu erkennen. Neben den eigentlichen Schemas wird bei den Eingaben unterschieden in:

Enterprise View: Diese Eingabe ist nur bei der Sichtintegration möglich. Es handelt sich um ein Anfangskonzept aus der Firmensicht auf die wichtigsten Konzepte. Diese Sicht kann die Arbeit bei der Vereinfachung erleichtern.

Assertions: *Intraview–Assertions* sind Regeln innerhalb eines Konzeptes, *Interview–Assertions* Regeln zwischen verschiedenen Konzepten.

Processing Requirements: Diese Anforderungen beziehen sich auf Operationen, die auf einzelne Komponentensichten definiert sind. Einige Methoden ignorieren diese Anforderungen gänzlich, andere beziehen sich nur auf die Transaktionen und Anfragen nach Abschluß der Integration, andere konzentrieren sich im Detail nur auf die Umsetzung von lokalen Anfragen auf globale Anfragen. Eine komplette Erfassung und Bearbeitung der "Processing Requirements" findet allerdings bei keiner Methode statt.
Die Art und Weise, in der Ein– und Ausgaben definiert werden, ist von keinem der Entwickler genauer angegeben worden. Man kann nur darauf schließen, daß es sich um eine wohl definierte Sprache oder um intern verwendete Datenstrukturen handeln muß, wenn die Methoden zu einer gewissen Automatisierung führen sollen.

Die Ausgaben lassen sich, neben dem globalen Schema, noch in eine Liste von ungelösten Konflikten, sowie in die Abbildungsregeln unterteilen.

Die Architektur der Methoden

In Kapitel 2.2 wurde bereits erwähnt, daß der Integrationsprozeß in die fünf Schritte Präintegration, Vergleichen, Anpassen, Zusammenfügen und Neustrukturieren aufgeteilt werden kann. Die verschiedenen Methoden können somit anhand ihrer Vorgehensweise und Schleifendurchläufe wie folgt klassifiziert werden:

1. Ständige Wiederholung von Vergleich, Anpassung und Mischen und somit Vermeidung der Neustrukturierung am Ende.
2. Die meisten Aktivitäten geschehen während und nach dem Mischen. Nur Anpassen und Mischen werden verwendet und ein Vergleich somit umgangen.
3. Alle fünf Aktivitäten werden benutzt.
4. Präintegration wird explizit erwähnt.

Bei näherer Betrachtung der Schleifendurchläufe lassen sich die folgenden Ähnlichkeiten feststellen:

1. "No–feedback"– Ansatz. Nur Mischen und Neustrukturierung werden verwendet.
2. Wie 1. nur mit "Feedback".
3. Eine große Schleife vom Ende zum Vergleichsschritt ist möglich.
4. Wiederholung von Vergleich und Anpassung vor dem eigentlichen Mischen ist möglich.

Die einzelnen Schritte

Präintegration

Auch wenn nur bei drei Methoden Präintegration erwähnt wird, so ist es für alle Methoden notwendig die Integrationsmethode festzulegen (Reihenfolge und Gruppierung). Hier wird in "binary" und "n–ary" unterschieden. Bei der "binary"–Methode werden jeweils nur zwei Schemas ineinander gemischt, während bei der "n–ary"–Methode mehr als zwei Schemas gemischt werden können.
Die Vorteile der "binary"–Methode liegen auf der Hand:

- Vereinfachung von Vergleich und Anpassung bei jedem Schritt der Integration,
- die meisten Methoden stimmen darin überein, daß dies wegen der zunehmenden Komplexität jedes einzelnen Integrationsschrittes bei steigender Anzahl von Einzelschemas die sinnvollste Methode ist,
- im allgemeinen kann gezeigt werden, daß Mischalgorithmen für n Schemas eine Komplexität von n^2 haben. Somit ist ein kleines n erstrebenswert.

Der Nachteil dieser Methode ist allerdings die Zunahme von Integrationsprozessen, die eine abschließende Analyse und eventuelles Einfügen von globalen Eigenschaften erzwingt.

Wenn zwei Schemas integriert werden und das sich ergebene Schema wiederum mit einem neuen Schema integriert wird, so nennt man diese Vorgehensweise "ladder"–Strategie. Der Vorteil dieser Strategie liegt darin die Einzelschemas gewichten zu können und die Zwischenergebnisse ebenfalls nach Wichtigkeit sortiert zu erhalten. Die "Enterprise–View" könnte somit z.B. als ein Startschema gewählt werden und wäre damit richtungsweisend.

Bei der “balanced”–Methode werden die Schemas wie in einem Binärbaum zusammengemischt. Diese Methode macht sich somit zum Vorteil, daß sie die Integrationsoperationen wieder minimalisiert.

Die beiden verwendeten “n–ary one–shot”-Varianten analysieren in Schritt 2 alle Schemas auf einmal. Nach Sammeln aller Daten wird dann einmal integriert. Eine ziemlich große Zahl von Analysen kann somit vor dem Mischen geschehen, was die Notwendigkeit von späteren Analysen und Korrekturen herabsetzt.

Die angewendete “iterative n–ary”–Methode gruppiert zunächst ähnliche Benutzersichten und mischt diese zusammen. Die Zwischenergebnisse werden wieder analysiert, gruppiert und gemischt.

Nicht bei allen Methoden wurde angegeben, welche Strategie sie verfolgen oder wie, und ob sie “ausbalancieren”.

Vergleichen der Schemas

Hier müssen nun die Namens– sowie die Strukturkonflikte aufgedeckt werden. Bei den Namenskonflikten unterscheidet man in “Homonyme”, bei denen gleiche Worte für verschiedene Konzepte verwendet wurden, und “Synonyme”, bei denen das gleiche Konzept verschieden benannt wurde. Wo immer “Homonyme” durch vergleichen erkannt werden können, können “Synonyme” nur noch extern aufgespürt werden. Wörterbuchdatenbanken haben sich als ein nützliches Tool herausgestellt. Bei den meisten Methoden wurde scheinbar vorausgesetzt, daß im Vorfeld solche Konflikte bereits beseitigt wurden, denn sie gehen gar nicht darauf ein.

Die Strukturkonflikte lassen sich wie folgt gliedern:

Typkonflikte: Ein Entity in einem Schema ist Attribut in anderem Schema.

Abhängigkeitskonflikte: 1:1 – Verbindungen in dem einen Schema hat eine n:m – Verbindung in dem anderen Schema.

Schlüsselkonflikte: Beispiel: In einem Schema ist der Name Schlüssel, in dem anderen die Kundennummer.

Verhaltenskonflikte: Unterschiedliche Integritätsbedingungen liegen vor.

Anpassen der Schemas

Hier werden die zuvor gefundenen Fehler behoben, um die Schemas kompatibel für den Mischvorgang zu machen. Namenskonflikte können z.B. durch simple Umbenennung umgangen werden. Meist wurde in den Methoden gar nicht erwähnt wie die Probleme gelöst werden, sondern nur gesagt, daß dies an dieser Stelle erledigt werden muß – im Bestfalle wurden Hinweise gegeben. Ebenfalls wurde bei keiner der Methoden gezeigt oder bewiesen, daß alle Konflikte gefunden wurden und diese auch alle lösbar sind.

Mischen und Neustrukturieren

Es werden nun also die verschiedenstes Operationen auf die Einzelschemas und Zwischenschemas angewendet. Jede Methode definiert dazu ihre eigenen Funktionen, die Manipulationen an den Schemas vornehmen können.

Nach Abschluß der Integration sollte auf die folgenden Punkte hin kontrolliert werden:

Vollständigkeit: Dadurch, daß der eigentliche Mischvorgang ein reines Zusammenführen der vorhandenen Schemas ist, wird die Vollständigkeit impliziert.

Minimalität: [Batini und Lenzerini] erwähnen hierbei, daß das Aufdecken von Redundanzen eine Aufgabe des Konzeptentwurfes ist und deren Beseitigung somit während der Implementationsphase geschehen muß. Die meisten Methoden beschäftigen sich mit diesem Punkt während der Neustrukturierung – wie ist allerdings nicht näher beschrieben.

Verständlichkeit: Dieser Punkt wurde nur bei [Batini und Lenzerini] explizit erwähnt. Meist wurde diese Notwendigkeit bei den anderen Methoden diffus verstreut und nicht auf den Punkt gebracht. Dies mag sicherlich daran liegen, daß es bislang keine "Kodiermethode" gibt, die "Übersicht" irgendwie erkennen oder vereinfachen kann.

1.2.4 Zusammenfassung

Die meisten Methoden wurden in Rahmen von Projekten entwickelt, die nicht das Ziel hatten vollautomatische Systeme zu ergeben. Es ist jedoch erkennbar, daß man Entwicklungstools entwerfen kann, wenn man Konzepte aus den verschiedenen Methoden verwendet. Es gibt Berichte darüber, daß Teilimplementationen von einzelnen Methoden gemacht wurden, doch es gibt noch keinerlei Hinweise, daß wirklich eine Methode zur kompletten Sichtintegration verwendet wird.
Das ER–Modell hat sich als das am meisten verbreiteste Modell in der Praxis herausgestellt und sich als sinnvoll erwiesen.

Zum Thema **Vollständigkeit und genaue Spezifikation** bleibt anzumerken:

- fast keins der Modelle stellt Algorithmen für die Integration bereit,
- nur ganz selten wird darauf eingegangen, ob Konflikte vollständig erfaßt und/oder behoben werden können,
- bei den auftauchenden Schleifenkonstrukten ist deren Terminierung dem Wohlwollen des Benutzers überlassen.

Die folgenden Aspekte wurden ganz ausgelassen:

Ablaufspezifikation: Am Anfang wurde erwähnt, daß Integrationsmethoden eigentlich zwei Ziele bezüglich der Anfragen und Transaktionen der Einzelschemas haben sollte:

Möglichkeiten: Das integrierte Schema sollte all das erlauben, was die Einzelschemas auch ermöglichen.

Performance: Ist das integrierte Schema "optimal" in Bezug auf Anfragen der Einzelschemas? Auf dem Level des Konzeptentwurfes ist diese Frage sicherlich noch nicht sinnvoll, da ein entsprechender "maßgebender Indikator" fehlt. Man sollte diese Frage bei einer späteren vollständigen Definition in einem DBMS allerdings nicht außer Acht lassen.

Verhaltensspezifikation: Keins der Modelle setzt sich vollständig mit diesem Thema auseinander. Ein Beispiel hierfür wäre: Gehälter eines Angestellten können nicht sinken. Was passiert mit solchen Definitionen?

Schemaabbildungen: Wird von allen erwähnt, alle haben meist als Output sogenannte "mapping–rules", jedoch nur eine Methode stellt wirklich ein Regelwerk dafür bereit.

Über verteilte Datenbanken wurde in diesem Zusammenhang noch gar nicht berichtet. Ebenso wurde über den objektorientierten Ansatz bei Datenbankmanagementsystemen nichts berichtet. Dies liegt allerindgs hauptsächlich an der Tatsache, daß die Literatur von 1986 ist.

1.3 CORBA - Ein Standard für verteilte Objekte

Verfasser: Peter Ziesche

1.3.1 Einleitung

Die *Object Management Group (OMG)* ist ein internationales Firmenkonsortium, daß 1989 von acht Firmen ins Leben gerufen wurde. Inzwischen umfaßt sie mehr als 500 Mitglieder. Ihre Aufgabe besteht darin, einen Standard zu schaffen, der

- die Kommunikation und Zusammenarbeit von Objekten in einer verteilten Umgebung regelt,
- den Aufbau von Anwendungen aus verteilten Objekten ermöglicht und
- dabei auf bereits existierenden Technologien aufbaut, so daß eine Implementierung dieses Standards direkt möglich ist.

Das Ergebnis der bisherigen Arbeit ist eine Referenzarchitektur, in der die aufgeführten Anforderungen berücksichtigt sind. Sie trägt den Namen *Object Management Architecture (OMA)*. Es handelt sich dabei nur um eine Schnittstellen- und Protokollspezifikation. Die OMG bietet also keine Standardimplementierung an.

Inzwischen existieren einige Implementierungen der OMA-Architektur, z.B. von IBM (DSOM), Digital (Object Broker) oder CHORUS (COOL).

Dieser Artikel basiert im wesentlichen auf dem State-of-the-Art-Bericht über OMG/CORBA [YD95] und geht nicht auf CORBA-Implementierungen ein.

1.3.2 Die OMA Architektur

Ein OMA System besteht ausschließlich aus miteinander kommunizierenden Objekten. Die gesamte Funktionalität eines Systems muß also in Objekte verkapselt werden. Die einzelnen Objekte tauschen Informationen nicht direkt miteinander aus, sondern benutzen einen gemeinsamen Kommunikationsbus, den ORB (*Object Request Broker*). Das Verfahren ist in etwa mit der Kommunikation verschiedener Hardware-Komponenten in einem PC oder einer Workstation vergleichbar.

Dieser Kommunikationsbus bildet den Kern von OMA. Der Aufbau eines ORBs ist in der CORBA-Spezifikation (*Common Object Request Broker Architecture*) beschrieben. Ein Objekt (*Client*) sendet Anfragen an den ORB. Dieser leitet die Anfrage an das Zielobjekt (*Server*) weiter, so daß Client und Server nie direkt miteinander in Verbindung stehen.

Die Objekte, die den ORB zur Kommunikation nutzen, teilen sich in drei Kategorien:

- *Objektdienste* sind in Objekten gekapselte low-level Funktionalitäten, die unabhängig von einem bestimmten Problembereich häufig benötigt werden. Durch ihre genaue Spezifikation hat die OMG quasi eine Schnittstelle zwischen dem Implementierer einer OMA-Architektur und dem Anwendungsprogrammierer geschaffen.
- *Anwendungsdienste* haben ein höheres Abstraktionsniveau. Sie bauen auf den Objektdiensten auf und stellen dem Anwendungsprogrammierer z.B. Funktionen für User-Interfaces oder Information-Management zur Verfügung. Im Gegensatz zu den Objektdiensten sind sie bisher nicht von der OMG spezifiziert. Da es sich teilweise um problemspezifische Funktionen handeln dürfte, ist eine solche Spezifikation auch nicht unbedingt sinnvoll.
- *Anwendungsobjekte* sind das OMA Gegenstück zu Applikationen in einem konventionellen System. In Ihnen werden also problemspezifische Lösungen verkapselt. Eine komplette Applikation ist meist aus mehreren, miteinander in Verbindung stehenden Anwendungsobjekten aufgebaut.

1.3.3 Objekte

CORBA-Objekte bestehen aus einer Schnittstelle und einer Objektreferenz. Ein Objekt (Client) richtet über den ORB Anfragen an ein anderes Objekt (Server). Der Server kann natürlich wieder Anfragen stellen, so daß ein CORBA-Objekt sowohl Client als auch Server sein kann.

Der Objektbegriff in CORBA

Der Objektbegriff in CORBA unterscheidet sich wesentlich von den Definitionen für Objekte, wie man sie z.B. aus objektorientierten Programmiersprachen kennt. Die wesentlichen Merkmale werden im folgenden kurz erklärt:

Schnittstellen: Da es sich bei der gesamten OMA nur um die Spezifikation eines Standards handelt, wird auch auf Objektebene nichts über die Implementierung ausgesagt. Lediglich die Spezifikation der Schnittstelle für ein Objekt wird festgelegt. Die Art und Weise, wie eine Schnittstellenspezifikation implementiert wird, unterscheidet sich also bei verschiedenen OMA-Implementierungen.

Referenzen: Referenzen werden in der CORBA-Spezifikation nur abstrakt beschrieben. Die Festlegung des genauen Aufbaus bleibt einer OMA-Implementierung vorbehalten. Eine Vergleichsoperation auf Referenzen ist ebenfalls nicht vorgesehen. Referenzen müssen nicht eindeutig sein, ein ORB muß lediglich mit Hilfe einer Referenz immer das selbe Objekt identifizieren können. Wenn Objekte verschiedener ORBs miteinander kommunizieren, sind die ORBs für die gegenseitige Übersetzung der Referenzen verantwortlich.

Klassen: CORBA-Objekte sind nicht als Instanzen von Klassen zu verstehen, d.h. das gemeinsame Verhalten mehrerer Objekte kann nicht in einer Klassendefinition beschrieben werden.

Vererbung: Durch das Fehlen von Klassen kann der Vererbungsbegriff, wie wir ihn aus OO-Sprachen kennen, nicht übernommen werden. Ein CORBA-Objekt kann lediglich die Schnittstelle eines anderen Objekts erben. Redefinitionen sind dabei nicht erlaubt. Code-Reuse ist auf dieser Ebene nicht realisierbar. Ebenso kann es keine Polymorphie geben.

Granularität: CORBA-Objekte sind in der Regel grob granularer als in OO-Programmiersprachen. Hinter einem solchen Objekt kann sich eine komplette Applikation verbergen. Bei der Implementierung mit einer OO-Sprache kann ein CORBA-Objekt vielleicht aus vielen Objekten bestehen.

Migration: Durch die Beschränkung auf die Schnittstelle ist die Migration nicht obejektorientierter Anwendungen möglich. Die Funktionalität einer Applikation kann durch ein CORBA-Objekt mit entsprechender Schnittstelle verkapselt werden (*Object Wrapper*). Anfragen an dieses Objekt werden an die eigentliche Applikation delegiert.

Schnittstellen

Wie erwähnt sind Objekte auf Ebene der CORBA-Spezifikation abstrakte Gebilde mit einer Referenz und einer Schnittstelle. Zur Spezifikation einer Schnittstelle hat die OMG IDL (*Interface Definition Language*) entwickelt. Es handelt sich dabei um eine an C++ angelehnte Sprache, wobei die auf Implementierung ausgerichteten Konstrukte entfernt wurden. In dieser Sprache wird die Schnittstelle spezifiziert.

Konkrete OMA-Implementierungen können dem Entwickler anschließend einn Abbildungsmechanismus auf eine Programmiersprache zur Verfügung stellen. Dadurch wird die automatische Generierung von Coderahmen und Importbibliotheken für die Kommunikation mit dem ORB ermöglicht.

Anfragen

Ein Client richtet Anfragen an einen Server, indem er dem ORB die Objektreferenz des Servers und die aufzurufende Methode angibt. Zu diesem Zweck muß er aber die Schnittstelle des Servers kennen. CORBA bietet im wesentlichen zwei Arten von Aufrufen an:

- statische Anfragen und
- dynamische Anfragen

Bei statischen Anfragen ist die Schnittstelle des Servers zur Übersetzungszeit des Clients bekannt, d.h. es kann auf die IDL-Spezifikation zugegriffen werden. Eine OMA-Implementierung kann, ähnlich wie auf der Server-Seite, die IDL auf eine Programmiersprache abbilden und Importbibliotheken erzeugen. Für den Programmierer des Client stellt sich eine Anfrage dann als ein normaler Funktionsaufruf dar. In den meisten Fällen dürfte diese Art der Anfrage ausreichen.

Es besteht jedoch auch die Möglichkeit, daß der Client die Schnittstelle eines Servers zur Laufzeit ermitteln muß. Zu diesem Zweck stell der ORB ein DII (*Dynamic Invocation Interface*) zur Verfügung. Der Client richtet eine Anfrage an den ORB, in der er das Zielobjekt spezifiziert und Methoden und Parameter der geplanten Anfragen mitteilt. Der ORB versucht daraufhin, ein Zielobjekt zu bestimmen und generiert bei erfolgreicher Suche eine entsprechende Anfrage an den Server. Für den Server ist nicht ersichtlich, auf welche Weise eine Anfrage an ihn gerichtet wird.

In der CORBA-Spezifikation sind sowohl synchrone als auch asynchrone Anfragen definiert. Auf einer tieferen Abstraktionsebene arbeitet CORBA mit RPC (*Remote Procedure Call*), einem sychronen Verfahren. Eine asynchrone Anfrage muß daher emuliert werden. Die Problematik asynchroner Kommunikation ist aber nicht CORBA-spezifisch und soll deshalb hier nicht näher erläutert werden.

Objektadapter

Der Objektadapter dient als Zwischenschicht, einer Art API, zwischen dem ORB und den Implementierungen der Objekte. Er ermöglicht es dem Programmierer, einige Funktionen direkt zu nutzen und nicht alles über Afragen erledigen zu müssen. Die OMG hat einen BAO (*Basic Object Adapter*) spezifiziert, den alle OMA-Implementierungen bereitstellen sollen.

Dieses Prinzip der Zwischenschicht paßt nicht so ganz in das ansonsten sehr klare Konzept von OMA (es gibt nur Objekte und den ORB). Der BOA existiert jedoch praktisch nur während der Entwicklung.

1.3.4 Objektdienste

Da in der OMA-Spezifikation Objektdienste aufgezählt und genau beschrieben werden, sollen einige dieser Objekte kurz vorgestellt werden:

Event Notification Service: Dieser Dienst leitet bestimmte Ereignisse (ausgelöst von einem Objekt) an interessierte Objekte weiter. Dadurch wird die Kopplung zwischen den einzelnen Objekten lockerer, was ihre unabhängige Wiederverwendung erleichtert.

Lifecycle Service: Dieser Dienst ermöglicht es, Objekte zu erzeugen, zu löschen, zu kopieren oder zu verschieben. Clients, die ein Objekt erzeugen wollen, tun dies also nicht direkt, sondern benutzen diesen Dienst als Fabrikobjekt.

Persistent Service: Auf die Implementierung, insbesondere also auf den internen Zustand eines Objektes kann nur über die Schnittstelle zugegriffen werden. Trotzdem kommt es häufig vor, daß ein Objekt seinen internen Zustand dauerhaft speichern muß. Dieser Dienst bietet dazu eine Reihe von Erleichterungen an.

Security Service: In einigen Anwendungsbereichen werden hohe Anforderungen an die Sicherheit eines Systems gestellt. Der Security-Dienst bietet eine Reihe von Möglichkeiten, um z.B. nur authorisierten Anwendern den Zugriff auf ein Objekt zu ermöglichen oder Zugriffe auf ein Objekt aufzuzeichnen.

Collection Object Service: Dieser Dienst ermöglicht es, Sammlungen von Objekten (z.B. Stacks, Listen, Mengen) gleichförmig zu verwalten. Diese Funktionalität wird praktisch in jeder Anwendung benötigt.

1.4 Ein Kontrollsystem für die Angiographie

Verfasser: Ulf Radmacher

1.4.1 Einleitung

Thema dieser Ausarbeitung ist das *Therapiekontrollsystem für die Angiographie*, welches im Rahmen einer Diplomarbeit von Ingo Ullrich [Ull96] in Zusammenarbeit mit den Medizinern der Radiologischen Abteilung der Klinik *Wuppertal-Barmen*, entwickelt worden ist. Es handelt sich hierbei um einen in O_2implementierten Prototypen, der nicht, wie es ursprünglich geplant war, in *Wuppertal-Barmen* zum Einsatz gekommen ist. Die Applikation ist nach Meinung von Ingo Ullrich in der Lage, alle in der Praxis der Angiographie anfallenden Daten zu verarbeiten. Um die Daten auszuwerten ist das Statistik-Tool *SAS* verwendet worden, für welches der Prototyp die benötigten Informationen in Form einer Datei zur Verfügung stellt. Es besteht die Möglichkeit eine Patientenakte auszudrucken. Ziel der Diplomarbeit ist es gewesen, festzustellen, ob Behandlungsmethoden durch Auswertung ihrer Therapieerfolge verbessert werden können. Des weiteren sollte es ermöglicht werden, Behandlungsmethoden besser dokumentieren zu können.

1.4.2 Der Patientenfluß in der Angiographie

Wird ein Patient in der Angiographie stationär aufgenommen, so durchläuft er mehrere Prozesse, bei denen Untersuchungsdaten und Therapiedaten anfallen.

Zuerst werden in der **Anmeldung** die Stammdaten des Patienten aufgenommen (*Name, Vorname, Geburtsdatum, Straße, Postleitzahl, Ort, Telefon dienstlich, Telefon privat*).

Bei der **Anamnese** fallen Daten an, die nur schwer zu erfassen sind, da sie aus einem sehr großen Wissensbestand stammen können. Sie werden in der Applikation nicht erfaßt, weil sie für zu unscharf befunden worden sind.

Die nächste Station ist die **Aufnahme**. In der Aufnahme werden Informationen bzgl. Überweiser, Hausarzt, Untersucher und Radiologe des Patienten aufgenommen. Hier findet eine erste Dopplermessung statt, die den Blutdruck und die Indizes bestimmt. Des weiteren wird unter dem Oberbegriff Diagnose der Zustand des Patienten dokumentiert. Hier wird zum Beispiel festgehalten, in welchem Zustand sich die **Arterio-Venöse-Krankheit** des Patienten befindet, eine subjektive Einschätzung des Leidensdruckes und der Gehstrecke durch den Patienten, die Familienanamnese und wieviel der Patient am Tag raucht. Zusätzlich soll hier eine Mehrfachauswahl mit *Check-Buttons* und Datumsangabe vorhanden sein, die Informationen bzgl. des Gesundheitszustandes des Patienten beinhaltet, wie z.B. über Allergien, Diabetes, Hypertonie, Magengeschwüre, Infarkte etc. Alle Notizen, die in der Aufnahme in einer Zeichnung festgehalten worden sind, sollen nun in ein Piktogramm eingetragen werden können. Weitere bei der Aufnahme durchgeführte Untersuchungen werden klarschriftlich mit ihren Befunden erfaßt und es wird unter dem Begiff *Procedere* die weitere Vorgehensweise bei der Behandlung des Patienten festgehalten. Der untersuchende Arzt hat abschließend die Möglichkeit den Patienten als *normal* oder *vorgezogen* in die Warteliste einzutragen.

Nach der Aufnahme folgt die **stationäre Voruntersuchung**. Hier werden nach Bedarf vier verschiedene Untersuchungen durchgeführt. Alle Untersuchungen sind mit einem Datum zu versehen, um eine Historie des Krankheitsverlaufes erstellen zu können. Die Untersuchungen sind: Blutdruck- und Dopplermessungen, Farbduplexsonographie, Normierte Gehstrecke und MR-Angiographie.

Bei den **Blutdruck- und Dopplermessungen** werden am rechten und linken Arm sowohl der diastolische als auch der systolische Blutdruck gemessen, sowie die Dopplerwerte am rechten und linken Fuß, bzw. Unterschenkel.

Bei einer **Farbduplexsonographie** erhält man Informationen über die Flußgeschwindigkeit des Blutes an einer Stelle im Gefäßsystem und kann Aussagen über eine Läsion treffen (z.B. Plaque, Stenose, Aneurysma, Thrombose).

Bei der **normierten Gehstrecke** wird der Patient auf einem Laufband solange getestet, bis er die Schmerzen als zu stark empfindet. Hier müssen Informationen über Geschwindigkeit des Laufbandes, die Steigung, die gelaufene Strecke bis Schmerzbeginn bzw. bis zum Abbruch und die Seite, auf der die Schmerzen eingesetzt haben, dokumentiert werden. Die normierte Gehstrecke dient als Index für den Therapieerfolg. Sie wird bei der ambulanten Kontrolluntersuchung erneut bestimmt.

Die **MR(Magnetresonanz)-Angiographie** liefert den Querschnitt eines Gefäßabschnittes eines Patienten. Dieser wird dann bzgl. Stenose, Verschluß, Plaque und Gefäßwandverkalkung beurteilt.

Auf die Voruntersuchung folgt die **Therapie**. Hier werden das Datum, die Untersucher, die Assistenten, verwendetes Material, Medikamente und Kontrastmittel, sowie die Art der Punktion, Komplikationen, Untersuchungen und die Diagnose gespeichert. Dokumentiert werden außerdem die Befunde (Angioplastische Befunde, Embolisationen, Fremdkörperextraktionen, Urogenital-System, Venöse und portale Abflußstörungen) und daraus resultierende Maßnahmen.

Die bei der **stationären und ambulanten Nachuntersuchung** anfallenden Daten entsprechen denen der Voruntersuchung. Wird ein Patient entlassen, so wird dies nicht erfasst.

1.4.3 Anforderungen an die Funktionalität der Applikation

Die Anforderungen müssen in Zusammenhang mit den zu erfassenden Daten stehen und sind im wesentlichen die Dateneingabe, die Datenspeicherung, die Abfrage, die Auswertung, sowie der Ausdruck der Daten. Eine Korrektheitsprüfung von eingegebenen Daten sollte so früh wie möglich gemacht werden, bevor Fehler persistent gemacht werden. Die Patientendaten werden in einer Liste gespeichert, in der bei der Aufnahme nach bereits vorhandenen Daten gesucht werden muß. Es ist nicht möglich, die Daten eines Patienten ganz oder teilweise zu löschen, damit nicht versehentlich Teile der Krankheitsgeschichte verloren gehen. So fehlen auch für alle Basislisten (Medikamente, Ärzte, Drähte, ...) die Löschfunktionen, da Elemente dieser Listen eventuell schon bei der Dokumentation verwendet worden sind. Die Ergebnisse der normierten Gehstrecke sind als Liste implementiert, die nach dem Datum sortiert ist. Bei den Blutdruckwerten ist der diastolische Wert als Zwangsfeld implementiert, der auszufüllen ist, wenn bereits ein systolischer Wert angegeben worden ist. Dies soll sicherstellen, daß ein Abfall des Blutdruckes zwischen dem rechten und linken Arm anhand des diastolischen Wertes diagnostiziert werden kann. In ein Piktogramm (lag mir leider nicht vor) können die Befunde der Farbduplexsonographie graphisch eingetragen und differenziert betrachtet werden. Die Auswahl der Behandlungen die im Procedere ausgewählt werden können, erfolgt über *Check-Buttons*, wobei auch eine Mehrfachauswahl möglich ist. Mit Hilfe des *SAS*-Tools ist es möglich, Aussagen bzgl. der durchschnittlichen Bestrahlungsdauer, der Qualität der Therapie bezogen auf eine bestimmte Patientengruppen und der Eignung eines Medikamentes bezogen auf eine Erkrankung treffen zu können.

1.4.4 Entwurf und Implementierung

Ingo Ullrich ist beim Entwurf der Applikation der objektorientierten Methode nach *Booch* gefolgt. Leider hat er sich nach meiner Meinung bei der Ausarbeitung der Klassendiagramme und des Zustandsübergangsdiagrammes nicht besonders viel Mühe gemacht, was sich bei dem Versuch, den Entwurf zu verstehen, bemerkbar gemacht hat. Es existiert beispielweise keine Erläuterung der dargestellten Klassen, auch ist die Notation teilweise falsch angewendet worden. Da die graphische Oberfläche mit O2Look generiert worden ist, ist der Entwurf nicht vollständig übernommen worden. Die letztendlich implementierten Klassen lassen sich in drei Kategorien aufteilen:

In den **Basisklassen** werden grundlegende Dokumentationsinformationen gespeichert. Zu den Basisklassen gehören: *CPerson, CKompplikation, CMaterial, CArzt, CArterien, CKatheter, CVerbrauch, CMedikament, CMaßnahme, CBefund, CDraht, CUntersuchung.*

Die **Klassen zur Modellierung des Behandlungsprozesses** bilden den Verlauf eines Patienten während einer Behandlung nach. Zu ihnen gehören: *CPatient, CBehandlung, CAufnahme, CTherapie, CAmbulanteKontrolle.*

Die ursprünglichen Klassen *CVoruntersuchung* und*CNachuntersuchung* sind als Tupel in der Klasse *CBehandlung* aufgenommen worden, um so eine übersichtlichere Darstellung mit *O2Look* zu ermöglichen.

Schließlich existieren noch **Klassen zur Modellierung der Behandlungsdaten**. Diese Klassen entsprechen Containern, in denen die während der Behandlung anfallenden Daten eingetragen werden. Diese Klassen sind: *CDoppler, CGehstrecke, CDiagnose, CProcedere, CSeite, CPunktion, CAusfluß, CLeiden, CAVKStadien.*

Die Klasse *CMedAktion* vererbt das Datum an alle Klassen, die Untersuchungen entsprechen (*CDoppler, CGehstrecke, ...*).

1.4.5 Benutzung des Prototypen

Der Prototyp des *Therapiekontrollsystem für die Angiographie* ist in O_2mit O_2*C*implementiert worden. Für die graphische Oberfläche ist das Graphik-Tool O_2*Look*verwendet worden, wodurch die aufwendige Programmierung einer eigenen Oberfläche entfallen ist und der Prototyp schnell lauffähig war. Um mit O_2arbeiten zu können ist es notwendig im Home-Verzeichnis von O_2(zu erreichen mit *cd $O2HOME* nach Schritt 4 unten) in der Datei*system* Zugriffsrechte auf ein System zu haben. Das Ändern dieser Datei ist nur Mitgliedern der *O2Group* möglich.

Um den Prototypen zu starten, ist folgendes zu tun:

1. rlogin auf bern
2. *setenv DISPLAY eigenerRechner:0.0* (auf bern)
3. *xhost bern* (auf eigenerRechner)
4. *module add O2* (auf bern)
5. *o2server -system systemname* (ab hier alles auf bern)
6. *o2 -system systemname* , soll die graphische Entwicklungsumgebung O_2*Tools*verwendet werden, so muß der Aufruf *o2 -system systemname -toolsgraphic* heißen.
7. Ist die graphische Entwicklungsebene nicht verwendet worden, so erscheint jetzt die *O2Shell.* Eingaben in dieser Shell werden grundsätzlich mit RETURN und CTRL-D abgeschlossen.
8. Einlesen des Skriptes mit # "*skriptname*". Anstelle des Skriptnamens muß der Pfad zur Datei *klassen191295*, welche sich zur Zeit im Verzeichnis *radmache/docs* befindet, angegeben werden (Empfehlung: kopiert das Skript in das Verzeichnis aus dem Ihr O_2startet. Dann den Aufruf # "*klassen191295*" ausführen). Das Skript muß nur beim ersten Aufruf eingelesen werden, danach ist es gespeichert und kann von Hand aktiviert werden (mehr dazu weiter unten).
9. Laut Diplomarbeit soll die Applikation jetzt automatisch starten, was sie bei mir nicht getan hat. Dies kann daran liegen, das ein Bitmap fehlt, welches von der Applikation vergeblich gesucht wird. Die Applikation läuft trotzdem.
10. Für den Fall, daß die Applikation nicht automatisch startet oder das Skript bereits in einer vorherigen Sitzung eingelesen worden ist, sind folgende Befehle in der*O2Shell* auszuführen:*set schema iu; set base iu2; run application angio.*
11. Ein kleines Fenster irgendwo auf dem Bildschirm repräsentiert nun die Applikation. Zu einigen Benutzerhinweisen kommt später noch etwas.
12. Nach Beenden der Applikation muß die *O2Shell* mit *quit* verlassen werden.
13. Um den Server wieder herunterzufahren, ist der Befehl *o2shutdown -system systemname* notwendig.

Zur Bedienung der Applikation:

Beim Start erscheint ein kleines Fenster auf dem Bildschirm, welches die Applikation repräsentiert. Durch drücken der rechten Maustaste auf dem Schriftzug *angio* erhält man eine Auflistung der Methoden, die ausgeführt werden können. In der linken oberen Ecke befindet sich das Symbol eines *Radiergummis*, mit welchem die Applikation verlassen werden kann. Wählt man eine Methode aus, so öffnet sich ein weiteres Fenster, entsprechend dem Aufruf, z.B. *Neuer_Patient*. Hier existiert in der oberen linken Ecke ein weiteres Symbol, ein *Bleistift*. Verläßt man ein Fenster über dieses Symbol, so werden die zuvor eingetragenen Werte gespeichert. Verläßt man ein Fenster über das *Radiergummi-Symbol*, so werden die eingetragenen Werte verworfen. In einem Fenster können unterschiedliche Arten von Attributen existieren. *Expandierte* Attribute können direkt eingegeben werden. Man erkennt sie am Cursor. *Nicht expandierte* Attribute sind umrandet dargestellt. Die Methoden *nicht expandierter* Attribute können durch klicken mit der rechten Maustaste aus das Attribut ausgewählt werden. Die Methoden des aktuellen Fensters lassen sich über den Button*Methoden* ausführen. Hinweis: Es ist z.B. möglich im Fenster *Therapie* den Untersucher anzugeben, in dem man das Attribut *Untersucher* expandiert. Hier muß man nun das Fenster für *Neuer_Arzt* ausfüllen. Dieser neue Arzt wird allerdings nicht in der Liste der Ärzte gespeichert (Programmierfehler ??). Alternativ ist es möglich über den Button *Methoden* des Fensters *Therapie* die Methode *Neuer_Untersucher* auszuwählen, woraufhin man aus einer Liste vorhandener Ärzte auswählen kann. Vorschlag: Lieber die zweite Alternative wählen. Die geöffneten Fenster können nur in der Reihenfolge, in der sie geöffnet worden sind, auch wieder geschlossen werden.

1.5 Das O_2 ODMG Datenbank-System

Verfasser: Sven Gerding

1.5.1 Einleitung

O_2 ist ein objektorientiertes Datenbanksystem, das dem ODMG-Standard [Cat96] entspricht. Es besteht im wesentlichen aus der *O_2Engine* die auf dem speziellen Filesystem *O_2Store* aufsetzt. Zur DB-Programmierung bietet O_2 Schnittstellen für C und C++ an, besitzt aber darüber hinaus mit *O_2C* auch eine an C angelehnte 4GL. Als Anfragesprache steht OQL zur Verfügung, die in C und C++ eingesetzt werden kann, jedoch sind mit OQL auch ad hoc Anfragen möglich.

Zusätzlich werden APIs zu *O_2Engine* und *O_2Store* angeboten. Zum Entwurf graphischer User-Interfaces dienen *O_2Look* und *O_2Graph*. *O_2Tools* und *O_2Kit* bieten Source-Browser, Klassenbibliotheken und diverse Editoren an.

1.5.2 O_2 System-Eigenschaften

Verwaltung von komplexen und multimedia Daten: Anders als ein relationales Datenbanksystem verwaltet O_2 nicht einfache atomare Werte in Tabellen, sondern Objekte, die neben atomaren auch komplexe-, zusammengesetzte- und Multimedia-Daten enthalten können. Diese Objekte sind in hierarchischen Graphen organisiert, auf die über eine sogenannte **persistent root** Zugegriffen wird.

Client/Server System: *O_2Engine* besitzt eine Client/Server Architektur. Auf einem Rechner läuft der Server-Prozess, mit dem die Client-Prozesse über ein Netzwerk kommunizieren.

Verteiltes System: O_2 speichert seine Daten in sogenannten Volumes. Diese müssen sich nicht notwendigerweise auf dem Rechner befinden, auf dem der Server-Prozess läuft, sondern können über ein Netzwerk auf mehrere Rechner verteilt sein. Ein verteiltes System, wie in den Regeln von Date gefordert, liegt allerdings nicht vor, es handelt sich hier vielmehr um die Möglichkeit, Daten nicht nur auf einem System zu halten sondern auf mehrere Rechner im Netz zu verteilen.

Verwaltung mehrerer Datenbanken: Eine Datenbank enthält **persistent roots**, die zu einem bestimmten Schema gehören. Es können mehrere Datenbanken existieren, die einem einzigen Schema zugrunde liegen. Diese Datenbanken können ihre Daten untereinander im- und exportieren.

1.5.3 DB-Organisation

In O_2 existieren zwei Strukturmechanismen: **Schema** und **Base**. Im Schema sind Objekte über ihre Klassendefinitionen logisch definiert, während eine Base Instanzen dieser Klassen beinhaltet. Dies bedeutet, daß eine Base nicht ohne ihr zugehöriges Schema existieren kann. Das Schema wird normalerweise beim Anwendungsentwurf definiert, kann aber auch später noch verändert werden (Schema update/evolution). In Kapitel 7 und 8 des **Administration Guide** [O296b] werden die Themen Schema und Base bzw.Schema updates genauer behandelt.

Das O_2-System speichert Daten in einem **named-system**, deren Zahl theoretisch unbeschränkt sein kann. Ein named-system wird von O_2 als einzelne logische Einheit betrachtet und enthält mehrere **volumes**, die Dateien oder ganze Disk-Partitionen sein können.
Ein named-system enthält folgende volumes:

1. **catalogue volume** - Der DB-Katalog mit Verweisen auf alle anderen volumes des jeweiligen named-systems

2. **log-volume** - Der DB-Log. Wichtig für Rollback und Recovery von Transaktionen.

3. **shadow-volume** - Temporärer Speicher für große Transaktionen.
4. **user-volume** - Hier sind die Benutzerdaten gespeichert, d.h. Schema und Base.

Um z.B. ein neues named-system einzurichten, muß zuerst der **systems** Datei im O_2 Verzeichnis ein entsprechender Eintrag hinzugefügt werden, anschließend wird mit dem Befehl **o2init** das neue System eingerichtet. Die Option `-system`, die bei vielen O_2 Befehlen angegeben werden muß bezieht sich also auf den Namen eines named-systems, während mit der Option `-server` der Host gemeint ist, auf dem der O_2 Server-Prozess läuft.

1.5.4 Administration

Neben dem Einrichten eines neuen Systems, dessen Backup und Recovery und der Schemaverwaltung umfaßt die O_2 Administration besonders das Thema Performance-Maximierung. Dazu gehören **DB-Restrukturierung**, **Tuning** , **Indexing** und **Clustering.**

Zur **DB-Restrukturierung** gehört u.a. die Freigabe von Plattenspeicher durch die Entfernung gelöschter oder isolierter Objekte und die Vergrößerung von Volumes. Beim **Tuning** geht es um die Anpassung des Daten-Caches von Client und Server. Standardmäßig sind diese auf jeweils 4MB eingestellt. Um nun eine optimale Cache-Größe zu ermitteln stellt das System Analysefunktionen zur Verfügung. Auch statistische Informationen des **o2monitor** lassen sich dazu verwenden.

Wie auch bei anderen DB-Systemen läßt sich unter O_2 der Zugriff auf Daten über die Verwendung einer **Indexierung** erheblich beschleunigen. Da auch die Indexverwaltung zeitaufwendig ist, sollte auf die Indexierung von Daten, die häufigen Änderungen unterliegen, verzichtet werden.

Eine weitere Möglichkeit, die Performance zu steigern, stellt das **Clustering** dar. Hierbei wird die Position von Daten auf dem Datenträger gezielt verändert. So können Daten, auf die oft abwechselnd zugegriffen wird, auf der Platte dicht beieinander untergebracht werden, wenn möglich in einer Page, um so den Disk I/O zu reduzieren.

Als letzter Punkt sollen hier **Schema-Updates** erwähnt werden, wobei zwischen logischer und physikalischer Schemamodifikation unterschieden wird. Der Unterschied besteht darin, daß sich eine logische Änderung nur auf das Schema und nicht die Objektstruktur auswirkt, z.B. das Hinzufügen einer neuen Methode zu einer bestehenden Klasse. Bei einer physikalischen Änderung müssen alle betroffenen Objekte in der DB angepaßt werden, was z.B. bei einer Modifikation eines Typs einer Klasse der Fall ist. Normalerweise wird der Update-Vorgang vom System automatisch durchgeführt, er ist aber auch durch eigene Update-Prozeduren ersetzbar.

1.5.5 C++ Binding

Die C++ Schnittstellte von O_2 ermöglicht die einfache Datenbankintegration von (bestehenden) C++ Anwendungen. Die ODMG Anfragesprache OQL ist ebenfalls aus C++ heraus möglich. Das Konzept ist dabei folgendes:

- Bestehende O_2-Klassen werden in C++ Klassen exportiert oder C++ Klassen in O_2 importiert und anschließend in das jeweilige DB-Schema integriert.
- Um C++ Klassen persistent zu machen, d.h. sie in der DB zu speichern muß auf sie über einen sog. persistent pointer zugegriffen werden.
- Ein Makefilegenerator steht zur Verfügung, der aus speziellen Konfigurationsdateien Makefiles generiert, die dann den Klassen Im- bzw. Export und die Kompilierung der Anwendungen weitgehend automatisiert.
- Der schreibende Zugriff auf persistent Objekte kann nur innerhalb einer Transaktion erfolgen.

1.5.6 O_2C

Neben dem C++ Interface stellt das O_2-System mit O_2C eine Programmiersprache der 4. Generation (4GL) zur Verfügung. O_2C ist eine Erweiterung von C und ermöglicht die Manipulation komplexer Objekte, die Erstellung von Benutzungsschnittstellen, Datenbankanfragen über OQL, Transaktionsmanagement und Schemadefinition und Schemamodifikation. Zu O_2C gehört auch der *O_2Debugger*. Das O_2C Beginners Guide [O296d] und O_2C Reference Manual [O296e] beschreiben nicht nur die Sprache O_2C sondern geben auch eine Einführung in O_2 und in die Konzepte objektorientierter Programmierung.

1.5.7 OQL

OQL ist die an SQL angelehnte objektorientierte Anfragesprache des ODMG-93 Standards. Sie kann entweder interaktiv für ad hoc Anfragen benutzt werden, oder aus O_2C, C oder C++ heraus. OQL Anfragen sind aus den Modulen o2, o2dsa und o2dba über das query Kommando absetzbar.

1.5.8 Entwicklungsumgebungen

O_2Kit stellt eine Klassenbibliothek für Klassen wie Date, Text, Bitmap, Image zur Verfügung. Deren genaue Verwendung wird im *O_2Kit* User Manual [O296g] anhand von O_2C Code beschrieben. Für die meisten Klassen werden dabei Programmbeispiele angegeben.

Des weiteren gehört zum *O_2Kit* die Hyper Facility und ein Widget Editor. Mit Ersterem lassen sich Fenster mit Buttons erzeugen, die bei Aktivierung ein bestimmtes Ereignis auslösen.

Der Widget Editor ermöglicht die Einbindung von Motif-Fenstern in eine *O_2Look* Anwendung.

O_2Tools ist das Standardentwicklungsystem zu O_2. Es beinhaltet Werkzeuge zur Erzeugung und Verwaltung vollständiger Datenbanken. Dazu gehören u.a. Schema-, Klassen-, Application- und Funktions Browser und Source Editoren. Über den eingebauten *O_2Shell* Editor kann man O_2 alphanumerisch benutzen, z.B. um OQL Anfragen zu abzusetzen.

Gestartet wird das System mit o2 -toolsgraphic.

1.5.9 Graphisches Interface

Das graphische Interface von O_2 umfaßt die Pakete *O_2Look* und *O_2Graph*.

O_2Look ist ein Generator für graphische Benutzungsschnittstellen. Seine Benutzung erfordert keine Programmierkenntnisse, da die GUI-Entwicklung Code-frei ist.

Die graphische Präsentation von Objekten in der Datenbank wird automatisch und unabhängig von deren Komplexität erzeugt. Eine Präsentation ist hier definiert als Fenster, in denen Objekte angezeigt und verändert werden können.
Mit *O_2Look* lassen sich Masken erzeugen, mit denen sich kontrollieren läßt, welche Attribute von Objekten angezeigt werden sollen und welche sich verändern lassen.

O_2Graph ermöglicht die Darstellung von Objekten in einem Graphen. Der Graph-Editor ist eine Maske, die dazu benutzt wird, um O_2 Werte, deren Typ einer bestimmten Struktur entspricht, darzustellen.

1.5.10 *O_2Engine* Bibliotheken

Die *O_2Engine* Bibliotheken erlauben eine direkte Programmierung von *O_2Engine* bzw. *O_2Store*. Über die *O_2Engine* API hat man in C oder C++ Zugriff auf Schema-, Objekt- und Transaktionsmanagement. *O_2Store* API ermöglicht sogar die dirkete Manipulation von Named-systems, Volumes und Indizes.

1.5.11 Literaturüberblick

[O296b] **O2 System Administration Guide**

Beschreibt Installation, Datenbankverwaltung und Maßnamen zur Performancesteigerung.

[O296c] **O2 System Administration Reference Manual**

Beschreibung der O2 Administrationsbefehle und deren Optionen.

[O296j] **ODMG C++ Binding Guide**

Einführung in die Konzepte der C++ Schnittstelle, Erklärung von Objekt Im- und Export und des Makefilegenerators. Benutzung von OQL und O2Look in C++ incl. Beispielprogramm.

[O296k] **ODMG C++ Binding Reference Manual**

Liste aller O2 Befehle und C++ Klassen mit Syntax und Optionen für jede Methode.

[O296d] **O2C Beginners Guide**

Einführung in O2 und die Konzepte der objektorientierten Programmierung.

[O296e] **O2C Reference Manual**

Beschreibung der O2C Datentypen, der O2C Anwendungsstruktur, aller O2C Methoden und Funktionen und des O2 Debuggers.

[O296l] **OQL User Manual**

Benutzung der Anfragesprache OQL für ad hoc Anfragen und in O2C, C++.

[O296g] **O2Kit User Manual**

Erklärung der Klassenbibliothek mit Beispielen. Beschreibung der Hyper Facility und des Widget Editor.

[O296i] **O2Tools User Manual**

Beschreibung von Browsern und Source Editoren.

[O296h] **O2Look User Manual**

Erklärung der O2Look Methoden und Funktionen.

[O296f] **O2Graph User Manual**

Beschreibung der O2Graph Klassenbibliothek.

1.6 *Oracle*

Verfasser: Andreas Dinsch

1.6.1 Einleitung

Diese Seminararbeit ist entstanden im Rahmen der Projektgruppe 290 im Fachbereich Informatik an der Universität Dortmund. Projektgruppen bilden einen Schwerpunkt im Studium der Informatik und sind mit in anderen Fachbereichen üblichen betrieblichen Praktika zu vergleichen.

Die Projektgruppe 290 beschäftigt sich mit dem Thema „Föderierte Objektorientierte Krankenhaus Informations-Systeme", kurz FOKIS. Eine Grundlage zur Realisierung eines Informationssystems bildet das Datenbanksystem. Die Verwaltung der projektspezifischen Daten bei FOKIS soll mittels des relationalen Datenbank Management Systems (RDBMS) *Oracle* geschehen.

In den folgenden Kapiteln sollen dem Leser im Hinblick auf die PG Einblicke in das Produkt *Oracle*, die ihm zugrundeliegende Architektur sowie die Möglichkeiten der Datenmanipulation mit SQL (Structured Query Language) in *Oracle* und deren Programmierung gegeben werden.

Dieses Papier erhebt weder den Anspruch auf Vollständigkeit, noch ersetzt es diverse Handbücher, die zu diesem Thema geschrieben wurden, sondern soll als Hilfestellung für den Einstieg in das RDBMS *Oracle* dienen.

1.6.2 Einführung in *Oracle*

Als E.F. Codd 1970 seine Ergebnisse der Erforschung von Datenbankproblemen, das Relationenmodell für Datenbanken, veröffentlichte, wurde der Grundstein für die Entwicklung verschiedener relationaler Datenbanken, die SQL als Zugriffssprache verwendeten, gelegt. Bevor IBM seine Systeme SQL/DS (1982) und DB2 (1985) auf den Markt brachte, entstand 1977 in Belmont, Kalifornien, die Firma *Oracle*, damals noch unter dem Namen Relational Software Inc. Heute beschäftigt *Oracle* weltweit mehr als 6500 Mitarbeiter, die sich überwiegend mit der Entwicklung des gleichnamigen relationalen Datenbank Management Systems (RDBMS) befassen.

Nachdem die erste Version auf DEC-Rechnern in Assembler entwickelt wurde, reprogrammierte man sie unter dem Betriebssystem VMS in C zur Version 3. Erste Ansätze zur Datenverteilung auf verschiedene Rechner realisierte man in der Version 4, die auch stabiler war und über einen höheren Funktionsumfang verfügte. Die nächste Entwicklungsstufe brachte die Unterstützung heterogener verteilter Datenbanken, d.h. der Zugriff auf andere Datenbanken unter anderen Betriebssystemen wurde ermöglicht (Version 5). Mit Version 6 paßte man die SQL-Schnittstelle an den erweiterten ISO-SQL-Standard an und führte das Online-Backup und Online-Recovery ein, was einen Nonstop-Betrieb erlaubte. Die wichtigsten Neuerungen der aktuellen Version 7 sind das Zwei-Phasen-Sperrprotokoll für die verteilte Transaktionsverarbeitung, die Einführung von PL/SQL zur Speicherung von Prozeduren in der Datenbank (stored procedures), sowie die hundertprozentige Anpassung an die ANSI-SQL-Norm. Ein weiteres Ziel ist es, *Oracle* auf möglichst allen Hardware-Plattformen zur Verfügung stellen zu können. Portierungen sind zur Zeit auf vielen gängigen Plattformem erhältlich, u.a. auf DOS, OS/2, Unix, AIX und Macintosh II.

1.6.3 Architektur von *Oracle*

Dieses Kapitel beschäftigt sich mit der Architektur des RDBMS *Oracle*, dabei insbesondere mit den für den Benutzer weitgehend verborgenen Komponenten, sowie mit der logischen Datenbankstruktur, wie sie der Benutzer beeinflussen und auch optimieren kann. Hierbei werden u.a. die Begriffe *System Global Area* (**SGA**), *Log Writer* (**LGWR**) oder *System Monitor* (**SMON**) erklärt, die Two-Task-Architektur mit Hilfe des Client/Server-Prinzips vorgestellt und der Vergleich zwischen der logischen und physischen Datenbankstruktur angestellt.

Die Komponenten des Datenbanksystems

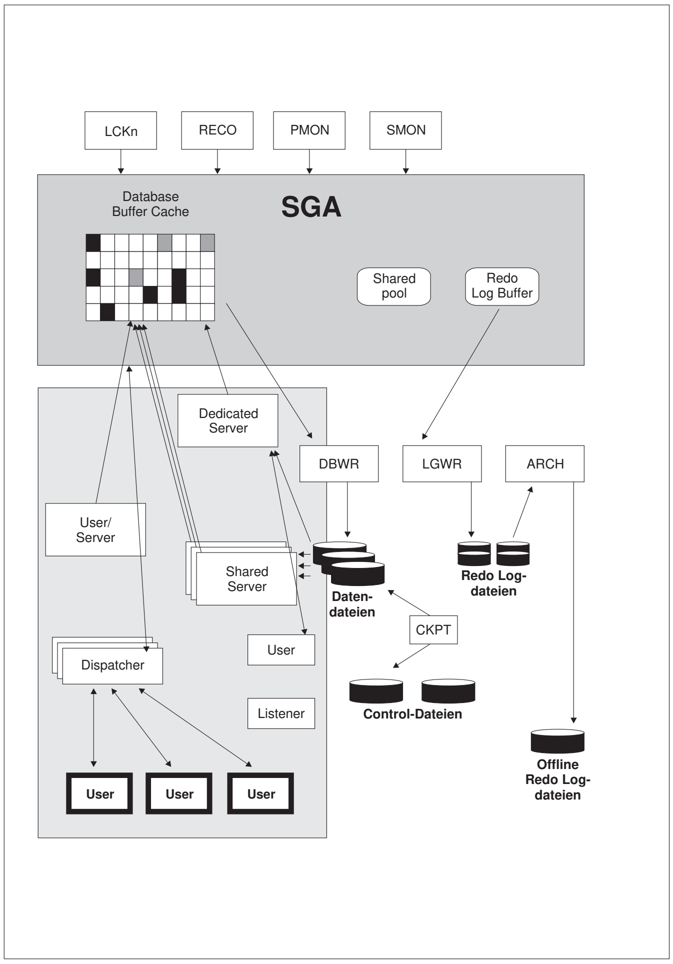


Abbildung 1.1: Die Komponenten des DBS *Oracle*

In Abbildung 1.1 auf Seite 32 wird der Zusammenhang zwischen den einzelnen Komponenten des Datenbanksystems erläutert. Diese können eingeteilt werden in einen passiven und einen aktiven Teil. Der passive Teil besteht dabei aus den eigentlichen Daten, die in Form von Dateien vorliegen und nur durch die Prozesse des aktiven Teils, der sogenannten *Instance*, manipuliert werden können.

Eine Instance setzt sich aus einem Speicherbereich, der *System Global Area* (**SGA**), und verschiedenen Prozessen, die auf die **SGA** zugreifen dürfen, zusammen. Aus der sehr komplexen Struktur der **SGA** sollen hier nur die drei wichtigsten Elemente, näher erklärt werden:

Database Buffer Cache Der Database Buffer Cache sorgt dafür, daß zu häufige Plattenzugriffe vermieden werden. Dies wird erreicht, indem alle Daten in Blöcken im Speicher gehalten und jeweils nach dem LRU-Prinzip ausgelagert werden. Somit werden die Daten, auf die oft zugegriffen wird, im Speicher gehalten und müssen nicht ständig von der Platte gelesen werden. Der schreibende Zugriff auf die Platte erfolgt über den *Database Writer* (**DBWR**), welcher später beschrieben wird.

Redo Log Buffer Der Redo Log Buffer beinhaltet alle sich veränderten Daten des *Database Buffer Cache*, bzw. nur die veränderten Bytes, welche über den *Log Writer* (**LGWR**) in die *Redo Log Dateien* geschrieben werden. Der *Log Writer* wird ebenfalls weiter unten erklärt.

Shared Pool In einer Multi-User-Umgebung haben häufig mehrere Prozesse Zugriff auf dieselben Daten. Um diese Zugriffe zu beschleunigen, ist in der **SGA** der *Shared Pool* integriert. In ihm werden Informationen aus dem Data Dictionary (z.B. Spaltendefinitionen, Index-Strukturen etc.) und bereits geparste SQL-Anweisungen oder -Programme bzw. Trigger gespeichert und für den gemeinsame Zugriff zur Verfügung gestellt.

Neben der **SGA** existieren diverse Hintergrundprozesse, um Das Datenbanksystem lauffähig zu halten. Davon wurden bereits der *Database Writer* und der *Log Writer* erwähnt. Hier sollen nun zusätzlich die Prozesse **ARCH**, **PMON** (Process Monitor) und **SMON** (System Monitor) erläutert werden.

DBWR Bei schreibenden Datenbankzugriffen unterscheidet man zwischen synchronem und asynchronem Schreiben. Während beim synchronen Schreiben alle Veränderungen sofort auf Platte zurückgeschrieben werden und somit Aktivitätsspitzen auftreten können, wird beim asynchronen Schreiben verzögert gearbeitet, d.h. Datenmanipulationen werden erst im Speicher vorgemerkt und zu einem späteren Zeitpunkt zurückgeschrieben. Somit wird ein gleichmäßiges Aktivitätsniveau erreicht. Der *Database Writer* (**DBWR**) übernimmt diese Aufgabe des asynchronen Schreibens, indem er alle 3 Sekunden diejenigen Datenblöcke aus der **SGA** auf Platte schreibt, die zwischenzeitlich verändert wurden. Da der **Database Buffer Cache** nach dem *Least Recently Used*-Prinzip arbeitet, können einzelne Veränderungen an Datenblöcken zu einem Schreibzugriff zusammengefaßt werden, was wiederum die Anzahl der notwendigen Zugriffe verringert und die Performance des Datenbanksystems steigert.

LGWR Der *Log Writer* ist der Hintergrundprozeß, der für die Erhaltung der Konsistenz des Datenbestandes der Datenbank zuständig ist. Wie bereits oben angesprochen, werden alle Änderungen im *Database Buffer Cache* byteweise im *Redo Log Buffer* festgehalten. Ist dieser Speicher bis zu einem gewissen Prozentsatz gefüllt, wird der *Log Writer* aktiviert und schreibt die Änderungen in die *Redo Log Dateien*. Zusätzlich löst der Abschluß einer Transaktion durch COMMIT eine Aktivierung des **LGWR** aus, um Transaktionen auch im Falle eines Absturzes zurückrollen zu können.

ARCH Der Archivierungsmechanismus **ARCH**, welcher von *Oracle* lediglich optional angeboten wird, bietet einen weiteren Schutz vor Datenverlust, indem er im Falle der Zerstörung externer Speichermedien die Lücke zwischen der letzten Gesamtsicherung des Datenbestandes durch ein Backup und ersten Aufzeichnung in den *Redo Log Dateien* schließt. Der **ARCH**-Hintergrundprozeß sorgt nun dafür, daß bei einem Plattencrash alle Daten, die nicht durch ein Backup oder die *Redo Log Dateien* abgedeckt sind, in sogenannten *Offline Redo Log Dateien* gespeichert wurden, die dann zur Wiederherstellung eines konsistenten Datenbestandes verwendet werden.

PMON Die Prozesse **DBWR**, **LGWR** und **ARCH** sind für die Verwaltung der Plattenzugriffe bzw. die Konsistenz der Daten auf der Platte zuständig. Was geschieht jedoch im Falle von irregulären Beendigungen von Prozessen durch den Benutzer (CTRL-C etc.), durch Stromausfall oder dergleichen? Der *Process Monitor* (**PMON**) prüft zu diesem Zweck in regelmäßigen Zeitintervallen, ob irgendwelche Ressourcen durch nicht mehr vorhandene Prozesse blockiert sind und gibt sie dann wieder frei. Desweiteren rollt er Transaktionen, die durch den Benutzer abgebrochen und noch nicht durch COMMIT beendet wurden, zurück.

SMON Der *System Monitor* überwacht das gesamte Datenbanksystem. Nach einem *Instance Failure*, z.B. durch Stromausfall, sorgt er für die konsistente Herstellung des Datenbestandes, indem er mit Hilfe der *Redo Log Dateien* die Veränderungen am Datenbestand nachvollzieht.

Die Client/Server–Architektur

Ein Benutzerprozeß besteht aus zwei Teilen, der **Benutzer-Funktionalität** und der ***Oracle*-Funktionalität**. Während der Benutzer meistens über SQL-Anweisungen oder -Programme mit dem Datenbanksystem kommuniziert, interessieren ihn die internen Vorgänge, die er damit auslöst, weniger bis gar nicht. Müßte er sich z.B. darum kümmern, welche Datenblöcke im Database Buffer Cache gerade frei sind

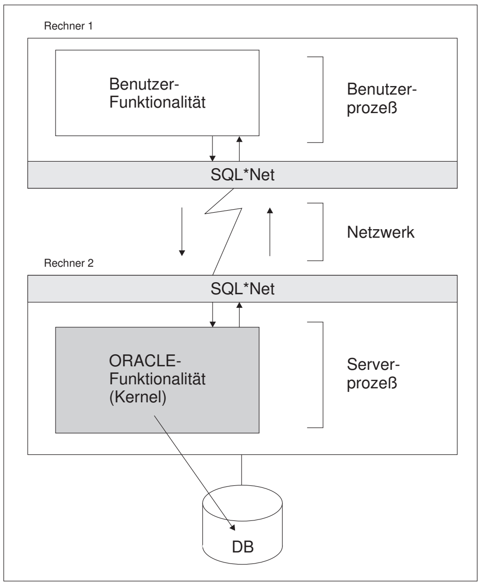


Abbildung 1.2: Die Client/Server–Architektur

und welche nicht, so würde jeder Benutzer Aufgaben übernehmen, die automatisch im Hintergrund durch die ***Oracle*-Funktionalität**, dem sogenannten **Kernel**, erledigt wird. Trennt man nun die beiden Funktionalitäten in zwei Prozesse, erhält man eine *Two-Task-Architektur*, die es ermöglicht, **Benutzerprozesse** mittels vom Betriebssystem bereitzustellender Interprozeß-Kommunikations-Mechanismen mit einem **Server-Prozeß** zu verbinden. Dies führt dazu, daß Benutzerprozesse nicht unbedingt auch auf demselben Rechner wie der Server-Prozeß laufen müssen. Über das die Rechner verbindende **Netzwerk** kommuniziert der Benutzer-Prozeß mit dem Kernel, der sich in diesem Fall auf demselben Rechner befinden muß wie das Datenbanksystem. Da dies hardwareunabhängig geschieht, ist auf beiden Rechnern, dem des Benutzer- und dem des Server-Prozesses, ein Vermittler zwischen Prozeß und Netzwerk-Protokoll erforderlich. Diese Aufgabe übernimmt die *Oracle*-Option **SQL*Net** (siehe auch Abb. 1.2). So läßt sich durch zentrale Datenhaltung und dezentrale Datenauswertung eine Lastverteilung erreichen, die mit einer optimalen Zuordnung der Aufgaben zu den verschiedenen Rechnern die Performance des Datenbanksystems steigert.

Die Datenbankstruktur

Wie in Abb. 1.3 zu erkennen ist, unterscheidet man die Datenbankstruktur in eine **logische** und eine **physische**. Die logische Datenbankstruktur ist diejenige, wie sie der Benutzer sieht, dem gegenüber hat der Datenbankadministrator, kurz DBA, den Blick auf eine Vielzahl von **Dateien**, die auf der Platte gespeichert und in **Betriebssystemblöcke** unterteilt sind. Aufgabe des Datenbanksystems ist es, die Verbindung zwischen beiden Strukturen herzustellen und dem Benutzer einen transparenten Blick auf seine Daten zu gewährleisten. So unterteilt sich die Datenbank aus logischer Sicht nicht in Dateien, sondern in **Tablespaces**, welche in **Segmente**, die wieder in **Extents** und schließlich in ***Oracle*-Blöcke** aufgeteilt sind.

Tablespace Der Tablespace ist die größte logische Einheit innerhalb der Datenbankstruktur und umfaßt mehrere verschiedenartige Objekte (Tabellen, Indizes, Cluster etc.). Verwaltet wird der Tablespace vom DBA, der die Anzahl und die Größe der in ihm vereinigten Dateien festlegt, in denen die oben genannten Objekte gespeichert werden. Dies hat den Vorteil, daß die physische

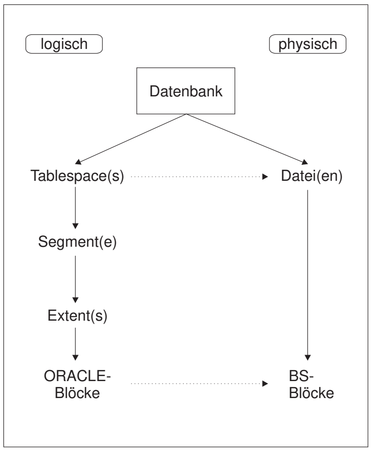


Abbildung 1.3: Die Datenbankstruktur

Struktur vom DBA jederzeit geändert werden kann, ohne daß der Benutzer etwas davon merkt, etwa zum Einspielen eines Backups von Platte B, da Platte A zerstört wurde. Außerdem kann die I/O-Last eines Tablespace durch Verteilung seiner Dateien auf mehrere Platten verringert werden.

Segment Das Segment bildet die logische Einheit aller zu einem Objekt gehörenden Daten und ist genau einem Tablespace zugeordnet. Da Segmente unterschiedliche Funktionen haben können, die von den in ihnen vorhandenen Daten abhängt, gibt es verschiedene Gruppen von Segmenten:

Daten-Segment Innerhalb der Daten-Segmente werden alle Daten einer *Tabelle* gespeichert, d.h. in ihnen befinden sich die eigentlichen, vom Benutzer verwalteten Daten. Dies wird mit den SQL-Statements `CREATE/ALTER/DROP TABLE` erreicht. Zusätzlich lassen sich Tabellen zu sogenannten *Clustern* zusammenfassen, die mit den Befehlen `CREATE/ALTER/DROP CLUSTER` verwaltet werden, und mit denen sich der Speicherplatz innerhalb der Daten-Segmente steuern läßt.

Index-Segment Mit den SQL-Statements `CREATE/ALTER/DROP INDEX` werden Indizes über einer Tabelle verwaltet. Die dazu benötigten Daten sind im Index-Segment zusammengefaßt. Ein Index reduziert die Anzahl der Suchschritte in einer Tabelle erheblich, indem über den Werten einer oder mehrerer Spalten eine Baumstruktur, realisiert durch ausgeglichene *B*-Bäume*, aufgebaut wird. Verwendet man beim Aufbau eines Index die *Unique*-Option, so werden keine doppelten Werte auftreten.

Rollback-Segment Rollback-Segmente werden vom DBA durch die Befehle `CREATE/ALTER/DROP ROLLBACK SEGMENT` verwaltet. Sie enthalten die für die Lesekonsistenz erforderlichen Daten, um im Bedarfsfall einen älteren, aber konsistenten Datenbankzustand zu rekonstruieren.

Temporäre Segmente Wird bei einer Datenbankabfrage die Klausel `ORDER BY`, `GROUP BY`, etc. zum Sortieren der Ergebnismenge benutzt, und ist der Arbeitsspeicher für diese Aktion (*Sort Area*) zu klein, so werden die Daten zwischenzeitlich in temporären Segmenten gespeichert. Das Anlegen sowie das Löschen dieser Segmente geschieht automatisch.

Extent Ein Extent ist ein Speicherbereich, der immer vollständig in einer Datei liegt. Er belegt nach Möglichkeit, d.h. wenn es das Betriebssystem zuläßt, lückenlos nebeneinander liegende Speicherbereiche. Die in allen relevanten SQL-Statements `STORAGE`-Klausel ermöglicht dem Benutzer die Verwaltung dieser Extents.

***Oracle*-Block** Ein *Oracle*-Block ist die kleinste Einheit, die das Datenbanksystem besitzt. Aus ihnen sind die Extents zusammengesetzt, d.h. ein Extent besteht aus einem oder mehreren *Oracle*-Blöcken, die wiederum aus jeweils mindestens einem vollständigen Betriebssystem-Block. Die

Blöcke werden von *Oracle* selbst verwaltet, wodurch in ihnen sowohl Daten, als auch Verwaltungsinformationen gespeichert werden müssen.

1.6.4 Datenmanipulation mit SQL und SQL*Plus

An dieser Stelle die Abfragesprache **SQL** im vollen Umfang darzustellen, wäre sicherlich utopisch und ist auch im Rahmen dieses Seminars nicht vorgesehen. Hier sollen nur diejenigen Statements vogestellt werden, die für den Einstieg in **SQL*Plus**, der Eingabe-Oberfläche von *Oracle*, nötig sind.

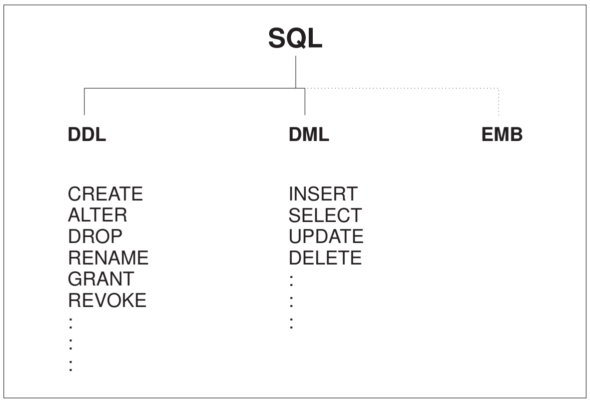


Abbildung 1.4: SQL-Befehle im Überblick

Dabei unterscheidet man zwischen Befehlen zur Manipulation von Datenbank-Strukturen und deren Inhalten. Erstere bilden die **Data Definition Language** (DDL), letztere die **Data Manipulation Language** (DML). Die in Abb. 1.4 gezeigte Gruppe der **Emb**edded SQL-Statements umfaßt die Befehle, die DDL- und DML-Befehle in prozedurale Programmiersprachen einbetten.

Data Definition Language DDL

Wie schon erwähnt handelt es sich bei der *Data Definition Language* um Befehle, die die Struktur der Datenbank beeinflussen. Dabei geht es um die Erstellung, Änderung und das Löschen von Datenbankobjekten, die Erteilung von Rechten an Objekten etc. Im folgenden sollen die in Abb. 1.4 aufgeführten DDL-Befehle kurz erklärt werden.

CREATE Der Create-Befehl erzeugt neue Datenbank-Objekte. Die von Benutzern anzulegenden Objekte sind:

- TABLE
- CLUSTER
- INDEX
- VIEW
- SYNONYM
- SEQUENCE
- SPACE
- DATABASE LINK

Die Objekte

- DATABASE

- TABLESPACE
- PARTITION
- ROLLBACK SEGMENT

lassen sich nur durch den DBA anlegen, der außerdem das Recht besitzt, Objekte jeglicher Art zu Lasten von Benutzern zu definieren, indem er vor den Objektnamen die Benutzerkennung angibt. Abbildung 1.5 zeigt eine mögliche Form des `CREATE TABLE`-Befehls, Einzelheiten müssen

```
CREATE TABLE Tabellenname
    (SPALTENNAME DATENTYP [NOT NULL]
     [,SPALTENNAME DATENTYP [NOT NULL]] ...
    )
    [SPACE SPACENAME [PCTFREE n]
     | CLUSTER CLUSTERNAME (SPALTENNAME
                  [,SPALTENNAME] ...
                 )
    ]
```

Abbildung 1.5: Der `CREATE TABLE`-Befehl

in Handbüchern nachgelesen werden.

ALTER Über den ALTER-Befehl lassen sich bereits erstellte Datenbankobjekte nachträglich ändern. Die folgenden Objekte sind davon betroffen:

- TABLE
- CLUSTER
- INDEX
- SEQUENCE
- SPACE
- DATABASE
- TABLESPACE
- PARTITION
- ROLLBACK SEGMENT
- USER

DROP Mittels `CREATE`-Befehl definierte Objekte werden mit dem `DROP`-Befehl wieder gelöscht.

RENAME Zum Ändern des Namens einer Tabelle, eines Synonyms oder einer View verwendet man den `RENAME`-Befehl. Privilegien und Indexe werden dabei übernommen. Die Syntax des Befehls lautet `RENAME` *`ALTERNAME`* `TO` *`NEUERNAME`*.

GRANT Mit dem `GRANT`-Befehl werden Datenbank- bzw. Objektprivilegien erteilt. Während Datenbankprivilegien vom DBA erteilt werden, kann der Benutzer Objektprivilegien erteilen, die sich auf die Manipulation von Daten beziehen. Außerdem kann ein Benutzer mit diesem Befehl sein eigenes Passwort ändern.

REVOKE Rechte, die mit dem `GRANT`-Befehl an Benutzer vergeben wurden, können mit dem `REVOKE`-Befehl wieder entzogen werden. `GRANT` und `REVOKE` sind somit unmittelbare Gegenspieler.

```
DROP {TABLE TABELLENNAME |
     CLUSTER CLUSTERNAME
      [INCLUDING TABLES] |
     INDEX INDEXNAME [ON TABELLENNAME] |
     VIEW VIEWNAME |
     [PUBLIK] SYNONYM SYNONYMNAME |
     SEQUENCE SEQUENCENAME |
     [PUBLIK] DATABASE LINK LINKNAME |
     SPACE [DEFINITION] SPACENAME |
     [PUBLIK] ROLLBACK SEGMENT SEGMENTNAME |
     TABLESPACE TABLESPACENAME
      [INCLUDING CONTENTS]}
```

Abbildung 1.6: Der `DROP`-Befehl

Data Manipulation Language DML

Bei der Vorstellung der Befehle zur Datenmanipulation beschränke ich mich auf die vier wesentlichen Befehle zum Auswählen, Einfügen, Verändern und Löschen von Tabelleninhalten. Im Einzelnen sind dies die Befehle `SELECT`, `INSERT`, `UPDATE` und `DELETE`.

SELECT Der `SELECT`-Befehl ist der umfangreichste der hier beschriebenen Befehle. Mit ihm werden Daten aus einer Tabelle oder View selektiert. Um eine genaue Beschreibung der gewünschten Daten zu erzielen, besitzt `SELECT` eine Menge von Klauseln, die in Abb. 1.7 zu sehen sind. Vielfach wird `SELECT` auch als Unterabfrage in anderen Befehlen verwendet.

```
SELECT [ALL | DISTINCT]
   {* | OBJEKTNAME.*
    | AUSDRUCK [A_ALIAS]
   }
   [,{OBJEKTNAME.* | AUSDRUCK [A_ALIAS]}]...
   FROM [BENUTZERKENNUNG.]OBJEKTNAME [O_ALIAS]
   [,[BENUTZERKENNUNG.]OBJEKTNAME [O_ALIAS]]...
   [WHERE BEDINGUNG]
   [CONNECT BY BEDINGUNG
    [START WITH BEDINGUNG]]
   [GROUP BY AUSDRUCK [,AUSDRUCK]...
    [HAVING BEDINGUNG]]
   [{UNION | INTERSECT | MINUS} SELECT...]
   [ORDER BY {AUSDRUCK | POSITION}[ASC|DESC]
    [,{AUSDRUCK | POSITION}[ASC|DESC]]...
   ]
   [FOR UPDATE OF SPALTENNAME [,SPALTENNAME]...
    [NOWAIT]
   ]
```

Abbildung 1.7: Der `SELECT`-Befehl

INSERT Mit `INSERT` (Abb. 1.8) werden neue Sätze in eine Tabelle oder eine View eingefügt.

```
INSERT INTO [BENUTZERKENNUNG.]OBJEKTNAME
   [(SPALTENNAME [,SPALTENNAME]...)]
   {VALUES (WERT [,WERT]...) | UNTERABFRAGE}
```

Abbildung 1.8: Der `INSERT`-Befehl

```
   UPDATE [BENUTZERKENNUNG.]TABELLENNAME [ALIAS]
      SET SPALTENNAME = AUSDRUCK
      [,SPALTENNAME = AUSDRUCK]...
      [WHERE BEDINGUNG]
oder
   UPDATE [BENUTZERKENNUNG.]TABELLENNAME [ALIAS]
      SET (SPALTENNAME [,SPALTENNAME]...)
        = (UNTERABFRAGE)
      [,(SPALTENNAME [,SPALTENNAME]...)
        = (UNTERABFRAGE)]...
      [WHERE BEDINGUNG]
```

Abbildung 1.9: Der `UPDATE`-Befehl

UPDATE `UPDATE` (Abb. 1.9) verändert Daten in einer Tabelle.

DELETE Um einzelne Sätze, selektiert durch die `WHERE`-Bedingung, oder alle Sätze einer Tabelle zu löschen, benutzt man den `DELETE`-Befehl (Abb. 1.10).

```
DELETE FROM [BENUTZERKENNUNG.]TABELLENNAME
   [ALIAS]
   [WHERE BEDINGUNG]
```

Abbildung 1.10: Der `DELETE`-Befehl

1.6.5 Programmieren mit PL/SQL

PL/SQL stellt eine Erweiterung der strukturierten Abfragesprache SQL in Richtung einer Programmiersprache dar. Dabei steht PL/SQL für „Programming Language/SQL“. Neben den SQL-Statements bietet es die Möglichkeit, Variablen und Konstanten zu deklarieren, Schleifen und Verzweigungen zu definieren und auszuführen und Ausnahmebedingungen zu setzen. PL/SQL ist im Gegensatz zu SQL nicht mengen-, sondern satzorientiert, was die Einführung eines sogenannten „Cursors“ erfordert. Ein Cursor ist ein Zeiger auf einen Satz einer Ergebnismenge von Sätzen, die ein SQL-Befehl liefert.

Die Block–Struktur

PL/SQL ist eine block-orientierte Sprache. Jeder Block besteht aus maximal drei Teilen:

- einem Deklarationsteil
- einem Ausführungsteil

- einem Ausnahmebedingungsteil.

Der Ausführungsteil ist in jedem Block Pflicht, die anderen Teile sind optional.

Deklarationsteil Folgende Objekte werden im Deklarationsteil deklariert:

- Variablen
- Konstanten
- Sätze
- Cursor
- Ausnahmebedingungen

Der Deklarationsteil wird durch das Schlüsselwort `DECLARE` eingeleitet.

Ausführungsteil Im Ausführungsteil befinden sich, wie der Name schon vermuten läßt, ausführbare Befehle. Dies können sowohl PL/SQL-, als auch SQL-Befehle sein, mit Ausnahme der DDL-Kommandos. Der Ausführungsteil beginnt mit dem Schlüsselwort `BEGIN`.

Ausnahmebedingungsteil Tritt eine im Deklarationsteil aufgeführte Ausnahmebedingung in Kraft, so verzweigt sie in den Ausnahmebedingungsteil des Blocks und führt hier die entsprechenden Aktionen aus. Das Schlüsselwort `EXCEPTION` markiert den Beginn des Ausnahmebedingungsteils.

Variablen und Konstanten

Für Variablen und Konstanten, die im Deklarationsteil eines PL/SQL-Blockes deklariert werden, gibt es vier Arten von Datentypen:

NUMBER(m,n)	Numerische Literale der Länge m mit n Nachkommastellen.
CHAR(n)	Alphanumerische Literale der Länge n.
DATE	Datum und Uhrzeit.
BOOLEAN	TRUE oder FALSE.

Konstanten werden durch das Schlüsselwort `CONSTANT` definiert, indem es vor die Typbezeichnung geschrieben wird. Die Wertzuweisung erfolgt durch ":= *WERT*" hinter der Typbezeichnung.

Kontroll–Strukturen

Als Kontrollstrukturen stehen dem PL/SQL-Anwender bedingte und unbedingte Verzweigungen sowie Schleifen zur Verfügung. Die Grundform der bedingten Verzweigung lautet:

```
IF BEDINGUNG1 THEN ANWEISUNG1
   [ELSIF BEDINGUNG2 THEN ANWEISUNG2]...
   [ELSE ANWEISUNGn]
END IF;
```

Die unbedingte Verzweigung wird durch "`GOTO` *label_name*`;`" realisiert. Dazu muß im Programm ein Label gesetzt sein, welcher durch ein in doppelte spitze Klammern eingefaßter Name ausgedrückt wird (≪`label_name`≫).
Schleifen können in PL/SQL auf verschiedene Arten gebildet werden. Allen gemeinsam ist die Schachtelung der zu wiederholenden Anweisungen in die Schlüsselwörter `LOOP` und `END LOOP;`. Stehen diese alleine, so werden alle eingeschlossenen Anweisungen solange durchgeführt, bis ein `EXIT` erreicht wird, oder mit einer `GOTO`-Anweisung die Schleife verlassen wird. Durch das Voranstellen einer Bedingung durch `WHILE` *BEDINGUNG* wird die Schleife nur dann ausgeführt, wenn die Bedingung wahr ist. Eine weitere Alternative bietet die `FOR`-Schleife. Dabei wird ein Zähler mit jedem Schleifendurchlauf um 1 erhöht, bis das Ende des Wertebereichs erreicht ist. Allgemein lautet die Schleifen-Anweisung:

```
[WHILE BEDINGUNG | FOR ZÄHLER]
LOOP ANWEISUNGEN END LOOP;
```

1.7 Die Object Modeling Technique (OMT)

Verfasser: K.-H. Schulte

OMT [RBP+91] dient zur Beschreibung eines Systems aus drei Gesichtspunkten: objektorientiert, funktional und dynamisch. Für jede dieser Sichtweisen gibt es in OMT ein Modellierungskonzept: das Objektmodell, das dynamische Modell und das funktionale Modell.

1.7.1 Charakterisierung und Abgrenzung der Modelle im Überblick

Objektmodell

Beschreibt:	Klassen, Objekte, Beziehungen, Verbindungen, Attribute, Methoden
Sicht:	statisch, objektorientiert
Kernfrage:	*Womit* geschieht etwas?
Notation:	graphische Objektdiagramme

Dynamikmodell

Beschreibt:	Zeitpunkte und Folgen von Methodenausführungen
Sicht:	dynamisch
Kernfrage:	*Wann* geschieht etwas?
Notation:	Zustandsübergangsdiagramme, Interaktionsdiagramme

Funktionsmodell

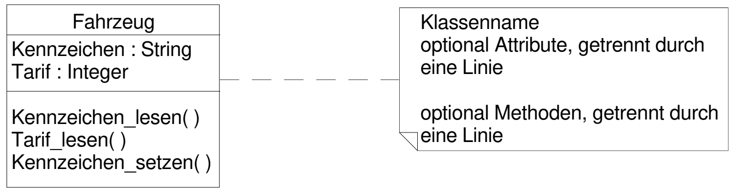
Beschreibt:	funktionale Abhängigkeiten und Wertemanipulationen
Sicht:	transformatorisch, funktional
Kernfrage:	*Was* geschieht?
Notation:	Datenflußdiagramme

1.7.2 Das Objektmodell und seine Notation

Mit dem Objektmodell werden Klassen, Objekte und ihre Struktur innerhalb eines Systems dargestellt. Es handelt sich also um eine statische Sichtweise die „Daten"-Aspekte klären soll. Da OMT ein objektorientierter Ansatz ist, liegt auf dem Objektmodell eine besondere Gewichtung. Es liefert den Rahmen innerhalb dem das dynamische und funktionale Modell eingebettet sind. Fälschlicherweise werden die Begriffe OMT und Objektmodell häufig synonym gebraucht.

Klassen und Objekte

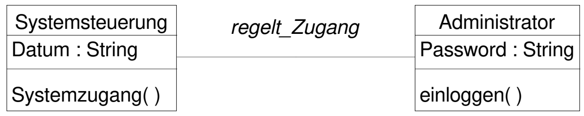
Kernelemente dieser Diagramme sind **Klassen**. Das Werkzeug *Rose* ermöglicht es, alle beliebigen Elemente in allen beliebigen Diagrammen mit einer *Notiz* zu versehen. Das grafische Element dafür ist ein „Zettel mit Eselsohr". Dies ist allerdings nicht methodenspezifisch. Die Notation ist wie folgt:



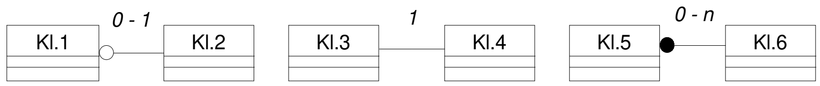
Diese Notation wird **Klassendiagramm** genannt. Die Angabe von Attributen, Methoden und Datentypen ist optional und kann somit dem jeweiligen Entwicklungsstand angepaßt werden. Klassendiagramme beschreiben den Allgemeinfall. Ergänzend dazu gibt es auch noch **Instanzdiagramme**, um das Verstehen komplexer Klassendiagramme zu erleichtern. Instanzdiagramme stellen also Objekte dar. In der Notation sind lediglich die Ecken des Klassenrahmens abgerundet und die Attribute erscheinen mit ihrer konkreten Belegung.

Beziehungen und Verbindungen

Attribute sollen in OMT nur elementare Datentypen aufnehmen, jedoch keine Objekte. Solche Beziehungen sollten als Assoziation modelliert werden. **Beziehungen** bestehen zwischen Klassen, z. Bsp.:



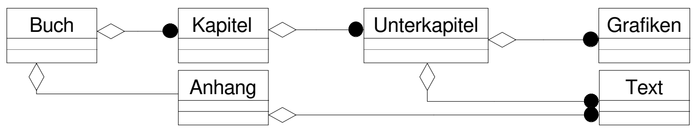
Zwischen Objekten bestehen Verbindungen (Links). Die Notation von Verbindungen ist der von Beziehungen äquivalent. Die **Kardinalität** kann ebenfalls durch die Notation ausgedrückt werden. Folgendes Diagramm zeigt die drei vorhandenen Notationselemente und die zugehörigen Kardinalitäten als Annotation.



Beziehungen höherer Stelligkeit sind auch möglich. Dazu wird in OMT eine Raute im Schnittpunkt der Assoziationslinie benutzt. *Rose* bietet dieses Notationselement nicht an. Programmiersprachen unterstützen in der Regel keine Konzepte zur direkten Implementierung von Beziehungen. Links werden häufig durch Referenzen von einem Objekt zum anderen gebildet; dieses ist allerdings keine konzeptionelle Sicht.

Aggregation als eine spezielle Form der Beziehung

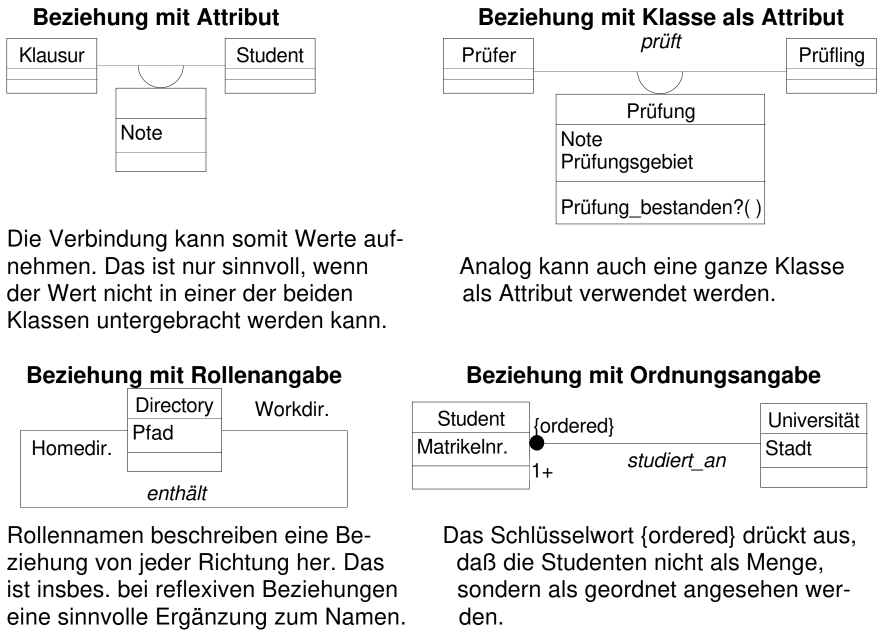
Die Notation besagt für diesen Fall, daß eine kleine Raute an die aggregierte Klasse angehängt wird und alle zu aggregierenden Klassen an diese Raute angehangen werden. *Rose* wandelt diese Notation nun ein wenig um, indem es für jede Klasse eine eigene Raute anhängt, wie folgende Grafik zeigt:



Aggregation sollte nur dann benutzt werden, wenn die Summe der Teile das Ganze ergibt.

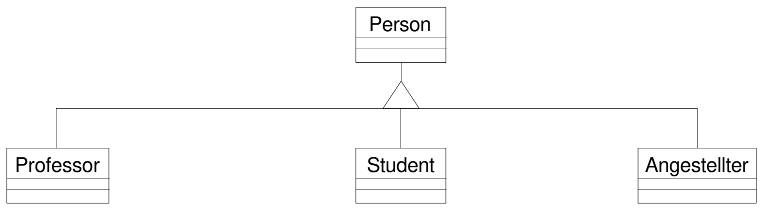
Fortgeschrittene Konzepte zu Beziehungen und Verbindungen

Auf die fortgeschrittenen Konzepte soll an dieser Stelle nicht weiter eingegangen werden. Zur attributierten Beziehung ist lediglich zu erwähnen, daß es sich um eine reine Entwurfsmodellierung handelt. Programmiersprachen unterstützen dieses Konzept in der Regel nicht. In der Implementierung wird der Programmierer einen Kompromiß schließen müssen. Also entweder das Attribut in einer oder sogar mehreren der beteiligten Klassen unterbringen oder aber eine weitere Klasse dafür entwerfen.



Vererbung

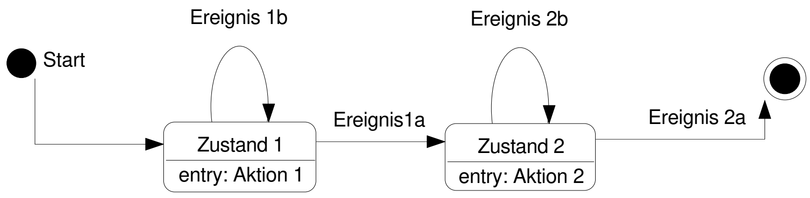
Das Notationselement zur Angabe der Vererbungsbeziehung ist ein kleines Dreieck. Die Erb- bzw. Vererbrichtung ergibt sich aus der intuitiven Semantik des folgenden Beispiels:



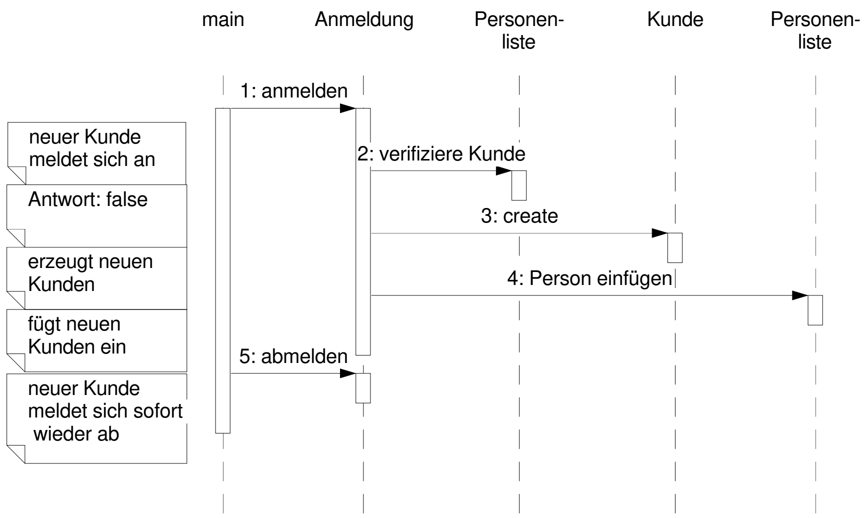
Abstrakte Klassen werden in OMT zusätzlich mit dem Schlüsselwort {abstract} gekennzeichnet. Auch Mehrfachvererbung ist in OMT vorgesehen.

1.7.3 Das Dynamikmodell und seine Notation

Das Dynamikmodell soll die „Kontroll"-Aspekte darstellen. Die Dynamik wird in Form von **Zustandsübergangsdiagrammen** modelliert. Die Notation ist von *Harel* [Har87] übernommen und daher soll an dieser Stelle ein kleines Beispiel genügen:



Ein Zustandsübergangsdiagramm beschreibt das Verhalten einer Klasse. Ereignisse sind Methoden des Objektmodells. Es werden alle möglichen Wege innerhalb des Lebens eines Objektes dargestellt. Ein Szenario ist ein spezieller Weg unter vielen möglichen. Solch ein **Szenario** kann mit Hilfe eines **Interaktionsdiagramms** dargestellt werden. Ein Beispiel:



1.7.4 Das Funktionsmodell und seine Notation

Das Funktionsmodell spezifiziert die Funktionalität der Methoden des Objektmodells und der Aktionen des Dynamikmodells. Zur Darstellung dienen **Datenflußdiagramme** der üblichen Form mit Prozessen, Quellen/Senken als Schnittstellen zur Umwelt und Datenspeichern [Bal96]. In OMT ist ergänzend die Modellierung von Kontrollfluß möglich und eine Diagrammhierarchie aufbaubar. *Rose* unterstützt dieses Modell nicht.

1.8 Die Booch-Methode

Verfasser: K.-H. Schulte

Die Notation der Methode ist das wesentliche Ausdrucksmittel der Vorgehensweise. Da sie sehr umfangreich ist, kann im Rahmen dieser Einführung nur ein Teil dargestellt werden, Booch nennt ihn die *Booch-Lite-Notation*. Für ein komplettes Verständnis der Notation ist Booch's Buch [Boo94] unabdingbar, für einen Überblick über sämtliche Notationselemente ist der Einband des Buches gut geeignet. Auch Booch betrachtet das Problem aus verschiedenen Sichten. Dazu verwendet er sechs verschiedene Diagrammarten. Er unterscheidet zwischen der physikalischen und der logischen Sicht, sowie zwischen der dynamischen und der statischen Semantik.

1.8.1 Charakterisierung und Abgrenzung der Diagramme im Überblick

Klassendiagramm

Beschreibt: Klassen, Beziehungen, Attribute, Methoden, Klassenkategorien
Sicht: statisch, logisch

Objektdiagramm

Beschreibt: Objekte, Attribute, Verbindungen
Sicht: dynamisch, logisch

Zustandsübergangsdiagramm

Beschreibt: Objektzustände, Methoden, alle möglichen Zustandsübergänge
Sicht: dynamisch, logisch

Interaktionsdiagramm

Beschreibt: Szenario (einen speziellen „Lebensverlauf" eines Objektes)
Sicht: dynamisch, logisch

Moduldiagramm

Beschreibt: Module
Sicht: statisch, physikalisch

Prozeßdiagramm

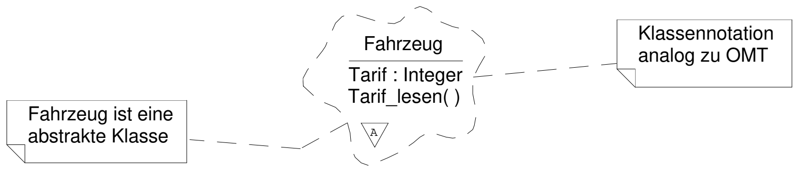
Beschreibt: Zuordnungen von Prozessen zu Prozessoren
Sicht: statisch, physikalisch

1.8.2 Das Klassendiagramm und seine Notation

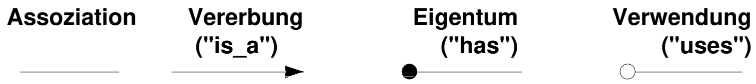
Das Klassendiagramm stellt die statische Sicht auf die Architektur dar. Die beiden wesentlichen Elemente sind Klassen und ihre Beziehungen.

Klassen

Zur Einführung von **Klassen** soll ein Beispiel genügen. Man beachte, daß die textuelle Notation in dem grafischen Klassensymbol identisch ist mit der Notation von OMT. Der „Zettel mit Eselsohr“ stellt auch hier wieder nur eine Notiz dar. Das kleine Dreieck mit dem „A“ drückt aus, daß es sich um eine abstrakte Klasse handelt.



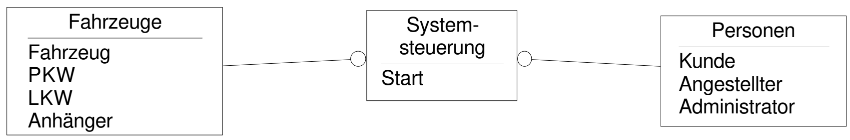
Beziehungen



(Die Leseweise der Icons ist von links nach rechts.) Eigentum und Verwendung sind nur Ergänzungen zur allgemeinen Assoziation. Die Eigentum-Beziehung ist als **Aggregation** zu verstehen. Wie in der OMT-Notation lassen sich auch hier **Kardinalitäten** angeben. In der Booch-Methode muß die Kardinalität allerdings als Klartext an die Beziehung notiert werden.

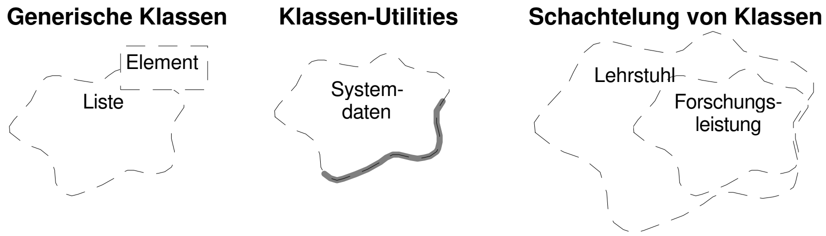
Klassenkategorien

Klassenkategorien dienen dazu, das logische Modell eines Systems aufzuteilen. Eine Klassenkategorie ist ein Aggregat aus Klassen und/oder Klassenkategorien. Nur wenige Programmiersprachen unterstützen dieses Konzept, daher dient es mehr dem Überblick bei der Analyse und beim Design. Ein Beispiel:



Fortgeschrittene Konzepte

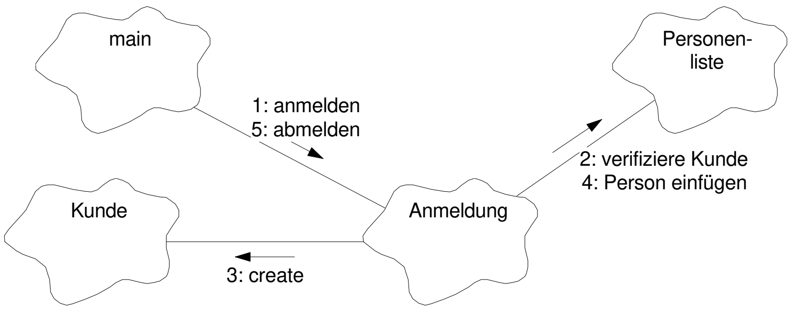
Die Elemente die bisher vorgestellt wurden entsprechen der *Booch-Lite-Notation*. Darüberhinaus bietet die Booch-Methode etliche weitere Notationen, um möglichst alle taktischen und strategischen Entscheidungen darstellen zu können die für das Verständnis des Gesamtsystems nötig sind. Unter anderem sind das:



Natürlich lassen sich generische Klassen auch instanziieren. Eine explizite Exportsteuerung ist genauso vorgesehen wie die Annotation der C++-Eigenschaften „static", „virtual" und „friend". Desweiteren läßt sich ein physikalisches Enthaltensein durch die Notation darstellen (d. h. eine physikal. Aggregation) und Assoziationen lassen sich mit Rollen, Einschränkungen und Attributen versehen (analog zu OMT).

1.8.3 Das Objektdiagramm und seine Notation

Das **Objektdiagramm** stellt eine Momentaufnahme eines sich ständig wandelnden Stroms von Ereignissen für eine bestimmte Konfiguration von Objekten dar, i. e. es wird ein spezielles **Szenario** beschrieben. Die Hauptelemente sind Objekte und ihre Verbindungen. Sequenzzahlen stellen den zeitlichen Ablauf dar. Ein Beispiel:



Die Synchronisationsart des Nachrichtenaustauschs kann durch verschiedene Pfeile ausgedrückt werden. Diese Notationselemente sollen hier nicht vorgestellt werden. Ein Werkzeug (wie z. Bsp. *Rose*) kann aus einem Objektdiagramm automatisch ein **Interaktionsdiagramm generieren**. Die Information bezüglich der Synchronisation geht dabei verloren, sonst sind die Diagramme aber semantisch äquivalent. Auch der umgekehrte Weg ist natürlich machbar: Obiges Diagramm hat *Rose* automatisch aus dem Beispielinteraktionsdiagramm für das Kapitel 3 aus Teil I generiert.

1.8.4 Das Zustandsübergangsdiagramm und seine Notation

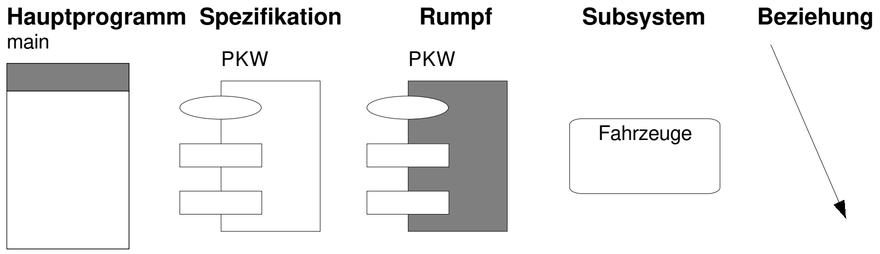
Zustandsübergangsdiagramme werden für Klassen angegeben die ein signifikantes ereignisgesteuertes Verhalten haben. Die Notation entspricht der von *Harel* [Har87]. Ein Beispiel ist bereits im OMT-Teil / 1.7.3 gegeben.

1.8.5 Das Interaktionsdiagramm und seine Notation

Interaktionsdiagramme sind semantisch nahezu äquivalent zu Objektdiagrammen. Die automatische Generierungsmöglichkeit durch *Rose* wurde bereits im dritten Kapitel dieses Teiles erwähnt. Ein Beispiel für ein Interaktionsdiagramm ist im OMT-Teil / 1.7.3 gegeben.

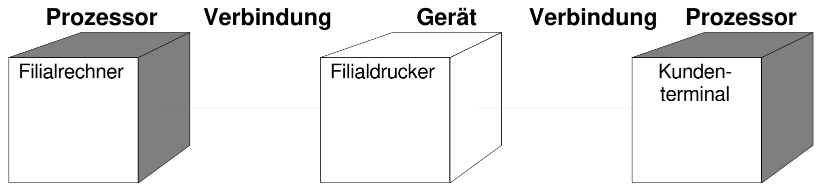
1.8.6 Das Moduldiagramm und seine Notation

Ein Moduldiagramm wird verwendet, um die Zuordnung von Klassen zu Modulen im physikalischen Systemdesign zu zeigen. Subsysteme sind Kategorisierungen von Modulen. Die einzige denkbare Beziehung ist die Compilierungs-Abhängigkeit.



1.8.7 Das Prozeßdiagramm und seine Notation

Ein Prozeßdiagramm wird verwendet, um aufzuzeigen, wie im physikalischen Systemdesign Prozesse den Prozessoren zugeordnet sind. Prozessoren können, bzgl. des zu bearbeitenden logischen Modells, Programme ausführen, Geräte können dies nicht. Verbindungen sind hier ebenfalls im physikalischen Sinne zu verstehen.



1.9 *Rational Rose*

Verfasser: K.-H. Schulte

Rose ist ein kommerzielles Entwicklungswerkzeug. Es ist für die Booch-Methode konzipiert worden und unterstützt diese hervorragend. Die OMT-Methode wird nur soweit unterstützt, wie sie semantisch äquivalent zur Booch-Methode ist. Das Werkzeug übernimmt ein automatisches Überführen der Diagramme gemäß der einen Methode in die entsprechenden Diagramme gemäß der anderen Methode. Etliche Feinheiten von OMT sind nicht realisierbar (z. Bsp. ternäre Beziehungen). Desweiteren unterstützt *Rose* die Use-Case-Analyse gemäß *Jacobson* [JCJO92]. Dieses Tool steht allerdings nicht mit den anderen in Zusammenhang und ist als Ergänzung zu sehen. *Rose* ist sprachenunabhängig, jedoch besonders abgestimmt auf C++. Generell gilt zu klären, welche Notationselemente eine Methode bietet und welche davon von dem verwendeten Werkzeug unterstützt werden.

1.10 C++

Verfasser: Djamel Kheldoun

1.10.1 Einführung

Die Progammiersprache C++ wurde von Bjarne Stroustrup während seiner Arbeit bei den AT&T Bell Laboratorien entwickelt. Frühe Versionen der Sprache 'C mit Klassen' gibt es seit 1980. Der Name C++ wurde von Rick Mascitti im Sommer 1983 geprägt. Der explosive Zuwachs in der C++ Anwendung hat einige Änderungen gebracht. 1987 wurde klar, daß eine formelle Standardisierung unumgänglich ist. Im Jahr 1989 wurde das X3J16-ANSI-Komitee einberufen und mit dieser Aufgabe beauftragt. Es kann davon ausgegangen werden, daß der ANSI-Standard Für C++ Bestandteil der internationalen ISO-Standardisierungsbemühungen für C++ wird. C wurde für C++ als Grundlage gewählt ,weil die Sprache

- für viele Aufgaben aus dem Bereich der Systemprogrammierung geeignet ist
- auf viele Systemen und Maschinen verfügbar ist
- in die UNIX-Programmierumgebung paßt.

Weitere Vorteile :

- es gibt einige Millionen Zeilen C-Code die weiterverwendet werden können
- die große Zahl der C-Programmierer muß nur die Neuerungen der Sprache C++ kennenlernen.

Ein anderer Haupteinfluß war die Sprache SIMULA67, das Klassenkonzept (mit abgeleiteten Klassen und virtuellen Funktionen) stammt hierher. C++ ist entworfen worden um:

- ein besseres C zu sein
- Datenabstraktion zu unterstützen
- objektorientiertes Programmieren zu unterstützen.

1.10.2 Klassen

Definition

Eine Klasse `class` ist ein benutzerdefinierter Typ. Es wird beschrieben, wie eine Datenstruktur geschützt, initialisiert, benutzt und schliesslisch wieder gelöscht wird. Ein Objekt (Instanz einer Klasse) besteht aus Daten und dazu gehörigen Funktionen (Methoden),die auf die Daten zugreifen dürfen. Eine Klasse wird in einem header-file `name.h` definiert und in einem C++-file `name.cc` implementiert. Eine Klasse wird wie folgt definiert

```
class  <classname> {
private:
    //  Klassenelemente mit Zugriffsrecht privat
    //  hier werden die Datenelemente deklariert
protected :
    // Klassenelemente mit Zugriffsrecht protected
public :
    // Klassenelemente mit Zugriffsrecht public
}
```

Bei der Definition einer Methode muß der Name der Klasse immer mit angegeben werden,da Methoden verschiedener Klassen die gleichen Namen tragen dürfen.

```
<gelieferter Typ> <classname>::<Elementfunktionname> ( <Parameter> )
{
        <body>
}
```

Beispiel:

```
#include <iostream.h>
class  date {
private :       int  month,day,year;
public :        void set(int, int, int);
                void get(int, int, int);
                void next();
                void print();
};

void date::set(int, int, int)  {  }
void date::get(int*, int*, int* )  {   }
void date::next()  {   }
void date::print()  {   }

date today;
date my_birthday;

int  main( void )
{       my_birthday.set(30, 12, 1950);
        today.set( 18, 1, 1996);
        my_birthday.print();
        today.next();
        //  ...
}
```

Zugriffsrechte

Bei einer Klasse kann der Zugriff auf die Elemente durch die Zugriffsrechte gesteuert werden. Hierbei unterscheidet man 3 Arten von Zugriffsrechten.

private: Ein Element, welches als private deklariert ist, kann nur von Elementfunktionen und Freunden (friends) der Klasse verwendet werden, in der es deklariert ist.

protected: ein Element, welches als protected deklariert ist, verhält sich gegenüber einer abgeleiteten Klasse wie ein public-Element und gegenüber dem restlichen Programm wie ein privates Element.

public: Ein Element, welches als public deklariert ist, ist für das gesamte restliche Programm zugänglich.

Konstruktor

Eine Funktion, die explizit für das Initialisieren eines Objektes zuständig ist , heißt Konstruktor (constructor). Konstruktor trägt denselben Namen wie die Klasse selbst.

```
class date  {
        // ...
        date(int,int,int);
};
```

Man kann auch mehrere Konstruktoren definieren

```
date (int,int,int);           //  Tag, Monat, Jahr
date (const   char*);     //  Datum in String-Darstellung
```

Destroktor

Der Destruktor sorgt dafür, daß die Objekte des Typs ordnungsgemäß gelöscht werden. Der Name des Destruktors für die Klasse `X` ist `~ X`.

```
class string  {
    int  size;
    char   *s;
public :
    string (int s);
    string ( )  {  delete s;  }
};
```

Friends

Eine Funktion ist eine friend-Funktion einer Klasse sobald sie in der Klassendeklaration als friend deklariert ist.
Beispiel: Multiplikation einer Matrix mit einem Vector

```
class   vector  {
        float   v[4];
        friend  vector  multiply(const  matrix&, const  vector&);
};
class   matrix  {
        vector  v[4];
        friend  vector  multiply(const  matrix&, const  vector&);
};
```

Die multiply( )-Funktion kann direkt auf die Datenelemente des Vectors und der Matrix zugreifen:

```
vector  multiply(const  matrix&  m, const  vector&  v)
{       vector r;
        for  (int  i=0; i<3; i++)  {   //  r[i] = m[i] * v;
                r.v[i] = 0;
                for  (int  j=0; j<3; j++)
```

```
                        r.v[i] += m.v[i].v[j] * v.v[j] ;
                }
        return r;
}
```

1.10.3 Templates

Containerklassen: Klassen, die Objekte irgendeiner Klasse enthalten, bilden eine Gruppe von Klassen (Arrays,Listen,Mengen...). Man kann einen Stack für Elemente beliebigen Typs definieren:

```
template <class  T>
    class   stack   {
        T*    v;
        T*     p;
        int     sz;
    public :
        stack(int  s)   {  v=p=new   T[sz=s];  }
        (stack( )        {  delete[]     v;}
        void  push(T  a)    {   *p++=a;   }
        T  pop( )   {  return *--p ;}
        int size( )   const    {return  p-v ;}
    };
```

Das Template <`class` T >-Präfix legt fest, daß ein Template deklariert wird und daß in der Deklaration ein Argument T anzugeben ist.

```
zum Beispiel :
        stack<char>   sc(100);     //   ein char-stack
```

1.10.4 Vererbung

Abgeleitete Klassen

Abgeleitete Klassen dienen dem Zweck, Gemeinsamkeiten zwischen Klassen zu formulieren und is-a-Relation darzustellen. Beispiel:

```
class   employee   {
        char *     name;
        //  ...
public :
        void   print( )   const;
};
class   manager  :  public employee  {
        short   level ;
        employee   *group;
        //   ...
public :
        manager ( char  *n, int  l, int d);
        void  print( )  const;
        //   ...
};
```

`employee` ist die Basisklasse (base class) `manager` ist vom employee abgeleitet (derived).

- die Element-Funktion einer abgeleiteten Klasse kann auf die Elemente der Basisklasse eines Objekts zugreifen.

- Elemente der abgeleiteten Klasse haben keinen Zugriff auf die privaten Elemente der Basisklasse. Eine abgeleitete Klasse kann wiederum als Basisklasse dienen.

Polymorphie

Polymorphie bezieht sich auf die Fähigkeit einer Referenz zur Laufzeit auf Instanzen verschiedener Klassen zu zeigen. Eine Referenz einer bestimmten Klasse kann nur auf Objekte dieser Klasse oder auf Objekte der abgeleiteten Klassen zeigen.

Virtuelle Funktionen

Virtuelle Funktionen erlauben dem Programmierer, in der Basisklasse Funktionen zu deklarieren, die in den abgeleiteten Klassen redefiniert werden können. Der Compiler garantiert dabei die exakte Übereinstimmung zwischen Objekten und den auf sie angewandten Funktionen.

Beispiel :

```
class employee   {
        char *   name;
        short     departement;
        //   ...
        employee*     next;
        static    employee*    list;
public :
        employee ( char*    n, int    d)
        //   ...
        static   void   print_list ( );
        virtual   void  print( )   const;
};

void   employee:: print( )   const
{
    cout <<name<<'\t'<<department<<'\n'
}
```

Das Schlüßelwort `virtual` zeigt an, daß von der Funktion `print( )` für verschiedene abgeleitete Klassen unterschiedliche Version existieren können. Eine virtuelle Funktion muß für diejenige Klasse, in der sie das erste Mal deklariert wurde, auch definiert werden.

```
class   manager : public   employee {
        employee*   group;
        short            level;
        //   ...
public :
        manager ( char*  n, int  l, int  d );
        //   ...
        void  print( )   const;
};
```

Die `employee-Liste` kann folgendermaßen ausgegeben werden:

```
void   employee :: print_list( )
{
    for (employee*   p=list;p;p=p->next )
        p->print( );
}
```

Templatevererbung

Templates sind ebenfalls Klassen, so daß auf sie auch die Vererbung anwendbar ist. Aus dem Listen-Template kann man durch Vererbung eine aufsteigend sortierte Liste ableiten. Das Template **`linlist`** enthält keine neuen Daten, sondern neue Methoden. Also sind Konstruktor und Destruktor leer.

```
template < class T >  class  linlist
:  public  list<T>
{
public :
     linlist( );
     (linlist( );
     void  insertelem( T  newelem);
     void  removeelem( T  oldelem);
     void  findelem( T  searchelem);
};

template < class  T >  linlist<T> :: linlist( void )   { }
template < class  T >  linlist<T> :: (linlist( void )   { }
template < class  T >  linlist<T> :: insertelem( T  newelem )
{
       //   ...
}

template < class  T >  linlist<T> :: removeelem( T  oldelem )
{
       //   ...
}

template < class  T >  linlist<T> :: findelem( T  searchelem )
{
       //   ...
}
```

Hier wurde `list<T>` public vererbt.

1.11 Sniff+

Verfasser: Mario Ellebrecht

1.11.1 Einleitung

Sniff+ ist eine *integrierte Software-Entwicklungsumgebung*, d.h. ein Tool zur Unterstützung der "Programmierung im Kleinen" für verschiedene objektorientierte Sprachen. Es soll dazu dienen, die verteilte Projektarbeit durch ein einheitliches Interface zu verschiedenen *Werkzeugen* zu unterstützen.

In diesem Referat soll aufgezeigt werden, wie Sniff+ der Projektgruppe konkret von Nutzen sein kann, ohne in eine detaillierte Beschreibung der einzelnen *Tools* zu verfallen.

In der Einleitung sollen zunächst kurz die Anforderungen an eine Entwicklungsumgebung dargestellt werden, wie sie in der Projektarbeit gegeben sind. Anschliessend wird angerissen, bei welchen Bereichen Sniff+ die Arbeiten unterstützen könnte.

Im nächsten Abschnitt wird dann näher auf die grundlegenden Konzepte und die Funktionalität von Sniff+ eingegangen. Dabei stehen die grundlegenden Strukturen eines Projekts sowie die wichtigen *Browsing*-Fähigkeiten im Vordergrund.

Danach soll versucht werden zu erörtern, welcher Aufwand durch die Verwendung von Sniff+ zusätzlich entsteht und welcher Aufwand eingespart werden kann. Es wird der Blick auf mögliche Alternativen gelenkt, um schließlich mit einerDarstellung von Problemfeldern und Erfahrungswerten zur kritischen Diskussion des Einsatzes von Sniff anzuregen.

Integrierte Enwicklungsumgebung Eine Software-Entwicklungsumgebung soll in der Projektarbeit dazu dienen, in möglichst vielen Phasen eine einheitliche Arbeitsgrundlage für alle Teammitglieder zu schaffen, die Arbeit von lästigen Routineaufgaben zu entlasten und den Überblick über das Projekt zu gewährleisten.

Die Teamarbeit soll koordiniert werden: zum einen müssen im verteilten Gesamtprojekt die Aufgabenbereiche der einzelnen Gruppen festgelegt werden und auch während der Arbeit transparent bleiben.

Zum anderen müssen im Rahmen dieser Einzelbereiche Dateizugriff und Objektstruktur organisiert werden. Diese Festlegungen müssen auch im Gesamtprojekt sichtbar sein, um Fehler und Mißverständnisse zu vermeiden.

Der Zugriff auf die Tools, die diese Anforderungen erfüllen, sollte für alle gleich sein. Dazu muß die Umgebung offen, d.h. mit evtl. benötigten besonderen Tools erweiterbar sein.

Objektorientierte Abstraktion Sniff+ ist zu einem Einsatz ab der Implementationsphase vorgesehen und unterstützt das objektorientierte Erstellen, das Übersetzen, die Analyse und das Testen von Software. Rahmen zur Erstellung von Dokumentation werden zur Verfügung gestellt.

Die Stärke von Sniff+ ist das *Browsing*: Objektstrukturen wie Klassenhierarchie, Klassenmember, Overriding etc. können in verschiedenen Formen dargestellt werden. Dabei abstrahiert Sniff+ sowohl von der Organisation der Objekte in den Dateien als auch von der spezifischen Sprache. Die Tools zum Browsing gehen stattdessen von allgemeineren objektorientierten Konzepten wie Klassen aus und ermöglichen so von Anfang an das Konzentrieren auf die Objektstrukturen. Auch kann man mit hierarchischen Darstellungen das Übereinstimmen der Implementation mit der Planung verfolgen.

Daneben stehen auch Features zur dateibasierten Analyse zur Verfügung.

1.11.2 Konzepte und Features

Projekte und Workspaces *Projekte* sind in Sniff+ hierarchisch organisiert: Das Gesamtprojekt besteht aus mehreren *Unterprojekten*, die von einem kleineren Team bearbeitet werden. Die Projekte selbst bestehen aus verschiedenen *Workspaces*.

Es gibt einen *Shared Source Workspace* für die Source-Dateien des gesamten Projekts, an denen die Entwickler arbeiten. Die daraus beim Übersetzen gewonnenen Objektdateien werden in einem *Shared Object Workspace* abgelegt und dort zu einer lauffähigen Version gelinkt. Jedes Teammitglied hat seinen eigenen *Private Workspace*, in den er Source-Dateien aus dem Shared Source Workspace kopiert und dort bearbeitet.

Zu festgelegten Zeiten können dann Änderungen der einzelnen Teams wieder in den Shared Source Workspace übernommen und anschliessend übersetzt werden. Für jeden Workspace ist ein Administrator zuständig.

Dateiverwaltung Zu einem Projekt gehören alle betroffenen Dateien: Projektbeschreibungen, Source-Files aller Sprachen, Objektdateien, lauffähige Programme und Daten wie Bilder etc. Jede dieser Dateien wird von der *Versionsverwaltung* kontrolliert. So ist es möglich, Änderungen von verschiedenen Teammitgliedern nachzuvollziehen und auch später noch frühere Versionen des gesamten Projektes zu rekonstruieren.

Konkurrierende Zugriffe werden durch *Locking* organisiert. Damit "*checkt*" man eine Datei aus dem Shared Source Workspace aus und editiert sie im Private Workspace. Währenddessen können die anderen Teammitglieder sehen, wer die Datei gerade bearbeitet. Nach den Änderungen wird sie wieder eingecheckt und steht für alle zur Verfügung.

Tools Sniff+ als integrierte Umgebung ist eine Oberfläche für verschiedene Werkzeuge, von denen einige in ihrer Funktionalität schon kurz angesprochen wurden. Diese Tools sind z.T. Bestandteil der Sniff-Distribution, es können aber auch fremde Tools eingebunden werden. So liefert Sniff+ keinen Compiler oder Debugger mit. Hier wird Standardsoftware der entsprechenden Plattform genutzt.

Der Vorteil von Sniff+ besteht auch in der guten Zugänglichkeit der Tools. Unterschiedliche Tools, die Information gemeinsam nutzen, sind meist untereinander verknüpft. So kann man z.B. von der Klassenhierarchie direkt zum Source-Code einer bestimmten Klasse wechseln.

Die bereits angesprochene Hierarchie von Projekten ist in allen Tools von Sniff+ sichtbar und stellt ein zentrales Moment zur Strukturierung dar.

Browsing Wie schon erwähnt, stellen die Browsing-Fähigkeiten von Sniff+ ein zentrales Element dar. Es sind hauptsächlich die Browsing-Tools, die einen schnellen und strukturierten Überblick über das Projekt ermöglichen. Von ihnen ausgehend kann man dann weiter nach bestimmten Informationen suchen. Änderungen, die in einem Tool durchgeführt werden, sind ohne neues Übersetzen sofort auch in anderen Sichten auf diese Information sichtbar. Ohne konkret die Tools beschreiben zu wollen, sollen im folgenden die wichtigsten Funktionalitäten dargestellt werden, damit der Nutzen sichtbar wird.

Klassenhierarchie - "is-a" Einen Überblick verschafft uns die Klassenhierarchie. Sie stellt in einem Baum die Vererbungshierarchie der Klassen dar (*is-aBeziehung*). Dabei kann man die Menge der angezeigten Klassen einschränken, z.B. auf ein Unterprojekt, einen Teilbaum oder einen Pfad zur gewählten Klasse.

Dabei werden virtuelle, konkrete und eigene Klassen unterschiedlich dargestellt, sodaß diese Merkmale auf einen Blick erkennbar sind.

Klassenmember - "has-a" Will man statt eines Überblicks mehr Information zu einer bestimmten Klasse sehen, so wählt man die Anzeige der Klassenmember (*"has-a"-Beziehung*). Hier sieht man alle *Symbole* einer Klasse oder Vererbungslinie. Symbole sind für Sniff+ z.B. Methoden, Variablen oder Typdeklarationen.

Zu den blossen Namen der Symbole lassen sich noch eine Fülle weiterer Informationen darstellen. Wichtig ist neben der *Signatur*, Typ und Zugriffsrechten eines Symbols dessen Entwicklung in der Klassenhierarchie. So wird u.a. dargestellt, ob ein Symbol in einer Kind-Klasse umdefiniert wird (*Overriding*), oder ob es selbst ein Symbol aus einer übergeordneten Klasse redefiniert.

Bottom-up Browsing Beide schon genannte Browsing-Arten gehen *Top-Down* vor. Aus einer Darstellung aller Klassen wählt man eine bestimmte Klasse aus, dort wieder einen Member usw. Dies ist für einen Überblick sinnvoll. Oft sucht man aber ausgehend von einem Symbol im Source-Code z.B. nach dessen *Deklaration* oder anderen Symbolen der gleichen Klasse. Hier geht man also *Bottom-Up* vor.

Sniff+ unterstützt dieses Vorgehen auf unterschiedliche Weise. Zum einen kann man sich wie schon erwähnt die Klasse zu einem Symbol ansehen und von dort aus z.B. die anderen Symbole der Klasse oder übergeordneter Klassen.

Zum anderen kann man jedoch auch nach Symbolen in allen Klassen suchen. Will man z.B. wissen, wo das gewählte Symbol noch als Komponente auftaucht, oder welche Methoden von einer bestimmten Funktion aufgerufen werden, so stellt Sniff+ diese Information ähnlich der Klassenhierarchie als Baum dar, von dem aus man dann zu weiteren Informationen weiterverzweigen kann.

Außerdem erlaubt Sniff+ eine Suche nach Symbolnamen in beliebigen Teilmengen des Source-Codes. Dies ist von Nutzen, wenn man die weitere Verwendung einer Variable wissen oder Namenskonflikte vermeiden will.

1.11.3 Bewertung

zusätzlicher und gesparter Aufwand Der Aufwand, der durch die Verwendung von Sniff+ einsparen läßt besteht vor allem aus den folgenden Punkten:

- schnelleres Verständnis des Codes auch anderer Teammitglieder hilft Vermeidung von Fehlern
- strukturierter Überblick wird recht gut unterstützt und ermöglicht so Erkennung von Fehlentwicklungen
- Änderungen des Codes sind gut organisiert und lassen sich zurückverfolgen
- Die Arbeit mit Dateien wird öfter durch Abstraktion erleichtert
- Spezielle Eigenschaften einzelner Tools werden so weit wie möglich voreingestellt und später für den Nutzer verdeckt
- die Dokumentation wird teilweise automatisch erstellt

Diese Vorteile sind jedoch Ergebnis von Aufwand, der zusätzlich zur Arbeit am Projekt geleistet werden muß:

- die Projektstruktur muß vorher festgelegt werden und sollte sich möglichst nicht ändern
- viele Eigenschaften der einzelnen Tools müssen beim Anlegen des Projekts gut überlegt und eingestellt werden, Änderungen sind schwierig
- Arbeit an Dateien des gesamten Projekts muß von einem Administrator zusätzlich erledigt werden

- Umgang mit der Oberfläche muß erlernt werden
- auftauchende schwere Probleme sind oft nur durch Neueinrichtung des Projekts zu beheben
- Überschneidung in der Funktionalität mit anderen Tools erfordern u.U. doppelten Aufwand für bestimmte Bereiche

Probleme und Erfahrungswerte Wie jedes Softwareprodukt wirft auch Sniff+ Probleme auf:

- Die am LS10 verwendete Version 2.2 ist noch neu und stürzt manchmal ab
- keine gute Integration mit anderen Phasen, z.B. Planung
- Änderung der einmal festgelegten Projekteinrichtung ist schwierig
- für die Verwendung mit anderen Sprachen als C++, z.B. Java liegen keine Erfahrungswerte vor
- (unbeabsichtigte) Änderungen an den Dateien ausserhalb von Sniff können schwerwiegende Probleme aufwerfen

Erfahrungswert anderer Projektgruppen ist z.B., daß jedes Projekt wegen der auftauchenden Probleme zweimal installiert werden muß.

Dennoch wird Sniff oft bei Projektgruppen, aber auch z.B. bei CERN und Softwarefirmen eingesetzt.

Alternativen Das liegt wohl auch an den mangelnden Alternativen. Es gibt Entwicklungsumgebungen mit ähnlicher Funktionalität wie Sniff+, z.B. Rose, die aber auch ähnliche Probleme haben und zusätzlich angeschafft werden müssten.

Die Funktionalität von Sniff+ überdeckt sich teilweise mit anderen Projektmanagement-Tools. So könnte es z.B. bei Verwendung von BSCW o.ä. dazu kommen, daß man bestimmten Aufwand doppelt leisten muß. Hier könnte man auch einfache *Development Kits* verwenden, die nur Compiler, Debugger und einzelne Anzeigewerkzeuge enthalten (z.B. Java DK) und die Projektverwaltung mit anderen Tools erledigen. Dabei fällt natürlich der Vorteil der Integration weg und es entsteht zusätzlicher Einarbeitungsaufwand.

1.11.4 Referenzen

- TakeFive Software

  `http://www.takefive.com/`
- Sniff+ 2.2 Online Documentation

  `file:/home/software/Sniff/doc/online/sniff.htm`
- CERN Software Development - Known products

  `http://www.cern.ch/PTTOOLS/SoftKnow.html`
- Object-Oriented Resources

  `http://aaimzb.mathematik.uni-mainz.de/Personal/Mitarbeiter/OOStuff.html`
- Silicon Graphics applications & solutions directory

  `http://193.242.86.10/Products/appsdirectory.dir/SolutionIXApplication_Development.html`

1.12 Architektur von föderativen Datenbanksystemen

Verfasser: Dilber Yavuz

1.12.1 Einführung

Datenbank, Datenbanksystem, Datenbankmanagementsystem, Föderative Datenbanksysteme

In den meisten Unternehmen und Organisationen ist der Einsatz eines DBMS für die Informationsverarbeitung unumgänglich. Man denke etwa an Banken, Versicherungen, Flugunternehmen und Universitätsverwaltungen. Die Daten werden innerhalb von Datenbanksystemen (DBS) verwaltet. Das DBS besteht dabei aus der eigentlichen Datenbank (DB) und dem Datenbankmanagementsystem (DBMS). Die Datenbank enthält eine Menge von Informationsdarstellungen aus einem bestimmten, abgegrenzten Informationsbereich. Das DBMS verwaltet diese Daten und führt sämtliche Zugriffe darauf aus. Für den Zugriff auf die DB werden in der Regel deskriptive und ausdrucksstarke Anfragesprachen wie SQL benutzt. Ein föderatives DBS (FDBS) ist eine Menge von zusammenarbeitenden, aber autonomen Datenbanksystemen. Das FDBMS liefert eine kontrollierte und koordinierte Manipulation von Datenbanksystemen [Rah94].

Merkmale von FDBS

Eine der wesentlichen Merkmale eines FDBS ist, daß die lokalen Datenbanksysteme ihre internen Ausführungen durchführen und gleichzeitig in einer Föderation beteiligt seinkönnen. Das FDBS sollte die Anforderungen wie Verteilungstransparenz, Verbergen der Heterogenität sowie die Autonomie erfüllen [Rah94].

Verteilungstransparenz

Die verteilte Datenbankverarbeitung sollte gegenüber Anwendungen und Benutzern unsichtbar bleiben. Diese Forderung ist wesentlich für die Einfachheit der Anwendungserstellung und -wartung, da so z. B. Änderungen in der Verteilung von Daten ohne Rückwirkung auf die Transaktionsprogramme bleiben.

Heterogenität

Unterschiedliche Formen der Knotenautonomie führen meist zu unterschiedlichen Arten der Heterogenität, welche die Nutzung der einzelnen Datenbanken erschweren. Man unterscheidet im wesentlichen folgende Arten der Heterogenität:

Heterogenität bezüglich der beteiligten DBMS Da verschiedene Benutzer unterschiedliche Bedürfnisse haben, können sie auch unterschiedliche DBMS auswählen. Aufgrund dieser Unabhängigkeit können sich die Lokalen DBS (LDBS) hinsichtlich Hersteller, Version, Datenmodell, Anfragesprache sowie interner Realisierung unterscheiden.

Heterogenität in der Ablaufumgebung Die Benutzer sind auch unabhängig bei der Wahl der Ablaufumgebung. Deshalb können die einzelnen Rechner Unterschiede in Hardware, Betriebssystemen, TP-Monitoren (Transaction Processing Monitor) sowie Kommunikationsprotokollen aufweisen.

Heterogenität im DB-Inhalt (semantische Heterogenität) Die semantische Heterogenität ist die Folge der Entwurfsautonomie, welche den unabhängigen Entwurf des logischen (und physischen) Aufbaus der einzelnen Datenbanken gestattet. Sie kann sich in vielfacher Weise zeigen, insbesondere bei der Benennung und Repräsentation von DB-Objekten. Gleiche oder verwandte Daten können verschiedene Namen erhalten und in unterschiedlichster Weise repräsentiert werden.

Autonomie

Der Zugriff auf mehrere Datenbanken innerhalb einer Transaktion setzt ein Mindestmaß an Kooperationsbereitschaft der beteiligten Knoten und ihrer DBMS voraus. Das führt zu einer reduzierten Unabhängigkeit oder Autonomie der Rechner. Man unterscheidet folgende Arten der Knotenautonomie:

- Entwurfsautonomie,
- Ausführungsautonomie,
- Kooperationsautonomie

Klassifikation von Mehrrechner-DBS

Der Begriff *Mehrrechner-Datenbanksysteme* faßt sämtliche Architekturen zusammen, bei denen mehrere Prozessoren oder DBMS-Instanzen an der Verarbeitung von DB-Operationen beteiligt sind. Dabei ist eine Kooperation der Prozessoren bzw. DBMS bezüglich der DB-Verarbeitung bestimmter Anwendungen vorzunehmen, um den Fall voneinander isoliert arbeitender DBMS oder Prozessoren auszuschließen.

Klassifikationsmerkmale

Eine erste Grobklassifikation von Mehrrechner-DBS ergibt sich durch die Verwendung der drei Kriterien Externspeicherzuordnung, räumliche Verteilung sowie Rechnerkopplung [Rah94].

Externspeicherzuordnung Hier unterscheidet man zwischen partitioniertem und gemeinsamem Zugriff. Beim *partitioniertem Zugriff* erfolgt eine Partitionierung der Externspeicher unter den Prozessoren bzw. Rechnern. Im Falle von Magnetplatten als Externspeicher ist jedes Plattenlaufwerk und die darauf gespeicherten Daten genau einem Rechner zugeordnet. Beim *gemeinsamen Zugriff* hat der Prozessor direkten Zugriff auf alle Platten und damit auf die gesamte DB. Damit entfällt zwar die Notwendigkeit einer verteilten Transaktionsverarbeitung, jedoch sind ggf. Synchronisationsmaßnahmen für den Externspeicherzugriff vorzusehen.

Räumliche Anordnung Hier unterscheidet man zwischen lokaler und ortsverteilter Rechneranordnung. *Lokal verteilte Systeme* ermöglichen eine wesentlich leistungsfähigere Interprozessor-Kommunikation und sind robuster gegenüber Fehlern im Kommunikationssystem. Der entscheidende Vorteil lokal verteilter Mehrrechner-DBS ist aber die effiziente Kommunikation. Dies ist vor allem für die parallele DB-Verarbeitung von Bedeutung. Lokal verteilte Mehrrechner-DBS weisen daneben Vorteile hinsichtlich der Administration gegenüber geographisch verteilten Systemen auf. Auf der anderen Seite kann nur mit ortsveteilten MDBS eine Anpassung an verteilte Organisationsstrukturen erfolgen, wie es in großen Unternehmen wünschenswert ist. Ortsverteilte Konfigurationen bieten auch eine größere Fehlertoleranz gegenüber 'Katastrophen'.

Rechnerkopplung Die bekanntesten Kopplungsansätze sind die enge und die lose Rechnerkopplung. Bei der *engen Kopplung* teilen sich die Prozessoren einen gemeinsamen Hauptspeicher. Software wie Betriebssystem, DBMS oder Anwendungsprogramme liegen nur in einer Kopie vor. Die Nutzung solcher Systeme zur DB-Verarbeitung bezeichnet man als Multi-Prozessor-DBS. Bei der *losen Kopplung* kooperieren mehrere unabhängige Rechner, die jeweils einen eigenen Hauptspeicher besitzen sowie private Software-Kopien. Dies ermöglicht eine weit bessere Fehlerisolation als bei enger Kopplung. Das Fehlen eines gemeinsamen Hauptspeichers führt auch zu einer besseren Erweiterbarkeit. Hauptnachteil ist die aufwendige Kommunikation.

Typen von Mehrrechner-Datenbanksystemen

Die vorgestellten Klassifikationsmerkmale ergeben die Unterscheidung zwischen drei verbreiteten Typen von Mehrrechner-DBS:

Shared-Everything: Die DB-Verarbeitung erfolgt durch ein DBMS auf einem Multiprozessor.

Shared-Nothing: Die DB-Verarbeitung erfolgt durch mehrere, i. a. lose gekoppelte Rechner, auf denen jeweils ein DBMS abläuft. Die Externspeicher sind unter den Rechnern partitioniert.

Shared-Disk: Wie bei Shared-Nothing liegt eine Menge von Verarbeitungsrechnern mit je einem DBVS vor, wobei jeder Rechner ein Multiprozessor sein kann. Hier liegt eine gemeinsame Externspeicherzuordnung vor.

Von diesen drei vorgestellten Typen ist das Multiprozessor-DBS (Shared-Everything) am einfachsten realisierbar, da das Betriebssystem die Verteilung weitgehend verbirgt. Die meisten kommerziellen DBS unterstützen den Shared-Everything-Ansatz, d. h. sie können Multiprozessoren zur DB-Verarbeitung nutzen.

Integrierte vs. föderative Mehrrechner-DBS

Im folgenden werden Mehrrechner-DBS in integrierte und föderative sowie in homogene und heterogene Mehrrechner-DBS klassifiziert. Diese Merkmale sind nur für Architekturen mit mehreren DBMS relevant, also für Shared-Disk und Shared-Nothing. Merkmal *integrierter Mehrrechner-DBS* ist, daß sich alle Rechner eine gemeinsame Datenbank teilen, deren Aufbau durch ein einziges konzeptionelles Schema beschrieben ist. Die DBMS sind in allen Rechnern identisch, d. h. *homogen*. Ein Beispiel für integrierte Mehrrechner-DBS sind die verteilten DBS. *Verteilte DBS* stellen geographisch verteilte, integrierte Shared-Nothing-Mehrrechner-DBS dar. Parallele DBS vom Typ Shared-Disk und Shared-Nothing repräsentieren ebenfalls integrierte Mehrrechner-DBS. Föderative Mehrrechner-DBS streben nach größerer Knotenautonomie, wobei die DBMS entweder homogen oder heterogen sein können. Das föderative DBS ist dadurch gekennzeichnet, daß jeder Rechner eine eigene DB verwaltet, die durch ein lokales konzeptionelles Schema beschrieben ist. Idealerweise unterstützen föderative Mehrrechner-DBS ein einheitliches Datenmodell bzw. bieten eine gemeinsame Anfragesprache an [SL90].

1.12.2 Referenzarchitekturen

Eine Referenzarchitektur dient der Aufklärung von verschiedenen Problemen und Auswahlmöglichkeiten bezüglich der DBS. Es liefert die Struktur zum Verständnis, der Klassifizierung und zum Vereinigen von verschiedenen architektonischen Möglichkeiten zu Entwicklung von föderativen DBS.

Komponenten der Referenzarchitektur

Die 3-Ebenen-Architektur

Das zentrale Ziel einer Datenbank, die Integration von Daten und die Datenunabhängigkeit der Anwendungsprogramme setzt voraus, daß sich alle Anwendungsprogramme auf eine einheitliche Beschreibung der Daten einigen. Um diese Aufgabe zu erfüllen, wurde eine 3-Ebenen-Architektur von ANSI/SPARC vorgeschlagen. Die Beschreibung der Daten erfolgt auf drei verschiedenen Ebenen, die jeweils durch eine andere Sicht geprägt sind. Zur Formalisierung einer Sichtweise dienen Schemata. Ein Schema ist eine in einer Datenbeschreibungssprache abgefaßte Definition der in einer Datenbank zugelassenen Datenstrukturen.

Die *interne Ebene* liegt dem physikalischen Speicher am nächsten. Sie ist von diesem zu unterscheiden, das sie die physisch gespeicherten Daten nicht als Pages oder Blöcke, sondern als interne Records betrachtet. Diese Sicht der Daten wird im internen Schema festgelegt. Das interne Schema enthält Informationen über die Art und den Aufbau der Datenstrukturen. Auf der *konzeptionellen Ebene* wird die logische Gesamtsicht aller Daten in der Datenbank und ihrer Beziehungen untereinander im konzeptionellen Schema repräsentiert. Dazu verwendet man die sprachlichen Mittel eines Datenmodells (z. B. 'Tabellenformat' im relationalen Modell). Die *externe Ebene* umfaßt alle individuellen Sichten der einzelnen Benutzer oder Benutzergruppen (also der Anwendungsprogrammierer sowie der Dialog-Benutzer) auf die Datenbank. Diese Sichten (Views) werden jeweils in einem eigenen externen Schema beschrieben, welches genau den Ausschnitt der konzeptionellen Sicht enthält, den der Benutzer sehen möchte bzw. sehen darf [Rah94].

Die 5-Ebenen-Architektur für FDBS

Die Architektur von FDBS basiert i. a. auf einem Zusatzebenen-Ansatz, da die Lokalen DBS weitgehend unverändert bleiben sollen. Aufgabe der Zusatzschicht ist es, in der gemeinsamen Anfragesprache formulierte globale Anfragen bzw. Transaktionen zu bearbeiten. Bei der Realisierung von föderativen DBS lassen sich grob zwei Klassen unterscheiden, die eng bzw. lose gekoppelte FDBS bezeichnet werden können. *Eng gekoppelte FDBS* streben eine Verteilungstransparenz an, in dem Benutzern gegenüber ein globales konzeptionelles Schema bzw. föderatives Schema angeboten wird. Dies setzt eine Schemaintegration voraus, während der auch eine Behandlung von semantischer Heterogenität erfolgt. Bei *lose gekoppelten FDBS* bleibt die Unterscheidung mehrerer Datenbanken für den Benutzer sichtbar. Er bekommt Hilfsmittel (insbesondere eine Anfragesprache) zur Verfügung gestellt, um in einfacher und mächtiger Weise auf die verschiedenen Datenbanken zuzugreifen.

Zur Beschreibung von FDBS ist die 3-Ebenen-Architektur von ANSI/SPARC nicht ausreichend, da die Merkmale Verteilungstransparenz, Heterogenität und Autonomie für FDBS nicht wie erforderlich unterstützt werden. Deshalb wurde ein erweitertes Modell mit fünf Ebenen für FDBS entwickelt. Diese Architektur hat, wie bei zentralisierten und Verteilten DBS ein internes sowie ein konzeptionelles Schema. Ebenso greifen Benutzer über externe Schemata auf die Datenbank zu. Neu hinzugekommen sind folgende Schematypen: Komponenten-, Export- sowie föderative Schemata.

Die *Komponenten-Schemata* basieren auf einem gemeinsamen Datenmodell, dem sog. Common Data Model (CDM). Jedes konzeptionelle Schema eines anderen Datenmodells wird durch eine Schema-Translation in ein äquivalentes konzeptionelles Schema des CDM transformiert.

Nicht alle Daten des DBS sind für die Föderation und dessen Benutzer verfügbar. Die *Export-Schemata* werden auf dem Komponenten-Schema, die für die FDBS verfügbar ist, definiert. Bezüglich der Benutzung durch Föderationsbenutzer können die Export-Schemata eine Zugriffskontrollinformation beinhalten. Die Export-Schemata dienen damit zur Unterstützung der DBS-Autonomie, insbesondere der Kooperationsautonomie.

Ein *föderatives Schema* umfaßt die Schemaangaben mehrerer Export-Schemata. Also enthält es die Information über Verteilte Daten, die als Export-Schema generiert sind. Ein Konstruktionsprozessor formt die Befehle der föderativen Schema in Befehle der Export-Schema. Diese beiden Schemata unterstützen damit die Verteilungstransparenz von FDBS.

Mit *externen Schemata* kann eine Einschränkungung der Daten und der zulässigen Operationen für einzelne Benutzer erreicht werden [SL90].

1.13 Transaktionen

Verfasser: Betül Ilikli

1.13.1 Einführung

Der Datenbankzugriff erfolgt innerhalb von Transaktionen, welche aus einer oder mehreren Datenbank-Operationen der jeweiligen Anfragesprache bestehen.

ACID-Transaktionseigenschaften

Für Transaktionen gelten die folgenden**ACID-Eigenschaften:**

1. Atomarität (Automicity) Änderungen einer Transaktion werden entweder vollständig oder garnicht ausgeführt. Wenn also während der Transaktionsausführung Fehler aufgetreten sind, dann bleibt dies ohne Auswirkung auf die Datenbank, da entsprechende Ausführung vollständig zurückgesetzt wird. Die nennt man die Alles-oder-Nichts-Eigenschaft von Transaktionen.

2. Konsistenz (Consistence) Transaktionen sind konsistenterhaltend, d.h. bei Beginn und Ende einer Transaktion sind die Integritätsbedingungen erfüllt.

3. Isolation Datenbanksysteme unterstützen typischerweise eine große Anzahl von Benutzern, die gleichzeitig auf die Datenbank zugreifen können. Trotz dieses Mehrbenutzerbetriebes wird garantiert, daß dadurch keine unerwünschten Nebenbedingungen eintreten (z.B. gegenseitiges Überschreiben derselben Datenbankobjekte).

4. Dauerhaftigkeit (Durability) Die Dauerhaftigkeit von erfolgreich beendeten Transaktionen wird garantiert. Dies bedeutet, daß Änderungen dieser Transaktionen alle erwarteten Fehler (insbesondere Rechnerausfälle, Externspeicherfehler und Nachrichtenverlust) überleben.

Struktur verteilter Transaktionen

Eine Transaktion besteht aus einer Folge von DB-Operationen, die durch eine **BOT**-Operation (**B**egin **o**f**T**ransaction) und eine Commit- oder **EOT**-Operation (**E**nd **o**f **T**ransaction) geklammert sind. In Verteilten Datenbanksystemen (DBS) können bei der Ausführung einer Transaktion ein oder mehrere Knoten beteiligt sein. Der Knoten, an dem die Transaktion gestartet wurde, wird als Koodinator-Knoten bezeichnet. Falls die Transaktion vollständig an dem Koordinator-Knoten stattfindet, dann spricht man von einer **lokalen Transaktion**, anderenfalls von einer **globalen** oder **verteilten Transaktion**. An jedem bei der Ausführung einer globalen Transaktion T beteiligten Knoten wird eine *Teil-* oder *Sub-Transaktion* ausgeführt, die alle DB-Operationen bzw. Teiloperationen von T umfaßt, die an dem Knoten bearbeitet werden. Die an dem Koordinator-Knoten ausgeführte Sub-Transaktion wird auch als *Primär-Transaktion* bezeichnet. Zwischen Primär- und Sub-Transaktionen besteht eine hierarchische Aufrufstruktur, die im Transaktionsbaum reflektiert ist. Darin bildet die Primär-Transaktion die Wurzel; jeder andere Knoten im Baum entspricht einer Sub-Transaktion.

Commit-Behandlung

Um zu einer global einheitlichen Entscheidung über Commit oder Abort zu kommen, müssen alle Agenten in den Zustand übergehen, in dem sie sowohl Änderungen persistent gespeichert haben als auch in der Lage sind, die entsprechende Sub-Transaktion atomar zurückzusetzen. Nachdem der Benutzer mit der EOT-Anweisung das Ende der Anwendungstransaktion signalisiert, werden zwei Phasen durchlaufen.

1.Phase Es werden sämtliche Datenbankänderungen sowie ein sogenannter Commit-Satz aufdie Log-Datei geschrieben. Wurde diese Phase erfolgreich abgeschlossen, d.h.,der Commit-Satz auf die Log-Datei geschrieben, ist das erfolgreiche Ende der Transaktion garantiert.

2.Phase Die Änderungen der erfolgreich beendeten Transaktionen werden anderen Transaktionen sichtbar gemacht, z.B. durch Freigabe der Sperren.

Kommunikationsstrukturen für verteilte Commit-Protokolle

Zur Wahrung der Transaktions-Atomarität in Verteilten Commit-Protokollen müssen sich alle an einer globalen Transaktion T beteiligten Knoten auf ein gemeinsames Commit-Ergebnis einigen. T muß also entweder an allen Knoten abgebrochen werden *(Abort)* oder an allen Knoten erfolgreich zu Ende kommen*(Commit)*. Um dies zu erreichen ist ein **verteiltes Commit-Protokoll** erforderlich.

Bei den vorzustellenden Commit-Protokollen nehmen wir an, daß an jedem Knoten ein **Transaktions-Manager (TM)** existiert, der für die Transaktionsverwaltung lokal ausgeführten Sub-Transaktionen verantwortlich ist und mit Transaktions-Managern der anderen Knoten im Rahmen des Commit-Protokolls kooperiert. In verteilten Commit-Protokollen kommt dem Transaktions-Manager des Koordinator-Knotens die Rolle des Commit-Koordinators zu. Aufgabe des Koordinators ist die Ermittlung des globalen Commit-Ergebnisses (Commit oder Abort) und dessen Mitteilung an die der Transaktions-ausführung beteiligten Rechner. Die Transaktions-Manager, an denen eine Sub-Transaktion ausgeführt wurde, wird hier als Agent bezeichnet.

Zur Kommunikation zwischen Koordinator und Agenten kommen unterschiedliche Aufrufstrukturen in Betracht.

- Bei der **zentralisierten** Commit-Struktur kommuniziert der Koordinator direkt mit jedem Agenten, während zwischen den Agenten i.d.R. keine Kommunikation stattfindet.
- Bei der **linearen** Commit-Struktur erfolgt die Commit-Behandlung sequentiell an den Transaktions-Managern der Knoten.
- Die **hierarchische** Commit-Struktur berücksichtigt die hierarchische Aufrufstruktur innerhalb einer globalen Transaktion, wie sie im Transaktions-Baum reflektiert ist.

Es gibt verschiedene Zwei-Phasen-Commit-Protokolle (2PC-Protokolle), welche auf diesen Kommunikationsstrukturen basieren.

- *Verteiltes* Zwei-Phasen-Commit-Protokoll
- *Lineares* Zwei-Phasen-Commit-Protokoll
- *Hierarchisches* Zwei-Phasen-Commit-Protokoll

Das Verteilte Zwei-Phasen-Commit-Protokoll realisiert die beiden Phasen der Commit-Behandung verteilt und basiert auf der zentralisierten Kommunikationsstruktur.

Der Nachrichtenfluß ist im folgenden aufgezeigt, wobei die gezeigten Nachrichten für jeden Agenten anfallen. Im einzelnen laufen dabei folgende Schritte ab:

1. Bei Transaktionsende sendet der Koordinator eine PREPARE-Nachricht gleich zeitig an alle Agenten, um deren lokales Commit-Ergebnis in Erfahrung zu bringen.
2. Nach Empfang der PREPARE-Nachricht sichert der bei einer erfolgreich zu Ende gekommenen Sub-Transaktion sämtliche Datenbankänderungen sowie einen Prepared-Satz auf die lokale Log-Datei. Anschließend sendet der Agent eine READY-Nachricht an den Koordinator zurück. Danach wartet der Agent bis der Koordinator den Ausgang der globalen Transaktion mitteilt. Für eine gescheiterte Sub-Transaktion werden ein Abort-Satz auf die lokale Log-Datei geschrieben

und eine FAILED-Nachricht zum Koordinator geschickt. Der Agent setzt die Sub-Transaktion zurück, wobei auch von ihr gehaltene Sperren freigegeben werden. Da das Scheitern der der globalen Transaktion damit feststeht, wird die Sub-Transaktion daraufhin bereits beendet.

3. Nach Eintreffen aller Antwortnachrichten der Agenten beim Koordinator ist *Phase 1* beendet. Haben alle Agenten mit READY geantwortet (und war das lokale Commit-Ergebnis am Koordinator-Knoten auch positiv), schreibt der Koordinator einen Commit-Satz in die lokale Log-Datei, woraufhin die globale Transaktion als erfolgreich beendet gilt. Danach wird eine Commit-Nachricht gleichzeitig an alle Agenten gesendet. Stimmte mindestens ein Agent mit FAILED, so ist damit auch die globale Transaktion zum Scheitern verurteilt. Der Koordinator schreibt daher einen Abort-Satz auf seinen Log und sendet eine ABORT-Nachricht an alle Agenten, die mit READY gestimmt haben.

4. Ein Agent schreibt nach Eintreffen einer COMMIT-Nachricht ebenfalls einen Commit-Satz auf die Log-Datei und gibt anschließend die Sperren der Sub-Transaktion frei. Bei einer ABORT-Nachricht werden ein Abort-Satz geschrieben und dieSub-Transaktion zurückgesetzt, wobei gehaltene Sperren ebenfalls freigegeben werden. Der Agent sendet danach zur Bestätigung noch eine Quittung (ACK-Nachricht) an den Koordinator. Nach Eingang aller ACK-Nachrichten beim Koordinator istdie globale Transaktion beendet, was durch einen Ende-Satz in der Log-Datei des Koordinators vermerkt wird.

Synchronisation

Damit viele Benutzer gleichzeitig ändernd und lesend auf die gemeinsamen Datenbestände zugreifen können, ist eine Synchronisation von Datenzugriffen nötig. Synchronisationsverfahren erfüllen das Korrektheitskriterium der *Serialisierbarkeit.* Dieses Kriterium verlangt, daß das Ergebnis einer parallelen Transaktionsausführung äquivalent ist zu dem Ergebnis irgendeiner der seriellen Ausführungsreihenfolgen der beteiligten Transaktionen. Die Synchronisation in verteilten Datenbanksystemen läßt sich in verteilten und zentralisierten Verfahren unterteilen.

Die Synchronisationsverfahren lassen sich in drei Klassen zuordnen:

- Sperrverfahren
- optimistische Verfahren
- Zeitmarkenverfahren

Die verteilten und zentralisierten Verfahren bestehen für Sperrverfahren sowieoptimistischen Verfahren, während Zeitmarkenverfahren nur für eine verteilte Realisierung sinnvoll ist. Die einzelnen Verfahrensklassen können vielfältig miteinander kombiniertwerden.

Sperrverfahren

Zur Sicherstellung der Serialisierbarkeit von Transaktionen werden 2-Phasen-*Sperrprotokolle* benutzt. Das *2-Phasen Sperrprotokoll* hat zwei Aktionen, **r/w LOCK** und **UNLOCK**. Durch die Aktion **LOCK** werden in der *Growing*-Phase auf das Objekt Sperren gesetzt, ohne einen wieder aufzuheben. Dadurch wird erreicht, da"s keine andere Transaktion das Objekt lesen und schreiben kann. Durch das erste Aufheben der Sperre mit der Aktion **UNLOCK** trifft die Transaktion in die sogenannte *Shrinking*-Phase, in welcher sie nur noch Sperren beseitigen kann. Man spricht von einem **strikten Zwei-Phasen-Sperrprotokoll**, wenn die Sperren bis zum Transaktionsende gehalten werden.

Optimistische Synchronisationsverfahren

Optimistische Synchronisationsverfahren greifen nicht in den Ablauf der Transaktion ein. Erst bei Transaktionsende wird überprüft, ob die Transaktion serialisierbar war. Ausführung einer Transaktion wird in drei Phasen unterteilt. In Lesephase, Validierungsphase und Schreibphase.

Zeitmarkenverfahren

Bei diesem Verfahren wird die Serialisierbarkeit durch Zeitstempel bzw. -markenan den Datenobjekten überprüft.Dabei bekommt jede Transaktion bei ihrem Anfang (BOT) eine eindeutige Zeitmarke fest zugeordnet, nach der sie serialisiert wird. Bei Zeitmarkenverfahren ist die Position einer Transaktion in der Serialisierungsreihenfolge bereits a priori durch ihre BOT-Zeitmarke festgelegt. Die Transaktionen werden also entsprechend ihrer Zeitmarke in der Serialisierungsreihenfolge abgearbeitet.

1.13.2 Deadlock

Bei Sperrverfahren gibt es die Gefahr von Verklemmungen auch **Deadlocks** genannt, deren charakterisierende Eigenschaft die *zyklische Wartebeziehung* zwischen zwei oder mehr Transaktionen ist. Im verteilten Fall können die Deadlocks zwischen Transaktionen verschiedener Rechner auftreten, so daß es zu **globalen Deadlocks** kommt.[Wei88]

Beispiel eines globalen Deadlocks

In einer Datenbankanwendung kann es zu einem globalen Deadlock zwischen zwei Übersetzungstransaktionen kommen, die auf dieselben Konten K1 (an Rechner R1) und K2 (an Rechner R2) zugreifen wollen. Dabei beabsichtigt die in R1 gestartete Transaktion T1 eine "Uberweisung von K1 nach K2, die Transaktion T2 in R2 eine Überweisung von K2 nach K1. T1 erwirbt zunächst eine Schreibsperre auf K1 und führt die Kontoänderung durch, danach startet sie eine Sub-Transaktion in R2, um auf K2 zuzugreifen. Die entsprechende Sperranforderung führt jedoch zu einem Konflikt, da T2 bereits eine Schreibsperre auf K2 erworben hat. T2 hat unterdessen eine Sub-Transaktion in R1 gestartet, um Konto K1 zu ändern; diese Sub-Transaktion gerät jedoch in einen Konflikt mit T1. Beide Transaktionen blockieren sich nunmehr gegenseitig; die Situation kann nur durch Rücksetzen einer der beiden Transaktionen behoben werden.

Deadlock-Verhütung

Die Deadlock-Verhütung ist dadurch gekennzeichnet, daß die Entstehung von Deadlocks verhindert wird, ohne daß dazu irgendwelche Maßnahmen während der Abarbeitung von Transaktionen erforderlich sind. Eine Transaktion fordert ihre Sperren bei BOT an. Verklemmungen können umgangen werden, indem jede Transaktion ihre Sperren in einer festgelegten Reihenfolge anfordert.

Deadlock-Vermeidung

Potentielle Deadlocks werden im voraus erkannt und durch entsprechende Maßnahmen vermieden. Im Gegensatz zur Deadlock-Verhütung ist eine Laufzeituntersuchung zur Deadlock-Behandlung erforderlich. Die Vermeidung von Deadlocks erfolgt generell durch Zurücksetzen von möglicherweise betroffenen Deadlocks.

Timeout-Verfahren

Bei diesem Verfahren wird auf die Transaktion eine Sperre festgelegt. Sobald die Transaktion die Zeitschranke (Time-out) überschreitet, wird sie zurückgesetzt.

1.13.3 Architektur von einem föderativen Datenbanksystem

Die Architektur von dem hier verwendeten Datenbanksystem basiert auf dem Konzept von **offen geschachtelten Transaktionen**.

Dazu wird die Zusatzschicht *Agent* eingeführt, welches verantwortlich ist, für die Kontrolle und Koordination von lokalen und globalen Transaktionen. Eine globale Transaktion wird von der FBDS in Sub-Transaktionen zerlegt, die dann von den jeweiligen lokalen Datenbanken (LDBS) bearbeitet werden. Aufgabe der FDBS ist es dann, die Teilergebnisse der jeweiligen Sub-Transaktionen zusammenzuf"uhren. Der Agent bekommt sowohl globale als auch lokale Transaktionen. Er mu"s daf"ur sorgen, da"s die Transaktionen miteinander koordiniert stattfinden. Man kann auch sagen, da"s der Agent wie ein Transaktionsmanager arbeitet. Die Applikationen können sowohl globale als auch lokale Transaktionen. Die lokalen Transaktionen gehen direkt in die Agenten-Schicht über.

Geschachtelte Transaktionen

Geschachtelte Transaktionen erlauben es, eine Transaktion in eine Hierarchie von Sub-Transaktionen zu zerlegen, die dann parallel zueinander ausgeführt werden können. Ein wesentlicher Merkmal geschachtelter Transaktionen ist es, daß Sub-Transaktionen isoliert zurückgesetzt werden können, ohne daß die gesamte Transaktion abgebrochen werden muß.[Rah94]

offen geschachtelte Transaktionen

Bei **offen** geschachtelten Transaktionen werden die Änderungen der Sub-Transaktionen einer Transaktion anderen Transaktionen bekannt gegeben. Bei Ende einer Sub-Transaktion werden also die Sperren freigesetzt. Andere Transaktionen können die Veränderungen sehen und ebenfalls Änderungen daran vornehmen.

geschlossen geschachtelte Transaktionen

Bei **geschlossen** geschachtelten Transaktionen werden die Sperren bis zum Transaktionsende festgehalten.

1.14 CSCW/BSCW Programmieren im Großen

Verfasser: Mischa Lohweber

1.14.1 Einleitung

Bei dem Prozeß der Softwareentwicklung spielt der Begriff des "Programmieren im Großen" eine zentrale Rolle. Dieses Referat soll die Problematik des "Programmieren im Großen" darlegen, die Begriffe CSCW und BSCW erläutern, deren Bedeutung erklären und Ideen zur Realisierung von CSCW aufzeigen. Unter anderem wird hier die Idee der "shared workspaces" ebenso dargestellt wie eine konkrete Lösung, der BSCW. Als Alternative wird das Programm Sniff+ unter dem Aspekt der "Programmieren im Großen" aufgegriffen und mit dem BSCW-System verglichen. Abschließend wird die Frage gestellt, ob sich BSCW für den Einsatz in unserer Projektgruppe lohnt.

1.14.2 Programmieren im Großen

Das "Programmieren im Großen" unterscheidet sich quantitativ und qualitativ vom "Programmieren im Kleinen". Die um Größenordnung höhere Komplexität der Problemstellung und der resultierenden Software des "Programmieren im Großen" im Gegensatz zum "Programmieren im Kleinen", die häufig Änderung während des Softwareeinsatzes und seit jüngerer Zeit die stärker in den Vordergrund rückende Wiederverwendung erfordern grundsätzlich neue Entwicklungsmethoden. Ursache für die Komplexität sind hohe Ansprüche an Softwarelösungen einerseits und der rasante Fortschritt auf dem Hardwaresektor andererseits. Moderne Hardware erlaubt die Implementation umfangreicher Software, deren Beherrschbarkeit wegen des ungleich langsameren Fortschritts auf dem Softwaresektor nicht mitgewachsen ist. Hauptgegenstand des "Programmieren im Großen" ist daher die Bewältigung der Komplexität. Wie beim "Programmieren im Kleinen" steht auch beim "Programmieren im Großen" die Zerlegung des Softwareproblems in kleine, handhabbare Teile, die dann dem "Programmieren im Kleinen" zugänglich sind, im Mittelpunkt. Allerdings orientiert sich eine gute Zerlegung nicht mehr an den Funktionalitäten, sondern an den Daten des Softwaresystems und geht auch nicht mehr streng 'top down' vor. Die Zerlegung resultiert in einer Architektur, die völlig verschieden von einer funktionalen Zerlegung ist, wie man sie vom "Programmieren im Kleinen" her kennt. Die Bausteine der Architektur, die Komponenten, sind Module und Teilsysteme, die Datenstrukturen und zugehörige Prozeduren zu Paketen verschnüren. Diese Komponenten sind erheblich reicher strukturiert als Prozeduren, welche die Bausteine des "Programmieren im Kleinen" bilden.

Von der Architektur des Softwaresystems hängt die spätere Änderbarkeit der Software in hohem Maße ab. Die Qualität einer Architektur ist schon deswegen so wichtig, weil die Änderbarkeit von Software zunehmend an Bedeutung gewonnen hat. Die häufige Änderung resultiert aus der langen Lebensdauer, die aus wirtschaftlichen Überlegungen zur Amortisierung der hohen Anschaffungskosten angesetzt werden muß. Fairley behauptet, daß Software mit einer Entwicklungszeit von ein bis zwei Jahren durchschnittlich zehn Jahre und länger eingesetzt wird[Fai85]. Für Änderungen gibt es üblicherweise zweierlei Anlässe. Sobald das Produkt von den Benutzern akzeptiert ist, wachsen die Ansprüche bezüglich erweiterter Funktionalität und verbessertem Komfort. Zum anderen überlebt eine erfolgreiche Software bei weitem die Computersysteme, für die sie ursprünglich erstellt worden ist. Die Anpassungen an Benutzerwünsche und neue Systemumgebungen summieren sich derart auf, daß - nach einer groben Faustregel - nach ca. 10 Jahren Einsatzzeit in einem Programm praktisch keine Originalzeile mehr existiert. Die Änderung bildet zusammen mit der Fehlersuche und -korrektur nach der Auslieferung des Produkts die Softwarewartung. Die Bedeutung der Wartbarkeit als Qualitätsmerkmal von Software belegt die Tatsache, daß es heute durchaus nicht ungewöhnlich ist, wenn Unternehmen 80% und mehr ihrer Entwicklungskapazitäten ausschließlich zum Warten bestehender Software verwenden. Man darf hierbei nicht vergessen, daß "Programmieren im Großen" sich auf die softwaretechnischen Aspekte des Entwicklungsprozesses beschränkt. Dies grenzt es von den allgemeinen Vorgehensmodellen ab, die auch weitergehende Tätigkeiten wie die Ermittlung der Produktanforderung, Qualitätssicherung und organisatorische Maßnahmen zur Planung, Durchführung und Kontrolle großer Projekte mit einbeziehen.[SP94]

1.14.3 CSCW - Computer Supported Cooperative Work

Der Begriff CSCW bedeutet computerunterstützte Gruppenarbeit und meint damit eine Koordinierung verschiedener Arbeitsabläufe in einem Projekt. In der heutigen Zeit spielt die Dezentralisierung von Firmen eine große Rolle. Internationale Zusammenkünfte, "joint ventures" und geographisch voneinander getrennte Organisationen müssen eine Möglichkeit finden effektiv zusammenzuarbeiten. Meist ist die Zusammenarbeit zwischen Organisationen in bestimmten Themen und Aufgaben zeitlich und räumlich beschränkt. Obwohl andere High-Tech Anwendungen wie die Video-Konferenzen in einigen Bereichen diese Grenzen bereits gebrochen haben, sind sie dennoch nicht so wichtig wie die Anwendungen die einen Zugriff auf Daten zu jeder Zeit und von über all aus erlauben[SP94]. Diese Systeme sollten auf minimalen technischen Infrastrukturen aufgebaut und plattformunabhängig sein. Untersuchungen haben gezeigt, das Gruppenarbeit nicht auf synchronen, sondern auf parallelen Arbeitsgängen basieren und somit ein Management unentbehrlich ist.

Idee des "shared workspace"

Im Gegensatz zu sogenannten "Workflow Systemen", welche Vorgaben zur Prozeßdefinition machen, also z.B. Abläufe beschreiben und fest definierte Koordinations- und Kooperationsmodule enthalten, basieren die "shared workspaces" auf dem Konzept, daß CSCW Anwendungen mehr informieren als Beschreiben sollen. Sie sollen eher als ein Medium als ein Mechanismus angesehen werden. Die Hauptaufgabe eines gemeinsam genutzten Arbeitsbereiches ("shared workspace") besteht darin, Objekte wie Dokumente, Tabellen, Texte, Binärdateien, Grafiken, etc. dem Benutzer zur Verfügung zu stellen. Diese Darstellung von Metainformationen und Aktivitäten der Objekte im "shared workspace" erlaubt es, dem Benutzer seine Zusammenarbeit mit anderen Gruppenmitgliedern auf der Basis der sozialen Protokolle und Kompetenzen zu ermöglichen. Als eine Konsequenz dieser Form von Zusammenarbeit, ungeachtet einer 10 Jahre langen Forschung auf dem Gebiet der Computerunterstützten Gruppenarbeit (CSCW), wird immer noch die Anwendung von e-mail und ftp oft als ausreichend angesehen. Obwohl diese Werkzeuge Möglichkeiten zum Informationsaustausch geben, sind die Möglichkeiten der Informationsteilung (information sharing) sehr begrenzt. So liegt es nahe, das eine Infrastruktur gebraucht wird, die den Ansprüchen eines CSCW-Systems genügen. Sogenannte "Groupware" Produkte wie "Lotus Notes" oder "Link Works" realisieren die Implementierung von "shared workspaces" in einer Organisation. Probleme aber treten meist erst auf, wenn die Grenzen der Organisation verlassen werden sollen und Daten mit anderen Firmen ausgetauscht werden. Verwenden die angesprochenen Firmen aber nicht dieselbe Software- und Hardwarekomponenten, sind viele Probleme wie z.B. die Verfügbarkeit auf anderen Plattformen, Absprache der Transportmöglichkeiten durch unterschiedliche Protokolle und Integration verschiedener Datenbanken zu lösen. Selbst wenn alle Firmen, die an dem kooperativen Prozeß beteiligt sind, dieselben Hard- und Softwarekomponenten besitzen, ist eine komplette Integration und ein reibungsloser Datenaustausch nicht immer sofort gewährleistet. Oft vergeht wertvolle Zeit zur Initialisierung und Abgleichung der Systeme verloren, die bei zeitlich begrenzten Projekten, wie zum Beispiel bei dieser Projektgruppe, zur Ineffektivität führen. So liegt die Lösung nahe eine Infrastruktur zu nehmen, die auf allen Plattformen existiert, dieselben Protokolle verwendet, und die keine, oder nur sehr wenig, Zeit zur Installation verschwendet. Das World Wide Web (WWW) bietet hier einige interessante Möglichkeiten.

- Das World Wide Web (WWW) ist unabhängig von der Art der Plattform, weil WWW Browser für alle gängigen Betriebssysteme wie UNIX, Windows und Macintosh OS existieren.
- Die Benutzeroberfläche von WWW Browser wie Netscape oder Mosaic ist einfach zu erlernen und scheint die Benutzeransprüche zu erfüllen, wie die große Akzeptanz des WWW von weniger fachlichen Benutzern zeigt.

- In vielen Organisationen sind bereits WWW Browser installiert und in Benutzung. Dies erfordert keine weitere Software oder zusätzliche Wartungsarbeiten.
- Ebenso sind in den meisten Firmen bereits WWW Server installiert, die unter Verwendung kleinerer Erweiterungen zur Nutzung eines "shared workspaces" genutzt werden können.

Im nächstem Kapitel 1.14.4 wird eine Anwendung präsentiert, welche die Idee des gemeinsamen Arbeitsbereiches ("shared workspace") benutzt und auf dem WWW aufbaut.

1.14.4 BSCW - Basic Support for Cooperative Work

Am Institut für angewandte Informationstechnik der Gesellschaft für Mathematik und Datenverarbeitung (GMD) existiert das Projekt BSCW [GMD96] . In diesem Projekt wird untersucht, wie nützliche Anwendungen für Kooperationsunterstützung auf das WWW aufgebaut werden können. Ein leicht modifizierter WWW Server stellt gemeinsame Arbeitsbereiche (shared workspaces) und erweiterte Funktionalitäten zur Verfügung.

Das BSCW System für gemeinsame Arbeitsbereiche

Das BSCW System für gemeinsam genutzte Arbeitsbereiche (shared workspaces) ist eine Dokumentenablage und ein Verarbeitungssystem mit erweiterten Funktionen zur Unterstützung von gemeinsamen Informationsaustausch. Das System besteht aus einem Server der mehrere Arbeitsbereiche beinhaltet, welche von unterschiedlichen Plattformen aus über unveränderte WWW Anwendungen, wie z.B. Netscape, erreicht werden können. Jeder dieser Arbeitsbereiche enthält ein Anzahl von Objekten, die von den verschiedenen Benutzern geholt und verändert werden können, oder über die man sich nähere Informationen anzeigen lassen kann. Die Objekte, die verwaltet werden können, sind Dokumente, Verknüpfungen (Links) zu WWW Seiten oder anderen Arbeitsbereichen, Verzeichnisse, Gruppen und Benutzer. Abbildung 1.11 zeigt ein typisches Bild eines gemeinsamen Arbeitsbereiches.

Jeder BSCW Server enthält einen Index aller Arbeitsbereiche, die auf diesem Server eingerichtet sind. Der Benutzer meldet sich beim System mit einer einfachen Namen und Paßwort Abfrage an und erhält dann eine Auswahl der ihm zugänglichen Arbeitsbereiche(Abbildung 1.12). Neue Benutzer melden sich mit ihrer e-mail Adresse beim System an, und erhalten dann automatisch eine e-mail, in welcher eine Verknüpfung (Link) aufgelistet ist. Der Benutzer kann, wenn er einmal angemeldet ist, sein Paßwort beliebig ändern. In der jetzigen Version muß ein neuer Benutzer von einem bereits existierenden Benutzer in einen Arbeitsbereich eingetragen werden, damit dieser Zugang zu diesem erhält. Es ist nicht möglich, sich selbst direkt in einem Arbeitsbereich anzumelden.

Ist der Benutzer nun erfolgreich beim BSCW System angemeldet, erscheint wie in Abbildung 1.12 dargestellt, eine Auswahl der ihm zugänglichen Arbeitsbereiche. In unserem Fall hat der Benutzer "Lohweber" die Auswahl zwischen den Arbeitsbereichen FOKIS und PRIVATE.

Hat der Benutzer einen Arbeitsbereich ausgewählt, sieht er das Hauptverzeichnis des Arbeitsbereiches ähnlich zu dem in Abbildung 1.11 dargestellten. Der Rahmen der Ansicht ist für jeden Benutzer derselbe, mit einer Navigationsleiste oben (und unten) auf der Seite, ein Banner für diesen Arbeitsbereich, eine Zeile mit Aktionsknöpfen und eine Auflistung der Inhalte des aktuellen Verzeichnisses dieses Arbeitsbereiches. Die Inhalte selbst werden unterschiedlich dargestellt, je nach Einstellung der Ansicht dieses Arbeitsbereiches. Im unserem Beispiel, Abbildung 1.11, hat der Benutzer "Lohweber" die Aktionszeile zu den jeweiligen Objekten und deren Beschreibung aktiviert. In unserem Fall sieht man fünf Einträge. Einen Artikeleintrag, ein Unterverzeichnis, eine Verknüpfung zu einer anderen WWW Seite, sowie zwei ASCII Texte. Die Aktionszeile unter den Objekten zeigt die Aktionen an, die auf dem jeweiligen Objekt möglich sind, bzw. zu denen der Benutzer befugt ist. Die Beschreibungszeile unter dem Objektnamen dient zur zusätzlichen Information des Objektes, was bei nicht aussagekräftigen Objektnamen, wie unter DOS von großem Vorteil ist. Rechts von den Objektnamen sind Ikonen zu sehen die Ereignisse darstellen welche, die Objekte betreffen. In unseren Fall wurde der Artikel

Abbildung 1.11: Beispiel eines gemeinsam genutzten Arbeitsbereiches.

Abbildung 1.12: Auswahl möglicher Arbeitsbereiche.

gelesen, das Unterverzeichnis neu erstellt, und es traten Ereignisse innerhalb des Unterverzeichnisses auf. Die Verknüpfung und die beiden Texte wurde in irgendeiner Form verändert. Diese Ereignisse beziehen sich auf Aktionen die passiert sind, seit sich der Benutzer "Lohweber" das letzte mal bei BSCW angemeldet hat. Da die angezeigten Seiten von dem BSCW Server immer direkt erstellt werden, ist zu beachten das Änderungen die von anderen Benutzern gemacht werden, nicht immer sofort sichtbar werden. Erst durch Aktivierung des "Catchup" Ereignisses wird der Status der Objekte zurückgesetzt, und die alten Ikonen verschwinden.

Um auf ein Objekt zuzugreifen, muß dieses angeklickt werden. Je nach Typ des Objektes können unterschiedliche Dinge passieren. Das Objekt wird als WWW Seite dargestellt, es wird eine bekannte Applikation gestartet oder ein Dateifenster erscheint um das Objekt lokal zu speichern. Die Aktionsleiste wird dazu verwendet dem Arbeitsbereich ein neues Objekt hinzuzufügen. Je nach Typ muß hier der Aktionsknopf "ADD URL" angeklickt werden, um eine Verknüpfung hinzuzufügen, "ADD DOC" aktiviert werden, um ein beliebiges (binäres) Objekt hinzuzufügen, und "ADD ARTICLE" gedrückt werden, um einen Artikel hinzuzufügen. Ein Artikel übernimmt die Funktion eines schwarzen Brettes. Hier können Nachrichten hinterlassen werden, auf welche dann geantwortet werden kann. Die Artikel, inklusive der Verweise auf andere Artikel, bleiben erhalten.

Am unteren Rand sind die festen Objekte "Bag", "Waste" und "Members" vorhanden. Der Koffer (Bag) enthält Objekte die mit der "Cut" Aktion ausgeschnitten worden sind und dient somit als Zwischenablage. Der Abfalleimer(Waste) erscheint gefüllt, wenn ein Objekt gelöscht wurde. Generell

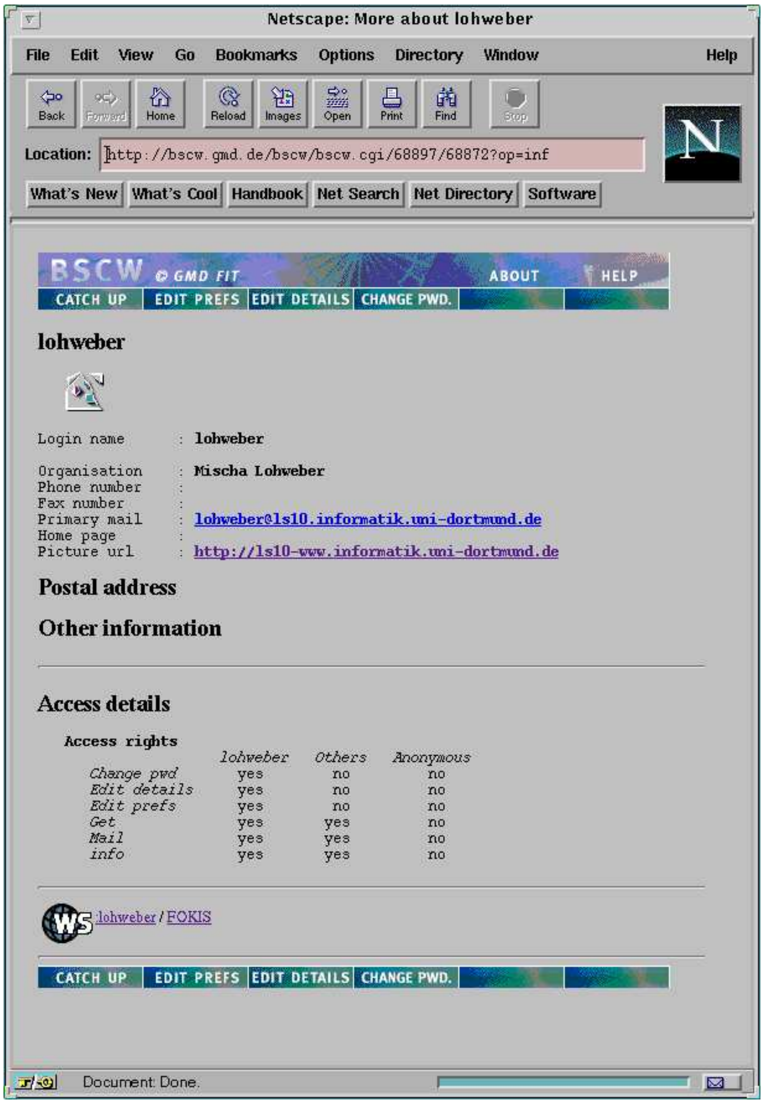


Abbildung 1.13: Beispiel für die Anzeige von Benutzerdaten.

kann jedes Mitglied des Arbeitsbereiches ein Objekt löschen. Dies erscheint dann im Abfalleimer und kann dort nur von dem Benutzer gelöscht werden, der es erstellt hat. Das Gruppensymbol (Members) liefert eine Auflistung aller Benutzer die auf diesen Arbeitsbereich Zugriff haben. Durch Auswahl eines dieser Mitglieder gelangt man in die Benutzeransicht, in welcher man weitere Informationen über den Benutzer einsehen kann, oder in welcher man seine eigenen Daten ändern kann. Ein Beispiel zeigt uns hier Abbildung 1.13. Der Benutzer "Lohweber" ist in der Benutzeransicht und hat nun verschiedene Möglichkeiten seine Daten zu ändern. So können Daten wie Telefonnummer, Faxnummer, Mailadresse, Verknüpfung zu einer Homepage und ein Verweis auf ein Bild angegeben werden. Ebenfalls werden die Zugriffsrechte auf diese Benutzeransicht dargestellt. Wie man erkennt, kann der Benutzer "Lohweber" beliebige Änderungen vornehmen, fremde, aber dem Arbeitsbereich zugehörige, Benutzer können sich dagegen die Daten nur ansehen. In dieser Ansicht hat der Benutzer die Möglichkeit, sein Paßwort zu ändern.

Begeben wir uns nun wieder zum Arbeitsbereich (Abbildung 1.11). Es sei noch zu erwähnen, daß um die Objekte systematisch zu ordnen, entsprechende Verzeichnisse angelegt werden sollten. Man erkennt also, daß das BSCW System eine erweiterte Dateiablage ist, die zusätzlich zur Verwaltung der Dateien

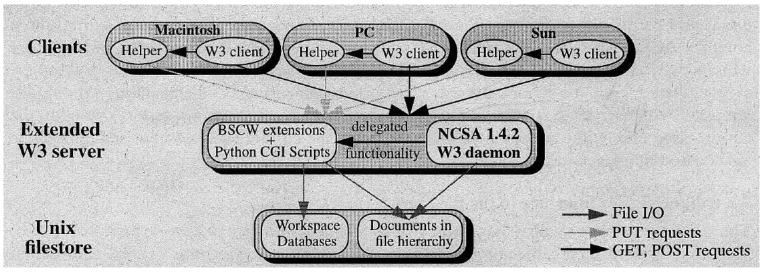


Abbildung 1.14: Architektur des BSCW Systems.

auch noch Metainformationen über diese liefert. Metainformationen meint Informationen, die keinem bestimmten Muster oder Schemata unterliegen, aber dennoch für den Benutzer informativ sind. Das System informiert also mehr, als das es bestimmt, was zu tun ist. Solch ein System kann natürlich nur funktionieren, wenn zusätzlich zu diesen Metainformationen das soziale Protokoll zwischen den Mitgliedern des Projektes funktioniert.

Implementierung des BSCW Servers

Die Schlüsselkomponente des BSCW Systems ist ein erweiterter WWW Server, eine einfache 'GDBM'-Datenbank in welcher Daten wie die Objekte des Arbeitsbereiches und Benutzerdaten gespeichert werden. Die Dokumente selbst werden in einem UNIX Dateisystem gespeichert, während die Datenbank Informationen über Dokumentersteller, Versionsnummer und Ereignisinformationen etc. enthält. Die aktuelle Version des BSCW Servers ist eine Erweiterung zu dem NCSA HTTP 1.4.2 daemon. Während ein Großteil der erweiterten Funktionalität von der CGI Schnittstelle, welche mit Phyton [Phy96] angesprochen wird, zur Verfügung gestellt wird, mußten etwa 30 Zeilen des original Server Codes verändert, bzw. erweitert werden. Warum diese Änderungen notwendig sind soll ein einfaches Bild über die Systemarchitektur des BSCW Systems zeigen (Abbildung 1.14).

Die Erweiterungen sind notwendig, weil z.B. das PUT Protokoll nicht trivial ist. Der Server muß sicher gehen, daß Dateien, die vielleicht als System- oder Scriptdateien erkannt werden nicht übertragen werden. Wird anstatt einer normalen WWW Anwendung wie z.B. Netscape der mitgelieferte "Helper" verwendet, bieten sich erweiterte Möglichkeiten. So wird anhand der Endung einer Datei versucht festzustellen, von welchem Typ die Datei ist, bzw. zu welchem Anwendungsprogramm sie gehört. Ebenfalls ist eine "Access Control List" (ACL) implementiert, welche in Form von Regel die Sichtbarkeit von Objekten und Aktionen, die der Benutzer machen kann und darf, regelt. Da der Server die angezeigten Arbeitsbereiche, also WWW Seiten, immer direkt generiert, ist außerdem eine erweiterte Schnittstelle zur Dateiablage nötig, die erkennt wer den Arbeitsbereich anfordert, welche Aktionen bei diesem Benutzer möglich sind, welche Ereignisse zuletzt eingetreten sind usw.. Der zusätzliche Code ist nicht direkt im Hauptteil des HTTP Servers verankert, sondern wird nur dann aufgerufen, wenn auf den Arbeitsbereich zugegriffen wird, ansonsten läuft der Server als normaler HTTP daemon.

Vor- und Nachteile von BSCW

Das BSCW System basiert auf einem modifizierten HTTP daemon. Es benutzt das WWW als Plattform und es kann mit unmodifizierten WWW Anwendungen auf verschiedenen Rechnerplattformen darauf zugegriffen werden. Es werden so keinen Systemvoraussetzungen oder eine neue Software gebraucht. Das System ist somit auch plattformunabhängig und -übergreifend. Ein weiter Vorteil ist die freie Verfügbarkeit, also keine, oder nur sehr niedrige Anschaffungskosten. Anstatt einer Befehlssprache erscheint eine einfache Oberfläche, die leicht zu bedienen und durch vorherige Arbeit mit WWW Browsern, wie z.B. Netscape, leicht verständlich ist. Ein einfacher Ereignisdienst, der es erlaubt, Buttons und Objekten einfach anzuklicken, um damit Aktionen auszulösen, unterstützt dieses Konzept noch. Die Erweiterung oder Veränderung der Arbeitsbereiche stellt ebenfalls kein Problem da. Da bei der Entwicklung des BSCW Systems darauf Wert gelegt wurde, einen schon bestehenden HTTP Server nicht, bzw. kaum zu modifizieren, um eine möglichst plattformunabhängige Umgebung zu schaffen, konnten einige Merkmale nicht, oder nur unbefriedigend realisiert werden. So ist z.B. keine Benutzerverwaltung vorhanden. Ein bereits bestehender Benutzer muß einen gemeinsamen Arbeitsbereich anlegen, und diesen dann für die anderen Benutzer freigeben. Ebenso gibt es keinen Administrator, der die Objekte verwalten könnte. Die Objekte, also Dateien, Links, etc. sind an den Benutzer gebunden und können auch nur von dem jeweiligen gelöscht werden. Die Erkennung von welchem Typ ein Objekt ist, ist noch unbefriedigend gelöst. Es wird versucht anhand der Dateiendung(suffix) zu erkennen, um was für eine Art Datei es sich handelt. Bei UNIX oder Macintosh Systemen kann dies zu einem Problem werden. Ebenfalls ist eine Synchronisation nicht möglich, die es z.B. erlauben könnte das zwei Benutzer ein Gespräch miteinander führen (talk). Durch die Verwendung von HTML als Sprache für die Benutzerschnittstelle treten einige Beschränkungen auf. So ist eine einfache Auswahl mehrerer Objekte, oder ein semantisches Feedback nicht leicht zu realisieren. Da das WWW auf einem hochfrequentierten Netz basiert, ist die Zeit die das Netz benötigt, um eine Anforderung, also eine HTML Seite darzustellen, zeitweilig extrem hoch. Teilweise treten Probleme beim Übertragen von Dokumenten in den Arbeitsbereich mit Netscape auf. Diese Probleme könnten durch die Installation eines mitgelieferten “Helpers” eventuell vermieden werden. Diese letzten negativen Aspekte würden höchstwahrscheinlich entfallen, wenn man einen lokalen BSCW Server einrichtet. Ebenfalls sollte nicht vergessen werden, daß das BSCW System keine leitende Funktion übernimmt. Es steuert in keinster Weise den Entwicklungsprozeß der Software, sondern dient ausschließlich dazu, die vorhandenen Module des Softwaresystems in Form von Dateien inklusive zusätzlicher Informationen zu verwalten.

1.14.5 Sniff+

Sniff+ ist eigentlich ein Programm, das zum Einsatz für das “Programmieren im Kleinen” gedacht ist. Deswegen ist es schwierig zu bewerten, ob Sniff+ sich für den Einsatz des “Programmieren im Großen” eignet. Die Vorteile von Sniff+ sind die guten Anbindungen an verschiedene Entwicklungs-, Versionierungs- und Compilerprogramme. Dies bietet dem Benutzer gute Möglichkeiten schnell und direkt zu programmieren. Die Objektdarstellung gibt eine Übersicht über die verwendeten Module und Klassen, wird aber bei großen Mengen sehr unübersichtlich. Die Projektstruktur muß vorher festgelegt werden und kann später gar nicht, bzw. nur mit sehr großem Aufwand geändert werden. Zusätzliche Einarbeitungs- und Einrichtungszeit fallen hier ebenfalls ins Gewicht. Andere Phasen der Softwareentwicklung, wie z.B. die Planung lassen sich nicht oder nur schwer einbinden. Außerdem ist Sniff+, soweit mir bekannt ist, nur auf UNIX Plattformen verfügbar und ein kommerzielles Produkt mit nicht zu verachtenden Anschaffungskosten. Sniff+ ist meiner Meinung somit nicht für den Einsatz des “Programmieren im Großen” in unserer Projektgruppe geeignet.

1.14.6 Abschluß

Der Einsatz im Rahmen der “Programmierung im Großen”, also der Zerlegung des Programms in verschiedene Module und die Bereitstellung dieser Module über das BSCW System, ist meiner Meinung nach nicht unbedingt sinnvoll. Da ein CSCW System informieren soll und die Zusammenarbeit somit hauptsächlich auf sozialen Protokollen beruht, könnte es zu Unstimmigkeiten kommen. Mit der Bereitstellung von Modulen meine ich die Ablage von Source Code. Das BSCW System wurde nicht als Schnittstelle zum Programmieren entwickelt und sollte somit auch nicht für die Verwaltung von Source Code verwendet werden. Das Programm Sniff+ ist dafür wesentlich besser geeignet. Als abstrakte Entwicklungshilfe für die Planung der Phasen und der Module kann BSCW eingesetzt werden. In einem Arbeitsbereich könnten Unterverzeichnisse für die einzelnen Module existieren, in denen dann Spezifikationen zu dem Modul und Anforderungen an Prozeduren und Funktionen in Bezug auf Implementierung stehen. Nochmals soll gesagt werden, das hier kein Source Code verwaltet werden soll. Ebenfalls ist das BSCW System hervorragend dafür geeignet, die Verwaltung der Dokumentation zu übernehmen. Eine weitere Einsatzmöglichkeit für das BSCW System ist die Möglichkeit unser entwickeltes Programm dem Kunden, oder dem Endbenutzer, via WWW zur Verfügung zu stellen. Der Endbenutzer findet im gemeinsamen Arbeitsbereich immer eine lauffähige Version. Er hat dann die Möglichkeit, über verschiedenen Artikel, die in diesem Arbeitsbereich existieren, Fragen zu stellen, Antworten zu bekommen, über Fehler des Programms zu berichten und somit über die Artikel mit den anderen über das Produkt zu kommunizieren. Weiterhin könnte ein Arbeitsbereich zur Fehlersuche eingerichtet werden. Meiner Meinung nach ist es unumgänglich, Protokoll über gefundene und später behobenen Fehler zu führen. Ein bekannt gewordener Fehler könnte dann als Artikel in den Arbeitsbereich gelegt werden. Dies hat den Vorteil, das alle Kommentare bezüglich dieses Fehlers in dem Artikel stehen würden und somit eine relativ gute Übersicht über vorhandene Fehler existiert. Es sei abschließend noch zu sagen, daß eine gewisse Disziplin beim Umgang mit dem System bewahrt werden sollte und eine rege Kommunikation zwischen den am Projekt arbeitenden Leuten unumgänglich ist. Da das System keinen ausgereiften Mechanismus zur Sperrung von Dateien hat, kann es bei unkommunikativem Verhalten leicht zum Verlust von Daten kommen.

1.15 Abrechnungssysteme im Krankenhaus

Verfasser: Xi Gao

1.15.1 Vorwort

Ziel der vorliegenden Arbeit ist es, die Teilfunktion „Abrechnungssysteme“ in einem Krankenhaus zu analysieren. Die dadurch gewonnenen Erkenntnisse können später in der Projektgruppe „Föderierte Objektorientierte Krankenhausinformationssysteme“ verwertet werden.

Um dieses Ziel zu gewährleisten und die Aussagefähigkeit der Arbeit auf einen möglichst aktuellen Stand zu bringen, basiert diese Arbeit auf dem Ergebnis einer empirischen Untersuchung eines Krankenhauses.

Das St. Josefs-Hospital ist 1870 gegründet worden. Jährlich nimmt das Krankenhaus ca. 3000 Patienten entgegen und entspricht mit seinen neusten Einrichtungen allen Ansprüchen und Anforderungen, die heute in betrieblicher und baulicher Hinsicht an ein modernes Krankenhaus gestellt werden.

Das St. Josefs-Hospital verfügt über ca. 360 Betten und besteht aus folgenden Stationen:

- Innere Medizin
- Chirurgie
- Urologie
- Frauenheilkunde und Geburtshilfe
- Hals-, Nasen-, Ohrenheilkunde

Die Abrechnungsabteilung des St. Josefs-Hospitals wird üblicherweise das Rechnungsbüro genannt, welches zuständig ist, Abrechnungen für sowohl stationäre als auch ambulante Fälle zu bearbeiten.

Um das Blickfeld auf das Wesentliche zu konzentrieren, wird im weiteren Verlauf der ambulante Fall nicht weiter berücksichtigt, da ohnehin der Kern der im Krankenhaus auftretenden Behandlungen von stationärer Natur ist.

Es sind zwei Mitarbeiter in dieser Abteilung beschäftigt. Sie verfügen über zwei IBM-3000-Terminals.

Diese Computerterminals sind in einem LAN (Local Area Network) vernetzt, so daß zwar innerhalb des Krankenhauses zugriffsberechtigte Personen Daten manipulieren können, aber keine Verbindung außerhalb des Krankenhauses besteht.

Alle Rechner innerhalb des LAN haben wiederum Zugriff auf einen sehr umfangreichen Datenbankbestand. In diesem sind die Patientendaten der letzten zehn Jahre gespeichert, und täglich kommen neue Daten hinzu.

Das bestehende Software ist vom Softwarehaus BOS in Bremen erstellt worden und heißt SPV (stationäre Patientenverwaltung).

1.15.2 Begriffserläuterung und -abgrenzung

Der Begriff „Krankenhausinformationssystem“ kann folgendermaßen definiert werden:

Ein umfassendes informationsverarbeitendes und informationsspeicherndes Gesamtsystem eines Krankenhauses, welches aus verschiedenen Teilsytemen wie z.B. Patientenaufnahme, Abrechnungssytem, Stationsverwaltung usw. besteht.

Das Abrechnungssystem ist dafür zuständig, alle Leistungen, die vom Krankenhaus erbracht und erfaßt wurde, in Rechnungsform umzusetzen.

Da der Großteil der Krankenhäuser zur Zeit nur bedingt oder teilweise computerisiert ist (insbesondere im Bereich des externen Informationsaustausches), und damit der Leser dieser Arbeit dennoch einen vollständigen Überblick über den Bereich der Abrechnung erhält, werden die Begriffe Krankenhausinformationssysteme und Abrechnungssysteme im weiteren Sinne verwendet, d.h. sie beinhalten sowohl den rechnergestützen Teil als auch den übrigen Teil der Systeme.

1.15.3 Über die Problematik des Pflegesatzes

Die neue Reform der Abrechnungsverfahren im Krankenhaus hat das Ziel, alle Krankenhaus- leistungen adäquat und wirklichkeitsnah zu erfassen und abzurechnen.

Unter Krankenhausleistungen versteht man insbesondere ärztliche Leistungen, Pflege, Versorgung mit Arzneimitteln, Unterkunft und Verpflegung.

Folgende Teilbeträge sind für die einzelnen Leistungsbereiche vereinbart:

1. Unterkunft und Verpflegung 35%
2. Pflege 25%
3. ärztliche Versorgung und sonstige medizinische Versorgung in den Pflegesätzen nach gesetzlichen Vorschriften 40%

Die vergüteten Leistungen des Krankenhauses werden durch gesetzlich vorgeschriebene Pflegesätze ausgedrückt und abgerechnet.

Nach dem neuen Recht fließen mehrere verschiedene Pflegesätze in die Abrechnung ein, nämlich Basispflegesatz, Abteilungspflegesatz, Fallpauschalen und Sonderentgelte. Diese sind die Bemessungsgrundlage für die Abrechnung.

Allgemeine Pflegesatzbestimmungen

Der allgemeine Pflegesatz besteht aus Basispflegesatz und Abteilungspflegesatz.

Der Basispflegesatz beinhaltet die grundlegenden Leistungen des Krankenhauses und ist der fixe Teil des zusammengesetzten Pflegesatzes. Die Höhe des Basispflegesatzes gilt einheitlich für alle Abteilungen / Stationen. Dagegen ist der Abteilungspflegesatz variable, d.h. er ist stationsabhängig.

Die Abrechnungsmethode lautet: (Basispflegesatz + Abteilungspflegesatz) * Verweildauer im Krankenhaus

Diese Abrechnungsmethode wird nur eingesetzt, wenn die entsprechende Behandlung nach der neuen Bundespflegesatz-Verordnung ab 01.01.1996 keine Anwendung von Fallpauschalen finden kann.

Fallpauschalen und Sonderentgelte

Fallpauschalen

Fallpauschalen beinhalten die gesamten Leistungen des Krankenhauses für einen Patienten unabhängig von der tatsächlichen Verweildauer.

d.h.

- In Fallpauschalen sind der Basispflegesatz und der Abteilungspflegesatz bereits berücksichtigt;
- Für jede Fallpauschale wird die sog. Grenzverweildauer veranschlagt.

Fallpauschalen gem. §11 Abs. 1 BPflV

Abteilung	FP-Nr.	Bezeichnung	Entgelt
.		.	.
.		.	.
.		.	.
Chirurgie	2,01	Einseitige subtotale Schilddrüsenresektion	4.519,21 DM
Chirurgie	2,02	Beidseitige subtotale Schilddrüsenresektion	4.833,93 DM
Chirurgie	12,01	Cholezystektomie, offen-chirurgisch	6.838,23 DM
Frauenheilkunde und Geburtshilfe	15,02	Hysterektomie	6.170,09 DM
Hals-, Nasen-, Ohrenheilkunde	5,01	Submuköse Korrektur am knöchernem Septum	2.135,72 DM

Abbildung 1.15: Fallpauschalen gem.11 Abs. 1 BPflV

Unter der Grenzverweildauer ist die gesetzlich vorgeschriebene Zeitschranke bzgl. einer bestimmten Erkrankung, innerhalb welcher der Patient behandelt und i.d.R. geheilt wird, zu verstehen. z.B. ein Patient, der an Mandelentzündung erkrankt ist, wird i.d.R. in 7 Tagen geheilt. Also liegt die Grenzverweildauer für Mandelentzündungen bei 7 Tagen.

Ein grundsätzliches Prinzip gem. BPflV ab 1. Januar 1996 lautet: stationäre Fälle, die als Fallpauschalen abgerechnet werden können, müssen auch in dieser Form abgerechnet werden. Das Verfahren Fallpauschalen zum Abrechnen wird z.Z. meistens in Situationen eingesetzt, wenn es um chirurgische Angriffe geht.

Die als Fallpauschalen abzurechnenden Behandlungen sind mit einheitlichen Punktewerten in der Bundespflegesatz-Verordnung aufgelistet. Ein kleiner Auszug davon:

Fallpauschalen gem. 11 Abs. 1 BPflV siehe Abb. 1.15

Im Prozeß der fortlaufenden Krankenhausreform sollen dem bisherigen Recht und dem alten Recht gegenüber noch mehr Fälle durch sog. Fallpauschalen und evtl. damit verbundenen Sonderentgelte ersetzt werden.

Diese Zielsetzung wird dadurch verwirklicht, indem man die repräsentativen Erkrankungsfälle, die bisher noch nicht als Fallpauschalen erfaßt sind, in Zukunft ebenfalls einbeziehen wird.

Sonderentgelte

Wie bei den Fallpauschalen, sind alle Sonderentgelte mit einheitlichen Punktwerten im gesetzlich vorgegebenen Katalog aufgelistet. Jedoch können Sonderentgelte niemals einzeln abgerechnet werden. Sie hängen immer mit dem jeweiligen Abteilungspflegesatz bzw. der Fallpauschale zusammen.

Sonderentgelte werden eingesetzt,

- wenn der Aufenthalt des Patienten die Grenzverweildauer überschreitet.
- wenn Nebenkrankheiten festgestellt werden.

z.B. ein Patient ist aufgrund eines Autounfalls, mit daraus resultierendem Schulterbruch ins Krankenhaus eingeliefert worden. Bei der Behandlung stellt man außerdem Herzprobleme bei ihm fest. Hier dürfen nicht zweimal Fallpauschalen abgerechnet werden, sonst wären die Unterkunft und die Verpflegung doppelt angerechnet.

In diesem Fall kann man zum einen dem Patienten in der Rechnungsform Fallpauschalen für die Behandlung wegen Schulterbruch abrechnen (Chirurgische Angriffe); zum anderen muß die Regelung für Sonderentgelte wegen der Herzoperation angewendet werden.

1.15.4 Ablauf eines typischen Abrechnungsprozesses

Im folgenden wird ein Prozeß beschrieben, der den Vorgang einer Abrechnung wiedergibt. Dabei werden die wichtigsten Ereignisse betrachtet.

Beschreibung der Aufgabenbereiche

Zu den Aufgaben der Mitarbeiter des Rechnungsbüros im St. Josefs-Hospital zählen folgende Teilbereiche:

- Abrechnunngsstatus der Patienten zu überprüfen,
- Rechnungen zu erstellen, wenn der Stichtag fällig ist,
- und Kostenübernahmeanträge an verschiedene Kostenträger zu stellen.

Kostenübernahmeabwicklung

Sobald der Patient ins Krankenhaus eingeliefert worden ist, wird er in die Datenbank des Krankenhauses aufgenommen. Die notwendigen personenbezogenen Daten (Familienname, Vorname, Versicherungsnummer, Haupt-/Zusatzkostenträger), die i.d.R. bei der Aufnahme bekanntgegeben werden, können sofort für das Rechnungsbüro zur Kostenübernahmeantrags- stellung dienen. Wenn zusätzlich die Diagnose erstellt und der Behandlungszeitraum von dem aufnehmenden Arzt voraussichtlich festgelegt werden kann, wird der Kostenübernahmeantrag an die zuständige Krankenkasse/Versicherung abgeschickt.

Mögliche Resultate Wenn alles problemlos verlaufen ist, bekommt man in kürzester Zeit die Garantie der Kostenübernahme des Patienten für den geschätzten Zeitraum von der Krankenkasse/ Versicherung schriftlich zurück.

Andernfalls wird der Kostenübernahmeantrag abgelehnt, wenn der Antrag nach Angaben seitens Krankenkasse/Versicherung nicht fehlerfrei ist. Es kann auch vorkommen, daß die Kostenübernahme nur für einen Teil des vom Krankenhaus geschätzten notwendigen Behandlungszeitraumes im Voraus bewilligt wird, wenn die jeweilige Krankenkasse/Versicherung für die Erkrankung des Patienten einen kürzeren Behandlungs- zeitraum veranschlagt hat.

In diesen Fällen muß der Abrechnungsvorgang für den Patienten vorerst gesperrt werden. Es müssen evtl. vom Rechnungsbüro Korrekturen vorgenommen werden, um schließlich erneut einen Kostenübernahmeantrag stellen zu können.

In kommender Zukunft soll die ganze Kostenübernahmeabwicklung mit den Krankenkassen/ Versicherungen in elektronischer Form durch Vernetzung zwischen Krankenhäusern und Krankenkassen/Versicherungen durchgeführt werden, um Zeit und unnötige Personal- aufwendungen zu sparen.

Faktorisierungsprozeß

Der Faktorisierungsprozeß wird ständig von einem bestehenden Computerprogramm ausgeführt. Die Bewertungsgrundlagen dieses Programmes sind die Kostenübernahmegarantie für einen Patienten und die notwendigen personenbezogene Daten. Während des Aktualisierungprozesses des Faktorisierungsprogramms werden zwei Listen erzeugt: die sogenannte Faktorisierungsvorschlagsliste, in der die Patienten mit Kosten- übernahmegarantie stehen, und die Restliste, welche diejenigen Patienten enthält, die nach Angaben der Patientendatenbank nicht abrechnungsfähig sind.

Die Faktorisierungsvorschlagsliste ist die Arbeitsgrundlage des Rechnungsbüros. Jedoch ist die Faktorisierungsvorschlagsliste kein Muß. d.h. Die Patienten, die in der Faktorisierungsvorschlagsliste stehen, müssen nicht abrechnungsfähig sein.

Der Faktorisierungsprozeß dient zur Erleichterung der Mitarbeiter des Rechnungsbüros, aber aus untengenannten Gründen müssen manchmal auch Korrekturen von Mitarbeitern vorgenommen werden.

Kriterien der Abrechnungsfähigkeit eines Patienten

Definition der Abrechnungsfähigkeit Die Behandlung eines Patienten ist abrechnungsfähig, wenn die Kostenträger des Patienten die Kostenübernahmegarantie innerhalb einer bestimmten Behandlungsdauer gewährleistet haben, und alle ärztlichen erbrachten Leistungen sowie die personenbezogenen Daten vollständig und korrekt eingetragen sind.

Die Überprüfung der Abrechnungsfähigkeit erfolgt demnach in zwei Schritten.

Überprüfung der Kostenübernahmegarantie

Wenn die Kostenübernahmegarantie für den Patienten gewährleistet ist, wird er in die Faktorisierungsvorschlagsliste eingetragen. Andernfalls wird er in die abrechnungsunfähige Liste eingetragen und der Abrechnungsvorgang wird somit erstmals gesperrt.

Aus Sicherheitsgründen und aus Erfahrungen mit Eigenarten verschiedener Krankenkassen/ Versicherungen muß das Rechnungsbüro nochmals die aufgerufene Faktorisierungvorschlags- liste nach Einzelheiten prüfen und evtl. den Abrechnungsstatus ändern. Es ist also durchaus möglich, daß der Mitarbeiter den Abrechnungsvorgang eines Patienten aus der Faktorisierungsvorschlagsliste erst wieder zurückstellt.

Dies ist z.B. der Fall, wenn es einen Schriftwechsel zwischen dessen Krankenkasse und dem Rechnungsbüro besteht, der nicht rechtzeitig im Computer erfaßt wurde.

Die Abrechnung wird dementsprechend zum späteren Zeitpunkt bearbeitet.

Überprüfung der Leistungserfassung

Nachdem der ersten Schritt ausgeführt worden ist, muß man im zweiten Schritt die Vollständigkeit und Korrektheit der personenbezogenen Daten, Behandlungszeitaum und aller ärztlichen erbrachten Leistungen überprüfen.

Wenn auch dieser Prüfungsvorgang erfolgreich abgeschlossen wurde, kann praktisch mit der Rechnungserstellung begonnen werden.

Falls die benötigten Daten doch nicht vollständig zum Abrechnungsstichtag vorliegen, muß man vom Rechnungsbüro aus den Abrechnungsvorgang vorerst sperren und bis zum nächsten Abrechnungsstichtag alle Daten vervollständigen.

Rechnungserstellung

Bis jetzt haben wir die einzelnen Maßnahmen der Vorbereitung auf die Rechnungserstellung betrachtet. Wenn der Patient nun nach sorgfältiger Nachprüfung des Rechnungsbüros abrechnungsfähig ist, kommt es zur eigentlichen Rechnungserstellung.

- Abrechnungsstichtag

  Im Krankenhaus St. Josefs-Hospital werden jeweils zum 1. und 15. des Monats Rechnungen erstellt. In einzelnen Situationen wird auch der individuelle Wunsch der Kostenträger nach Absprache in Rücksicht genommen.

- Rechnungsformen

  Für einen Kassenpatienten wird eine Rechnung an die Krankenkasse gestellt, während der Patient selber einen Eigenanteil von 12 DM/Tag nach gesetzlichen Vorschriften leisten muß.

Beispiel anhand eines Kassenpatienten:

Rechnung an Versicherung Bundesknappschaft

....
Pat.Nr.: 9601321
Patient: ******* 03.05.1928
Versicherter: ******* 03.05.1928
Aufn: 07.02.96 18:42
Entl: 12.02.96 11:00
Einw.-Diagnose:
Aufn.-Diagnose :574.2
Einw.-Inst. Notaufnahme
Entl.Art: 1

Von	Bis	Tarif	Anzahl	E-Preis	Betrag
Leistungsbezeichnung				%/DM Übernahme	
07.02.96	11.02.96				
allg.Pflegesatz		1000001	5	346,55	1.732,75
Abteilungspflegesatz nach neuem Recht → 1500					
07.02.96	11.02.96				
allg.Pflegesatz		1000001	5	141,11	705,55
Basispflegesatz nach neuem Recht → 1500					
07.02.96	12.02.96	1099999	6	12,00	72,00
Eigenanteil					
Rechnungsbetrag					**2.366,30**

Abbildung 1.16: Beispiel anhand eines Kassenpatienten

Im Gegensatz dazu entstehen zwei Rechnungen für einen Privatpatienten (auch Teilselbstzahler genannt), wenn die Behandlungskosten des Krankenhauses zum Teil von einer Versicherung übernommen werden.

Und schließlich bekommt ein Vollselbstzahler die entsprechende Rechnung zu eignen Händen

Erläuterung zu den Abrechnungstagen: Nach gesetzlichen Vorschriften dürfen die ärztlichen Leistungen mengenmäßig nur nach Übernachtungstagen abgerechnet werden. Im Gegensatz dazu wird die Verpflegung nach den wirklichen Aufenthaltstagen abgerechnet, deshalb ist die Anzahl des Eigenanteils an Verpflegung immer um eine Einheit höher als die Anzahl der Leistungsberechnung.

Quittierung der Abrechnung

Nachdem die Rechnung abgeschickt worden ist, muß man eigentlich nur noch auf die Überweisung von dem Kostenträger warten und den positiven Zahlungsstatus des Patienten in die Datenbank vermerken. Dann ist die Arbeit im Ganzen erledigt.

In seltenen Fällen weigert sich der Kostenträger allerdings zu zahlen. Dies kann selbstverständlich auch an einem Bearbeitungsfehler von Seiten der Abrechnungs- abteilung bzw. anderen Abteilungen liegen. In diesem Fall muß man dementsprechend die Fehler korrigieren bzw. korrigieren lassen und das ganze gleichzeitig auch in der Datenbank registrieren.

Andernfalls muß man bedauerlicherweise das 3-stufige-Mahnverfahren und evtl. weitere Gerichtsverfahren verwenden.

1.15.5 Schlußbetrachtung - Folgen und Auswirkungen der Krankenhausreform

Die fortlaufenden Krankenhausreformen, insbesondere das Gesundheitsstrukturgesetz (GSG) von 1992 und die Bundespflegesatz-Verordnung ab 01.01.1996 haben zu tiefgreifenden Veränderungen an den Krankenhäusern Deutschlands geführt.

Allein die neuen Entgeltformen haben sowohl sachliche als auch sozialpolitische Auswirkungen auf das Krankenhauspersonal und die ebenfalls betroffenen Patienten.

Auf der einen Seite werden die Ärzte vielmehr aufgefordert, eine ausreichende Kenntnis über die Behandlungsdokumentation zu besitzen, da sie schon während der Behandlung möglichst frühzeitig die Diagnose erstellen müssen, welche abrechnungstechnisch zur Kostenübernahme- abwicklung dient;

Auf der anderen Seite werden Behandlungen durch die Einführung von Fallpauschalen und Sonderentgelten standardisiert. Auf individuelle Therapien für einen Krankenfall muß evtl. verzichtet werden, da man möglicherweise bei der Abrechnung Schwierigkeiten haben kann.

Aus einem anderen Sichtwinkel betrachtet, wird die neue Entgeltform für die Krankenhäuser große sozialgesellschaftliche Auswirkungen haben. Betroffen sind vor allem ältere Menschen, deren tatsächliche Verweildauer aufgrund ihres schlechteren Gesundheitszustandes und längeren Heilungsprozesses meistens die Grenzverweildauer überschreitet.

Hierzu ein Beispiel: Eine 70-jährige Frau braucht ca. 14 Tage, damit sie sich von einer Operation wegen Mandelentzündung erholt und entlassen werden kann. Dagegen liegt die Grenzverweildauer für einen solchen Erkrankungsfall bei 7 Tagen. Häufig muß ein Patient deswegen den Aufenthalt schon frühzeitig beenden und das Krankenhaus verlassen. Dies hat es zur Folge, daß die Nachbehandlung bzw. die medizinisch notwendige Nachbetreuung im Krankenhaus zu kurz kommen kann.

Bemerkenswert ist es, daß die zuständige Krankenkasse dabei eine sehr große Rolle spielt, ob eine Verlängerung der Behandlung bewilligt wird und ob die Kosten übernommen werden. Dadurch wird der Unterschied der verschiedenen Krankenkassenphilosophien deutlich sichtbar.

Teil II

Die Erfahrungsberichte

1.16 Die STL - Tips zur Benutzung

Verfasser: Peter Ziesche, Djamel Kheldoun

1.16.1 Einleitung

Die Containerklassen der STL sind als generische Klassen (Templates) realisiert. Auf diese Weise kann jede Containerklasse mit jedem Datentyp benutzt werden. Man könnte das Konzept also zweidimensional nennen.

Die Container enthalten keine Algorithmen, um die in ihnen enthaltenen Elemente zu bearbeiten. Stattdessen sind Algorithmen als Funktionen implementiert, die auf allen Containern und allen Datentypen arbeiten. Sie stellen sozusagen die dritte Dimension dar.

Durch den Einsatz von Templates kann es keine Bibliotheken mit Objektcode geben. Der Compiler muß den benötigten Code abhängig von den benutzten Datentypen jedesmal neu generieren. Die STL ist also eine reine Quelltextbibliothek.

Bei der Entwicklung der STL wurde größter Wert auf Effizienz gelegt. Plausibilitätskontrollen oder Sicherheitsabfragen (z.B. ist ein Zeiger gültig) gibt es nicht. Wir werden für die Entwicklung in der PG daher die *Safe STL* einsetzen. Diese Bibliothek reimplementiert Teile der STL unter Berücksichtigung einer Reihe von Plausibilitätskontrollen.

Im weiteren werden die einzelnen Bestandteile genauer beschrieben. Die Informationen über die STL stammen im wesentlichen aus [Jos96] und [SL95].

1.16.2 Containerklassen

Die STL stellt mehrere Containerklassen zur Verfügung, die ihre Elemente in unterschiedlichen Datenstrukturen (z.B. verkettete Liste) verwalten. Entsprechend ergeben sich Vor- und Nachteile bei Einfüge-, Zugriffs- und Löschoperationen. Die Eigenschaften der Datenstrukturen werden als bekannt vorausgesetzt.

Die Container teilen sich in zwei Bereiche:

In *sequentiellen Containern* hat jedes Element seinen festen Platz. Dieser wird beim Einfügen festgelegt. Diese Container sind also erst einmal unsortiert. Die sequentiellen Container sind:

vector Die Elemente werden in einem dynamischen Array verwaltet, daß nur nach hinten wachsen kann. (Zugriff in $O(1)$, Einfügen am Ende in $O(1)$, Änderungen in der Mitte in $O(n)$)

deque Die Elemente werden in einem dynamischen Array verwaltet, das zu beiden Seiten wachsen kann. (Zugriff in $O(1)$, Einfügen am Anfang und am Ende in $O(1)$, Änderungen in der Mitte in $O(n)$)

list Die Elemente werden in einer doppelt verketteten Liste verwaltet. (Zugriff in $O(n)$, Einfügen in $O(1)$)

In *assoziativen Containern* werden die Elemente sortiert verwaltet. Sie sind als balancierte Binärbäume implementiert. Alle Operationen sind daher in O(log n) Zeit möglich. Im einzelnen stehen zur Verfügung:

set Es wird eine Menge im mathematischen Sinne simuliert. Die Elemente sind sortiert, jedes Element kann nur einmal enthalten sein. Wird ein bereits enthaltenes Element wiederholt eingefügt, ändert sich an der Menge nichts.

multiset Im wesentlichen handelt es sich auch hier um eine Menge, jedoch darf jedes Element mehrfach enthalten sein. Die Reihenfolge gleicher Elemente untereinander ist nicht spezifiziert.

map Dieser Container dient als Dictionary. Es wird ein Schlüssel/Werte-Paar verwaltet. Jeder Schlüssel darf nur einmal enthalten sein.

multimap Entsprechend Multiset handelt es sich um ein Dictionary, in dem jeder Schlüssel mehrfach enthalten sein darf.

Die Container verwenden zum Sortieren der Elemente die `<` und `==` Operatoren. Dabei wird standardmäßig ein bitweiser Vergleich zweier Elemente durchgeführt. Ist ein anderes Verhalten erwünscht, müssen die Operatoren für die Element-Klassen entsprechend definiert werden.

1.16.3 Benutzung der Container

Das Erzeugen eines Containers ist relativ einfach. Wir betrachten als Beispiel einen Vektor für ganze Zahlen:

```
typedef vector< int > TZahlen;
TZahlen aZahlen;
```

Damit haben wir einen neuen Typ definiert (Vektor für integers) und ein Objekt (Container) dieses Typs erzeugt (`aZahlen`). Nun wollen wir Elemente in den Container einfügen, am besten gleich mehrere, z.B. die Zahlen von 1 bis 10:

```
int iZahl;

for( iZahl = 1; iZahl <= 10; iZahl++ ) {
   aZahlen.push_bak( iZahl );
}
```

Um das Ergebnis sehen zu können, sprich den Inhalt des Containers ausgeben zu können, benötigen wir noch etwas Wissen über Iteratorn.

1.16.4 Iterieren über Container

Um über die unterschiedlichen Container iterieren zu können, sind abhängig von der Datenstruktur sehr unterschiedliche Verfahren erforderlich. Um das nächste Element zu finden, muß man in einem Binärbaum bekanntlich anders vorgehen als in einer Liste. Die STL vereinheitlicht die Iteration über Container durch den Einsatz von Iteratoren. Der Grundgedanke ist der: Ein Container weiß selbst am besten, wie er von innen aussieht und wie man über seine Elemente iterieren muß.

Jeder Container stellt den lokalen Typ `iterator` zur Verfügung. Damit können beliebig viele Iterator-Objekte erzeugt werden. Ein Iterator repräsentiert immer das aktuelle Element. Über den `*`-Operator kann so direkt darauf zugegriffen werden.

Außerdem bietet ein Container die Elementfunktionen `begin()` und `end()`. `begin()` liefert einen Iterator für das erste Element und kann gut zur Initialisierung von Iteratoren eingesetzt werden. `end()` gibt einen Iterator zurück, der *hinter* dem letzten Element steht.

Letztere Definition bedarf einer genaueren Erklärung. Auf Iteratoren ist im Allgemeinen kein `<`-Operator definiert. Durch die Definition von `end()` lassen sich trotzdem sehr elegant Abbruchbedingungen formulieren (siehe Beispiel). Auf diesen Iterator darf jedoch nie mit `*` zugegriffen werden.

Um aus unserem Beispielcontainer alle Elemente der Reihe nach auszulesen, können wir also wie folgt vorgehen:

```
TZahlen::iterator iPos;

for( iPos = aZahlen.begin(); iPos != aZahlen.end(); iPos++ ) {
   cout << *iPos << ' ';
}
```

Das Ergebnis lautet: “`1 2 3 4 5 6 7 8 9 10`”.

1.16.5 Algorithmen auf Container anwenden

Wie in 1.16.1 schon angedeutet, sind Algorithmen globale Funktionen, die auf Container angewendet werden können. Genauer gesagt arbeiten sie auf einem oder mehreren Bereichen, wobei ein Bereich durch je einen Iterator auf dem ersten und hinter dem letzten zu bearbeitenden Element besteht. So können also auch Teile eines Containers bequem bearbeitet werden.

Um z.B. die Reihenfolge der Elemente in `aZahlen` umzudrehen, können wir den `revers`-Algorithmus verwenden. Der Aufruf sieht so aus:

```
revers( aZahlen.begin(), aZahlen.end() );
```

oder etwas aufwendiger:

```
TZahlen::iterator iAnfang, iEnde;

iAnfang = aZahlen.begin();
iEnde   = aZahlen.end();

revers( iAnfang, iEnde );
```

Achtung: Das Ende eines Bereichs muß vom Anfang aus erreichbar sein, d.h. nach endlich vielen `iAnfang++`-Operationen muß `iAnfang == iEnde` gelten.

1.16.6 STL-Verzeichnis

Wir verzichten an dieser Stelle darauf, sämtliche Containerklassen mit ihren Elementfunktionen einzeln aufzuführen und zu beschreiben. Stattdessen verweisen wir auf den Report von Hewlett Packard [SL95].

1.17 Oracle-Forms

Verfasser: Dilber Yavuz, Betül Ilikli

1.17.1 Einführung

SQL*Forms ist ein Programmpaket, der die Erstellung von Anwendungen und getrennt davon deren Nutzung ermöglicht. Der Benutzer hat dadurch die mit SQL*Forms erstellten Anwendungen die Möglichkeit menügesteuert und maskenbezogen zu arbeiten. Mit nur minimalen SQL-Kenntnissen, kann er über Funktionstasten Abfragen in der Datenbank durchführen, Daten eingeben, ändern oder löschen.

1.17.2 Arbeitsweise von SQL*Forms

SQL*Forms ist, wie alle anderen ORACLE-Werkzeuge auch, an die Architektur des RDBMS und die Möglichkeiten von SQL gebunden. Datenbankzugriffe werden nur über SQL-Kommandos realisiert, die SQL*Forms automatisch generiert oder die der Entwickler explizit aufsetzen und zu genau definierten Zeitpunkten auslösen kann. Diese SQL-Zugriffe setzen Sperren, beginnen Transaktionen, die mit `COMMIT` oder `ROLLBACK` beendet werden. Im Gegensatz zu SQL*Plus hat SQL*Forms eine wesentlich verkürzte Sperrzeit. Dies ist daher möglich, daß SQL*Forms über einen eigenen Arbeitsspeicher verfügt. Alle Änderungen und Neueinträge werden zunächst nur in diesem Arbeitsspeicher durchgeführt. Erst wenn eine Transaktion mit `COMMIT` beendet wird, kommt es zu einer tatsächlichen Änderung, Einfügung oder Löschung der Datensätze in den jeweiligen Tabellen der Datenbank. Anschließend wird die Transaktion durch ein automatisches `COMMIT` beendet. Um die Datenintegrität zu gewährleisten, wird zum Zeitpunkt der Änderung eine exklusive Sperre auf den jeweiligen Satz in der Datenbank wirksam, indem SQL*Forms ein `select...for update` generiert. Das bedeutet, daß eine gleichzeitige Manipulation durch einen zweiten Anwender nicht mehr möglich ist.

1.17.3 Grundkonzepte von SQL*Forms

`Form`: Die oberste Arbeitseinheit bei SQL*Forms ist die **Form**. Eine Form ist eine Anwendung, die unter einem eindeutigen Namen generiert, aufgerufen und ausgeführt werden kann. Eine Form ist aus logischer Sicht eine Einheit von einer oder mehreren Bildschirmmasken.

`Block`: Eine Form muß aus einem oder mehreren **Blöcken** bestehen. Ein **Block** ist eine rein logische Größe. Er ist Bestandteil des Forms, der er angehört. Jeder Block muß innerhalb eines Forms durch einen eindeutigen Namen identifiziert werden. Ein Block erhält seinen Sinn durch die ihm optional zugeordnete Basistabelle.

`Item`: Jeder Block wiederum muß aus einem oder mehreren Feldern, sog. **Items** bestehen.Jedes Feld muß innerhalb eines Blocks durch einen eindeutigen Namen identifiziert werden. Felder können entweder Spalten der Basistabelle des Blocks sein, oder aber freie Felder.

`Page`: **Page** wird im Deutschen am besten mit Maske wiedergegeben. Eine Form besteht demnach aus einer oder mehreren Seiten, sprich Masken. Jede Seite wird über eine eindeutige Nummer identifiziert. Es können auf einer Seite mehrere Blöcke angeordnet werden. Umgekehrt kann sich aber ein Block auch über mehrere Seiten erstrecken.

1.17.4 Anwendung von SQL*Forms

Starten von SQL*Forms

Mit `f45desm` kann man SQL*Forms starten, wobei

- `f45` für die SQL*Forms Version 4.5 steht
- `des` eine Abkürzung für den Designer ist und
- `m` für Motiv steht.

Danach erscheint auf dem Bildschirm der Object Navigator.

Überblick über die Schritte für die Bildung der Form

- **Schritt 1:** Benennen der Form
  In den Object Navigator den Namen der Form reinschreiben.
- **Schritt 2:** Verbindung zu der Datenbank herstellen
  Mit dem `Connect`-Window unter Eingabe von Benutzernamen und Paßword Zugriff auf die Datenbank herstellen.
- **Schritt 3:** Erstellen von Blöcken
  Mit dem `NEW-BLOCK`-Window, welche in die Optionen
  tt General, Items, Layout und `Master/Detail` aufgeteilt ist, kann man einen neuen Block erstellen. In der Option General legt man fest, welches die Basistabelle zu dem Block sein soll und wie der jeweilige Block heißen soll. Mit Items legt man fest, welche Spalten von der Basistabelle als Items in dem Block vorkommen sollen und welche von Item es sein soll. Mit Layout legt man fest, wie die Bildschirmausgabe sein soll. Master/Detail wird, falls möglich die Join Condition zu dem jeweiligen Master Block hergestellt.
- **Schritt 4:** Fein-Einstellung des Layouts
  Bei diesem Schritt wird die Bildschirmausgabe überprüft und evtl. Veränderungen bezüglich der Anordnung der Items durchgeführt.
- **Schritt 5:** Setzen der Properties
  Mit dem `Properties`-Window kann man die Eigenschaften zu dem jeweiligen Item festlegen.
- **Schritt 6:** Hinzufügen eines Codes
  Die Funktionalität der Form kann durch hinzufügen von Triggern, welches durch den `PL/SQL` Editor bestimmt wird, erweitert werden.
- **Schritt 7:** Testen der Form
  Mit `Runform` kann die Form getestet werden.
- **Schritt 8:** Testen der Form

1.17.5 Trigger

In ORACLE hat man die Möglichkeit an einzelne Felder, Blöcke und Befehle die gesamte Eingabemaske zu knüpfen. Diese Bedingung nennt man Trigger. Zur Formulierung dieser Trigger dienen SQL-Befehle. Alle Trigger funktionieren situationsbezogen.

Benutzung von Triggern

Durch Trigger kann man folgende Situationen kontrollieren:

- Die Datenbank vor Operator-Fehlern schützen
- Operatorfehler auf spezifische Forms begrenzen

- Werte zwischen Feldern vergleichen
- Werte von Feldern berechnen und die Ergebnisse dieser Berechnung anzeigen
- Komplexe Transaktionen durchführen
- Fehler und Informationen dem Operator anzeigen

Strategien zur Entwicklung von Triggern

Bevor man Trigger schreibt, sollte man folgende Punkte beachten:

- In welchen Fällen wird der Trigger aufgerufen? (Bestimme den Namen oder den Typ des Triggers)
- Wo liegt der Kompotenzbereich des Triggers: Form, Block oder Item? (Bestimme die Ebene auf dem der Trigger definiert ist)
- Was soll der Trigger machen? (Bestimme die Befehle, die Logik und die built-ins, die benutzt werden)
- Welche Ausnahmen dürfen auftreten?

1.18 Der Cool–Orb

Verfasser: Patrick Koehne

1.18.1 Installation

Der Cool–Orb war bereits auf den Rechnern des LS10 installiert, sodaß sich von dieser Seite keine Schwierigkeiten ergaben. Im Prinzip sollte es dort auch relativ wenig Komplikationen geben, da die einzelnen Programme jeweils über eine globale Konfigurationsdatei gesteuert werden. Nachdem auch klar war, daß der Domain-Manager auf dem Rechner BERN gestartet werden muß, gab es keinerlei Probleme, den Cool-Orb zu starten.

1.18.2 Verwendung im eigenen Programmcode

Nach einigen anfänglichen Schwierigkeiten was das eigentliche Aufrufen der Cool-Orb Methoden anging, ist die Benutzung allerdings relativ reibungslos von statten gegangen. Einige kleinere Probleme, die aus der Anleitung nicht ersichtlich waren, konnten geklärt und bei der Implementierung entsprechend berücksichtig werden.

Von einigen programmiertechnischen Problemen also abgesehen, ist der Cool–Orb durchaus gut zu handbhaben.

1.19 Der ISO 290 - Standard für die Programmierung

Verfasser: Andreas Dinsch, Peter Ziesche

1.19.1 Einleitung

In der PG werden mehrere Leute unabhängig voneinander Programm-Texte schreiben müssen. Der gemeinsame Entwurf stellt sicher, daß die einzelnen Teile später zusammengesetzt werden können und zusammen funktionieren. Oft wird es jedoch auch notwendig sein, den eigenen Quelltext von einem anderen lesen zu lassen. Diese Richtlinien sollen helfen, die Quelltexte innerhalb der ganzen PG einheitlich zu gestalten um das Lesen fremder (und auch eigener) Quelltexte zu erleichtern.

1.19.2 Namenskonventionen

Typnamen

Es gibt im wesentlichen drei Arten von (selbstdefinierten) Typen in C++:

1. Klassen (`class`). Sie erhalten das Präfix C
2. Strukturen (`struct`). Sie erhalten das Präfix S
3. einfache Typen (`typedef`). Sie erhalten das Präfix T

Hinweis: Strukturen sollten mit `typedef` deklariert werden, so daß sie wie normale Typen verwendet werden können.

Beispiel für eine Struktur:

```
typedef struct {
   float fReal,
   fImaginaer
}
SComplex;
```

Beispiel für eine Klasse:

```
class CDame : public CPerson {
   string sName;
   void Tanze( CPerson* poPartner );
};
```

Beispiel für einen einfache selbstdefinierten Typ:

```
typedef int TIndex;
```

Variablennamen

Variablennamen sollen wie folgt aussehen:

- Ist der Name aus mehreren Wörtern zusammengesetzt, so werden diese ohne Trennzeichen hintereinander geschrieben.
- Jedes Wort beginnt mit einem Großbuchstaben

- Alle weiteren Buchstaben werden klein gechrieben.
- Enthält der Name Abkürzungen, die eigentlich in groß geschrieben werden (z.B. DOS) wird trotzdem nur der erste Buchstabe groß geschrieben.

Dem eigentlichen Namen wird ein *Präfix* vorangestellt, das über Lebensdauer und Typ der Variablen Auskunft gibt. Ein vollständiger Name hat also die Form:

```
[Lebensdauer]_[typ][Name]
```

Die Lebensdauer der Variablen wird durch folgende Kürzel beschrieben:

Präfix	Bedeutung
kein	lokale Variable
i_	lokale Variable, formaler Eingabeparameter
o_	lokale Variable, formaler Ausgabeparameter
m_	Membervariable einer Klasse
g_	globale Variable

Der Typ der Variablen wird durch eines der folgenden Kürzel beschrieben::

Präfix	Deklaration	Bedeutung
z	Tint	ganze Zahl (16 Bit)
n	Tuint	große ganze Zahl (32 Bit)
c	char	einzelnes ASCII-Zeichen
s	string	Zeichenkette
p	*	Zeiger des nachfolgenden Typs
b	bool	boolscher Wert (true oder false)
i	TIndex	Schleifen- oder Array-index
r	float	Fließkommazahl
d	double	Fließkommazahl mit höherer Genauigkeit
a		Container von Objekten des nachfolgenden Typs
o		Instanzen selbstdefinierter Klassen (Objekte)

Der Name wird gemäß den oben genannten Konventionen vergeben.

Einige Beispiele:

`int m_nKundenNummer`	eine ganze Zahl als Attribut einer Klasse
`TIndex iAktuellePosition`	Schleifenvariable, die die aktuelle Position festhält
`string i_sName`	ein String als Eingabeparameter einer Routine
`CKlasseX g_oInstanz1`	eine globale Instanz der Klasse CKlasseX
`CKlasseX* poInstanz2`	ein Zeiger auf eine Instanz der Klasse CKlasseX

Präfixe für selbstdefinierte Typen können ebenfalls definiert werden. Ein einfaches Beispiel dafür ist das Präfix `i` für Indizes. Eigentlich handelt es sich um ein `int`, müßte also mit einem `n` beginnen. Das neue Präfix macht jedoch den Verwendungszweck der Variablen viel besser klar.

Allgemein gilt: Der Name einer Variablen soll etwas darüber aussagen, *was* die Variable repräsentiert, *nicht wie* sie es repräsentiert.

Konstantennamen

Konstanten werden in Blockschrift (nur Großbuchstaben) geschrieben. Die Deklaration soll nicht mit der in C üblichen Präprozessoranweisung (`#define MAXSIZE 10`) erfolgen, sondern mit der `const`-Anweisung, also `const int MAXSIZE = 10;`

Aufzählungstypen werden wie Konstanten behandelt, d.h. auch sie werden in Blockschrift geschrieben.

Vordefinierte Typen in iso290.h

In der Datei "`iso290.h`"sind einige Typen vordefiniert, die in eigene Programme eingebunden werden können

1.19.3 Layoutkonventionen

Neben einer einheitlichen Namensgebung sollen die Quelltexte auch ein einheitliches Layout haben. Dadurch wird insbesondere das Lesen fremder Quelltexte erleichtert.

Zunchst einige grundsätzlichen Regeln:

- Pro Zeile wird immer nur eine Anweisung geschrieben.
- Pro Block wird um drei Zeichen eingerückt.
- Block-Klammern (geschweift) werden hinter dem sie aufrufenden Befehl geöffnet und in der gleichen Spalte des sie aufrufenden Befehls geschlossen.
- Öffnende Klammern werden ohne Leerzeichen an das vorhergehende Wort angehängt
- Zwischen der öffnenden Klammer und der nächsten Anweisung bleibt ein Leerzeichen, ebenso zwischen der letzten Anweisung und der schließenden Klammer.
- Zwei aufeinanderfolgende Klammern "()" werden ohne Leerzeichen geschrieben.
- Es werden ausschlielich C++ Kommentare "//" benutzt. Dadurch stehen C Kommentare "/* ...*/" noch zum auskommentieren ganzer Abschnitte zu Testzwecken zur Verfügung.
- Ein Kommentar wird so weit eingerückt wie der Quelltext, auf den er ich bezieht.

1.19.4 Beispiele

```
void CEineKlasse::EineMethode
(
   int     i_nParameterEins, // Beschreibung der Variablen
   string* o_psErgebnis      // Ergebnis ist ein String
)
// Implementierung
{
   // lokale Varibalen
   int nCount = 0; //Beschreibung der Variablen

   ...

   if( nCount > MAX_COUNT ) {
      DoSomething( i_nParameterEins );
   }
   else {
      DoNothing( i_nParameterEins );
   }

   ...

   while( nCount > 0 ) {
      DoSomethingOther( i_nParameterEins );
```

```
            nCount --;
        }

    } // Ende EineMethode
```

1.19.5 Annahmen dokumentieren

Bei der Implementierung einer Routine setzt man gewöhnlich gewisse Dinge über den Zustand des Systems oder die Werte der Eingabe-Parameter voraus. Solche Annahmen sind nur sinnvoll, wenn sie

- für den Benutzer einer Routine sichtbar sind und
- möglichst vom Compiler automatisch überprüft werden können.

Die Programmiersprache Eiffel unterstützt diese *programming-by-contract* Philosophie und stellt mächtige Konstrukte für die Realisierung bereit. Wir wollen versuchen, in C++ wenigstens einen Teil dieser Möglichkeiten nachzubilden.

Die Überprüfung von Eingabeparametern und alle sonstigen Annahmen über den Zustand des Systems sollten - soweit möglich - als boolesche Ausdrücke formuliert werden. Dadurch kann mit Hilfe des `assert`-Makros vom Compiler überprüft werden, ob die Annahmen erfüllt sind. Das `assert`-Makro erwartet als Parameter einen booleschen Ausdruck und einen String. Der Ausdruck wird ausgewertet. Ist er nicht erfüllt (`false`), wird der String ausgegeben und das Programm abgebrochen.

Ein Beispiel:

```
T* stack::Pop()
{
   // Require
   assert( Empty() != False, "Stack darf nicht leer sein" );

   T* poTopmost; //oberstes Element, Ergebnis
   // oberstes Element vom Stack nehmen

   // Ensure
   assert( *Top() != *poTopmost, "oberstes Element wurde entfernt" );

   return poTopmost;
} // Ende Pop()
```

Das `assert`-Makro sollte überall dort eingesetzt werden, wo Annahmen über irgendetwas gemacht werden. Läßt sich eine Annahme nicht als boolescher Ausdruck formulieren, sollte sie zumindest als Kommentar im Quelltext stehen. Damit wäre die zweite Anforderung an Annahmen erfüllt.

Um auch die erste erfüllen zu können, müssen die Annahmen nicht nur im Routinenrumpf, sondern auch bei der Deklaration, also in der Headerdatei stehen. Dazu gibt es in C++ jedoch keinen praktikablen Mechanismus. Auch die Konsistenz zwischen beiden Versionen kann nicht automatisch überprüft werden.

Es ist daher notwendig, die einmal geschriebenen Annahmen via copy-and-paste an die jeweils andere Stelle zu kopieren. In der Headerdatei sollte die obige Routine also wie folgt aussehen:

```
template<class T>
class CStack
{
   ...
```

```
    T* Pop();
    // Require
    // Empty() != False : Stack darf nicht leer sein
    // Ensure
    // *Top() != *poTopmost : oberstes Element wurde entfernt

    T* Top();
    ...
}; // Ende class CStack
```

Wenn eine Routine aus einer Basisklasse überschrieben werden soll, so müssen die dort aufgeführten Annahmen eventuell in die neue Routine hineinkopiert werden.

1.19.6 Tips- und Tricks

Initialisierung von Attributen

Im Konstruktor einer Klasse sollen die Attribute initialisiert werden. Oft ist es sinnvoll, vor der Initialisierung der Klasse selbst den Konstruktor der Basisklasse aufzurufen.

Die Zuweisung von Werten an Variablen sollte nicht im Rumpf des Konstruktors geschehen (z.B. a = 10), sondern vor dem eignetlichen Rumpf. Wenn eine read-only Instanz erzeugt wird, würde eine Zuweisung an eine Variable zu einem Compiler-Fehler führen.

Die Deklaration des Konstruktors sieht dann etwa so aus:

```
CKlasseX::CKlasseX()
  :CBasisKlasseY()
  :a( 10 )
  :b( "Name" )
//Implementation
{
   // Rumpf

}
```

Die Initialisierung der Variablen erfolgt in der Reihenfolge der Deklaration in der Header-Datei. Um Mißverständnissen vorzubeugen, sollten sie im Initialisierungsteil des Konstruktors in der Reihenfolge ihrer Deklaration stehen.

Vererbung

Nach Möglichkeit sollte Vererbung nur "public" vorgenommen werden.

Checklisten

Die Konventionen und Tips, die wir in diesem Artikel vorgestellt haben, basieren im wesentlichen auf [McC94]. Darin werden zu vielen Kapiteln auch Checklisten bereitgestellt, die den Inhalt in Form von "Haben Sie..." Fragen zusammenfassen. Sie eignen sich hervorragend dazu, die eigene Arbeit immer wieder abzuchecken. Einige Punkte sind ohne Kenntnis des jeweiligen Kapitels schwierig zu verstehen, was die Benutzbarkeit insgesamt aber nicht mindert.

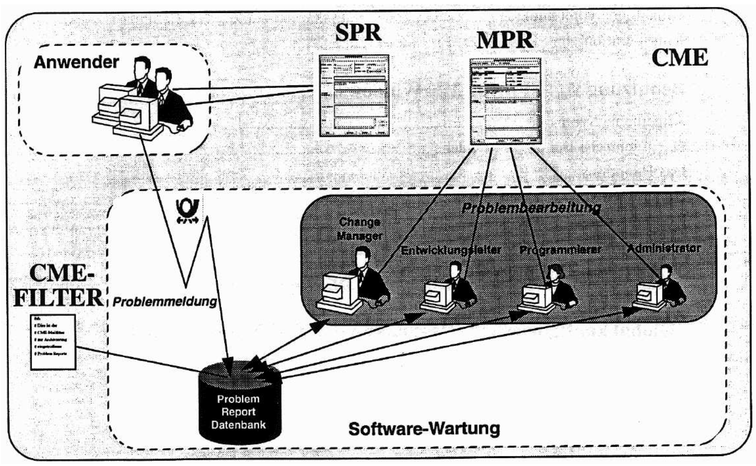


Abbildung 1.17: Problemmeldung und Bearbeitung.

1.20 CME

Verfasser: Mischa Lohweber

1.20.1 Einleitung

CME (Change-Management Enviroment) stellt eine Arbeitsumgebung dar, welche aus den Komponenten MPR (Manage Problem Report) und SPR (Send Problem Report) besteht. Diese Komponenten dienen zur Unterstützung der Entwicklung von Software. Bei Gebrauch von Software werden immer Fehler entdeckt, die den Softwareentwicklern in Form von Problemreporten mitgeteilt werden müssen. Um die Verwaltung der Reporte zu erleichtern, müssen alle eingehenden Fehlermeldungen bestimmte, notwendige Angaben enthalten wie z.B. Name der Software, Systemkomponente, Versionsnummer, Absender des Reports und eine detailierte Beschreibung des Problems. SPR verwaltet das Senden von Fehlerreports, wird also auf Seite der Personen eingesetzt, die die Software testen, oder Fehler finden. MPR dient zur Verwaltung der eingegangenen Fehlerreports auf Seiten der Entwickler (siehe Abbildung 1.17). CME wurde an der Fraunhofer-Einrichtung für Software- und Systemtechnik Dortmund [ISS] entwickelt und dem Lehrstuhl 10, im Rahmen der Projektgruppe, kostenlos zur Verfügung gestellt.

1.20.2 Installation

Das CME-System basiert auf dem `gnu`-Tool `gnats`, läuft aber eigenständig ohne eine vorherige Installation von `gnats`. Die Installation ist nicht trivial, da unter anderem ein eigener Benutzer im Unix-System angelegt werden muss, an welchen alle Fehlerreports gesendet werden. Der Mailfilter dieses Benutzers leitet die eingehende Mail an CME weiter und das System wertet die Mails dann aus. Die Einrichtung von CME war alles andere als leicht und wurde nur deshalb erfolgreich beendet, weil externe Hilfe beim ISST genutzt werden konnte.

Die Konfigurationsdateien sind nur über ASCII-Editoren zu bearbeiten. Die Installationsanleitung soll selbstbeschreibend sein, fällt aber etwas dürftig aus. Die mitgelieferten Beispieldateinen bringen ein wenig Licht in Sache. Hinzu kommt, daß das Tool nicht für das Betriebssystem Solaris entwicklet worden ist und es keine Erfahrungen über die Benuztung unter Solaris vorliegen. Meiner Meinung nach ist die alleinige Installation schwierig und bedarf der Hilfe einer Person, die schon einmal eine Installation durchgeführt hat.

1.20.3 Bewertung

Ist das CME Tool einmal installiert und lauffähig, stellt es ein gutes Werkzeug zur Verarbeitung und Organisation von Fehlermeldungen dar. Nach bisherigen Erfahrungen läuft das Tool unter Solaris stabil. Die Benuztung der Werkzeuge ist einfach und erfordert kaum Einarbeitungszeit. In einer guten Entwicklungsumgebung sollte ein solches Werkzeug nicht fehlen.

1.21 ODMG

Verfasser: Sven Gerding, Ulf Radmacher

Der ODMG-93 Standard [Cat96] gliedert sich in vier Hauptteile:

- Das Objekt-Modell
- Die Objekt Definitionssprache (ODL)
- Die Objekt Anfragesprache (OQL)
- Das C++/Smalltalk Binding

1.21.1 Das Objekt-Modell

Das ODMG-Objektmodell erweitert das Modell der OMG um datenbankspezifische Ansätze wie Persistenz von Objekten, Anfragen, Transaktionen und andere Eigenschaften von Objekten. Es soll eine formale Definition liefern, wie Daten in der Datenbank strukturiert werden sollen. Das Objektmodell gliedert sich, wie in Abbildung 1.18 beschrieben. Zuerst wird zwischen veränderbaren und nicht veränderbaren Objekten (Literale) unterschieden, anschließend zwischen atomaren und zusammengesetzten Literalen/Objekten. Jedes dieser Objekte hat eine bestimmte Charakteristik, welche sich aus den Methoden (*Operation*), die sich auf dem Objekt ausführen lassen, und seinen Eigenschaften (*Property*) ergibt. Eigenschaften können hierbei sowohl Attribute als auch Beziehungen zu anderen Objekten sein. Das Prinzip der Vererbung ist durch die Einführung von Subtypen und Supertypen realisiert worden, wobei Subtypen von Supertypen erben und Mehrfacherbung erlaubt ist.

1.21.2 Die Objektdefinitionssprache ODL

Die Objektdefinitionssprache *ODL* ist eine Spezifikationssprache, mit der die Schnittstellen von Objekten, die konform zum Objektmodell sind, beschrieben werden können. Durch die *ODL* besteht die Möglichkeit, Datenbankschemata zwischen konformen ODBMS auszutauschen. Die Grammatik der *ODL* kann in [Cat96] nachgelesen werden. Hier soll nur ein kurzes Beispiel angeführt werden:

```
   interface City
(  extent cities
   key city_code)
{
   attribute Unsigned Short city_code;
   attribute String name;
   attribute Set<Person> population}
```

Mit dem Schlüsselwort `extend` wird die Extension der Klasse bestimmt, also die Menge aller im Moment instantiierten Objekte. Des weiteren ist das Schlüsselwort `inverse` eingeführt worden, welches es der Datenbank erlaubt, die referentielle Integrität zu wahren. Beispiel:

```
interface Professor : Person
{
   extent professors;
   key (professor_id);
   attribute ...
   relationship Set<Course> teaches inverse Course::is_taught_by;};
```

So kann die Datenbank erkennen, daß die Beziehung `teaches` in Professor der Beziehung `is_taught_by` in Course entspricht und so beim Löschen eines Kurses sicherstellen, daß kein NIL-Pointer entsteht.

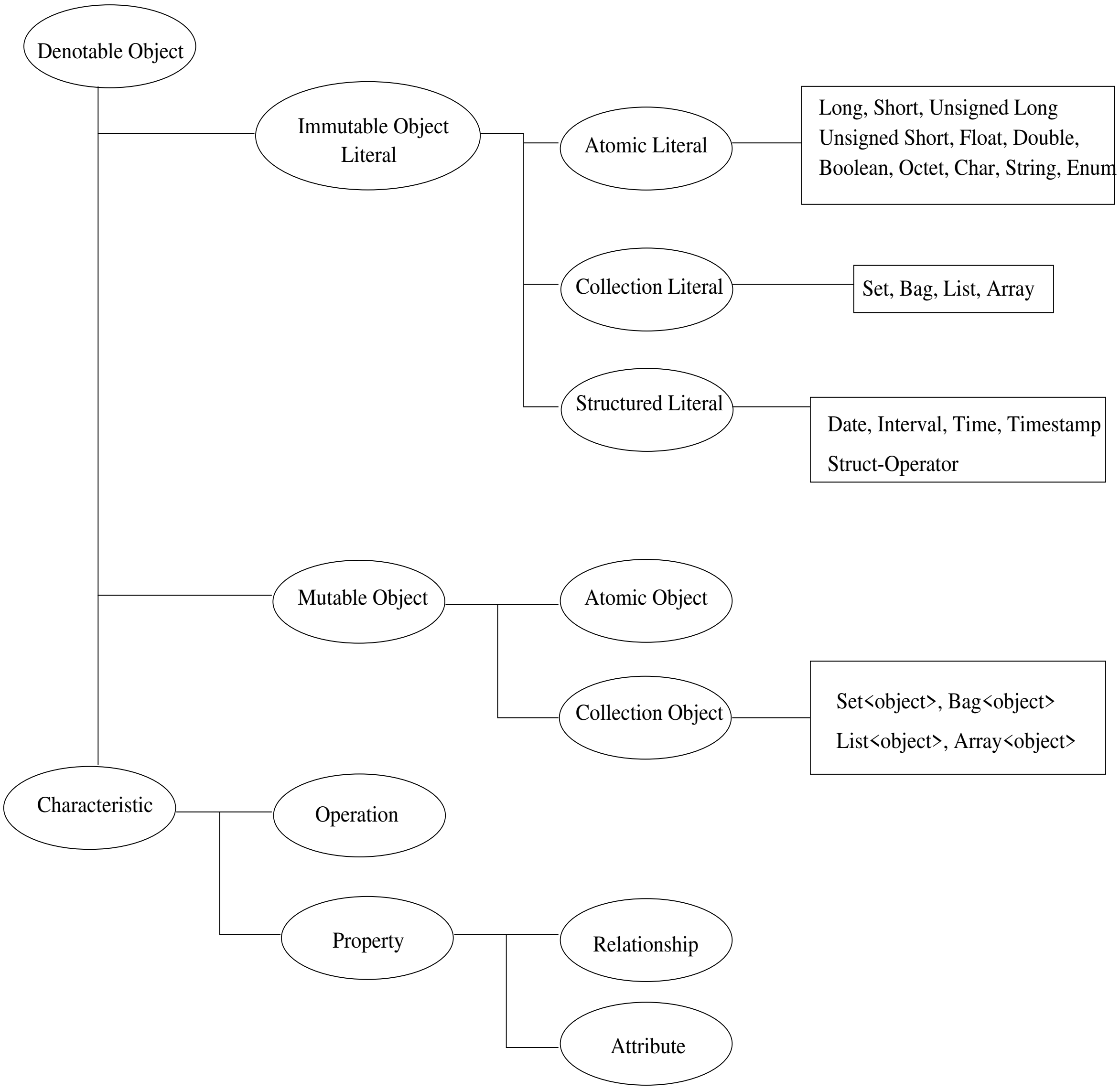


Abbildung 1.18: Das ODMG-93 Objektmodell

1.21.3 Die Objekt Anfragesprache OQL

- An SQL angelehnt
- Erlaubt DB-Anfragen und Erzeugung von Objekten
- Unterstützung von mutable Objects (Objekte mit OID)

  ```
  select distinct x
  from x in PG-Teilnehmer-Liste
  where x.name = "irgendwer"
  ```

  und immutable Objects (Literale)

  ```
  select distinct x.alter
  from x in PG-Teilnehmer-Liste
  where x.name = "irgendwer"
  ```

- Erzeugung von Objekten:

  ```
  PG-Teilnehmer(name: "a", alter: 33)
  ```

  Erzeugung von Strukturen:

  ```
  struct(name:"a", alter: 66)
  ```

  Erzeugung von Mengen:

  ```
  set(1,2,3)
  ```

- Anfragen `select` Ziel (`distinct`: nur unterschiedliche Werte) `from` Quelle `where` Selektionsbedingung
- Sortieren von Objekten (sort by)

  ```
  sort x in PG-Teilnehmer-Liste by x.alter x.name
  ```

- Unäre (max, min, sum, ...) und binäre (union, intersect, except) Mengenoperationen `set(1,2,3,4,5) intersect set(4,5,6,7) -> set(4,5)`
- define: Ausdruck als Abfrage

  ```
  define namen as select x.name from x in PG-Teilnehmer-Liste
  ```

- Universelle Quantifikation

  ```
  for all x in PG-Teilnehmer-Liste: x.alter > 20}
  ```

- Existentielle Quantifikation

  ```
  exists x in PG-Teilnehmer-Liste: n.name = "gerding"}
  ```

1.21.4 Das C++ Binding

- Einheitliches Typsystem
- Klassen, deren Instanzen persistent gemacht werden sollen, dürfen keine C++ Referenzen, unions oder Bitfelder enthalten. Zur Verfügung gestellt werden Typen wie string, date, time, ...
- Transparenter Zugriff auf Objekte über einen sog. "persistence Pointer", der ein Referenzobjekt vom Typ `Ref<T>` ist.
- Erzeugung von persistenten Objekten mit Hilfe des (überladenen) new-Operators:

temporär: `Ref<PG-Teilnehmer> teiln1 = new PG-Teilnehmer;`
persistent: `Ref<PG-Teilnehmer> teiln2 = new(database) PG-Teilnehmer;`
`Ref<PG-Teilnehmer> teiln3 = new(teiln2) PG-Teilnehmer;`

- Löschen von Objekten aus Speicher und Datenbank mittels `Ref::destroy` Funktion.

```
Ref<PG-Teilnehmer> teiln2;
teiln2.destroy();
```

- Modifikation von Objekten ist nur innerhalb von Transaktionen möglich. Hier muss nach der Modifikation eines Objektes (auch nach Lösch-Operation) die Funktion `mark_modified()` aufgerufen werden, um das ODBS auf den neuen Stand zu bringen.

```
Ref<PG-Teilnehmer> teiln;
transaction t;
t.begin();
...
teiln->name="gerding";
teiln->mark_modified();
...
t.commit();
```

- Beziehungen zwischen Objekten (referentielle Integrität)

```
Ref<T>                       1:1 und 1:n
Set<Ref<T> >  n:n und n:1
List<Ref<T> > n:n und n:1

Ref<Person> Gatte inverse Person::Gatte;
List<Ref<Person> > Kinder inverse Person::Eltern;
List<Ref<Person> > Kinder inverse Person::Eltern;
```

- Globaler Namensraum zum Auffinden von Objekten einer DB.

  Transaktionen: `begin()` Transaktionsstart
  `commit()` Transaktionsende - Änderungen speichern
  `checkpoint()` Änderungen schreiben
  `abort()` Änderungen verwerfen

- C++ OQL: Queries in C++ eingebunden

```
int query(Ref<Collection<T> > &result, const chat *predicate);

Ref<Set<PG-Teilnehmer> > informatiker;
PG-Teilnehmer->query(informatiker, select x
                                   from x in PG-Teilnehmer
                                   where x.alter > 25);
```

Teil III

Die Zeitplanung für das 1. Semester

Kapitel 2

Die Pert-Charts

Verfasser: Patrick Koehne

Die Verwendung des sogenannten Pert–Charts [GJM91] ermöglicht es relativ einfach, die einzelnen Phasen eines Projektes darzustellen, deren kausalen Zusammenhänge herzustellen und zeitliche Terminierungen mit einzubeziehen. Dadurch dient das Pert-Chart dem leichteren Überblick über den Projektverlauf und es können eventuell zeitkritische Situationen schon während der Planung erkannt werden, so daß sich automatisch ein verstärktes Augenmerk auf diese Situation ergibt.

2.1 Das Abrechnungssystem

Aus dem Pert-Chart (Abbildung 2.1) für das laufende Projekt kann man ersehen, daß der Entwurf und die Implementierung des Abrechnungssystems, welches erst im 2. Semester benötigt wird, völlig kontextfrei zu den anderen Komponenten abläuft. Erst am Ende des 1. Semesters, wenn die Einarbeitungsaufgabe ebenfalls abgeschlossen ist und die Planung für das 2. Semester ansteht, fließen die Ergebnisse der Arbeit am Abrechnungssystem wieder mit dem Rest der Gruppe zusammen.

2.2 Die Einarbeitungsaufgabe

Nach der gemeinsamen Analyse für die Einarbeitungsaufgabe verzweigt die weitere Bearbeitung des Projektes in vier parallele Stränge. Jede von diesen Teilgruppen entwickelt ihren eigenen Entwurf und implementiert anschließend ihren Teil.

Nach der Fertigstellung des Feinentwurfes der einzelnen Gruppen ergeben sich nun für die weitere Implementierung Abhängigkeiten. So muß zunächst die IDL-Implementierung abgeschlossen sein, die für die Kommunikation mittels Cool–Orb zuständig ist, bevor die Implentierungen der Datenbanktreiber darauf aufsetzen können. Die eigentlichen Applikationen, die auf der Datenbank aufsetzen, bleiben von dieser Abhängigkeit unbeeinflußt. Bei der Implementierung von RUSERS kann man eine ähnliche Unterscheidung machen. Es kann hier in den Daemon und den Treiber unterteilt werden. Der Daemon schreibt unabhängig von dem Gesamtprojekt immer die aktuellen Daten in eine Datei und der Treiber arbeitet dann mit dem Förderierungssytem zusammen. Letztendlich müssen alle Teilkomponenten integriert werden.

Die zeitkritischen Phasen in diesem Projekt sind der Entwurf und die Implementierung der IDL, weil, wie zuvor erläutert, andere Systemkomponenten davon abhängig sind. Sollte es in dieser Phase des Projektes zu Zeitverzögerungen kommen, so kann dies zu Verzögerungen im gesamten Projekt, bzw. zum Scheitern des Projektes führen. Diese Phasen sind in Abbildung 2.1 fett angedeutet.

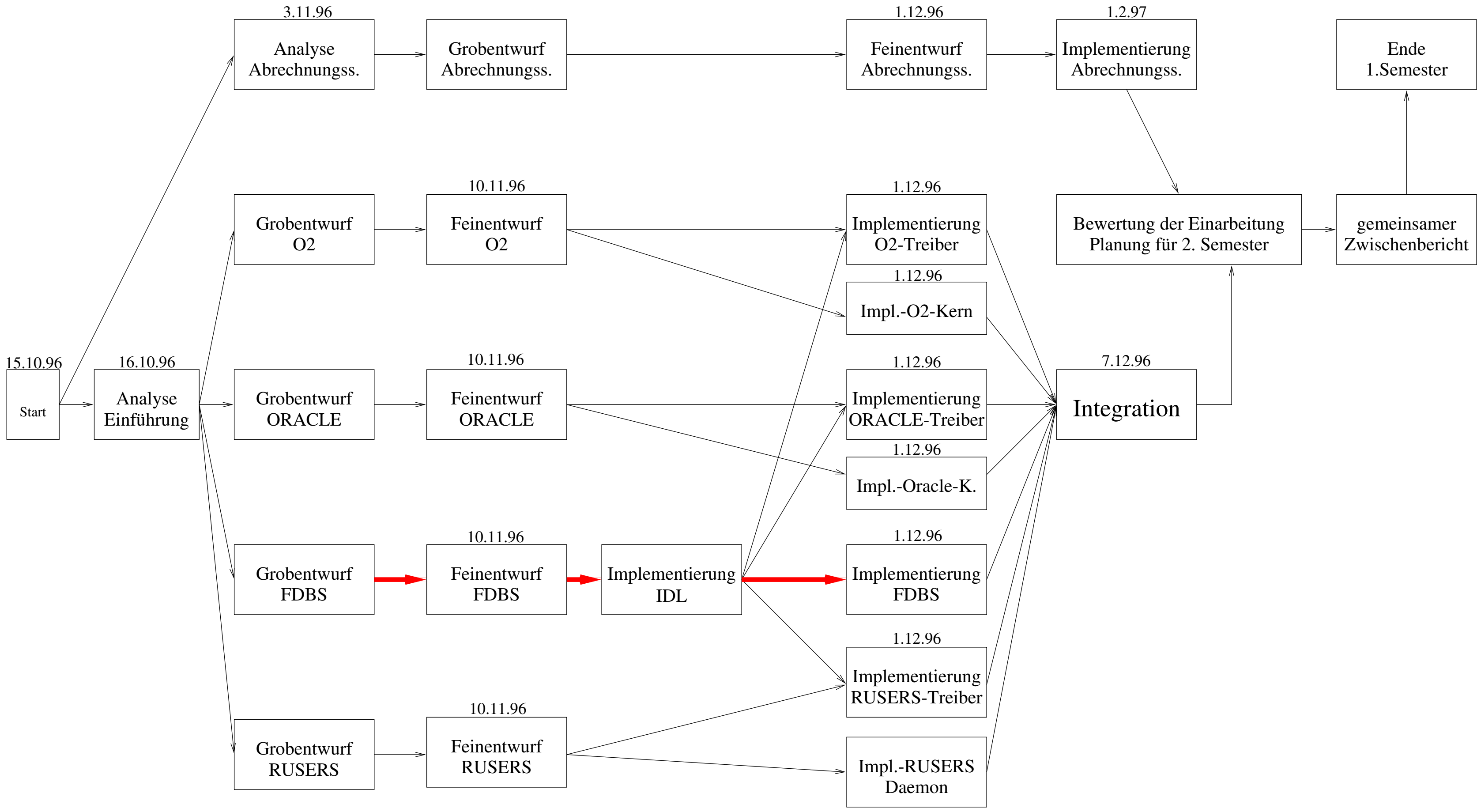


Abbildung 2.1: Das Pert–Chart für das 1. Semester.

Kapitel 3

Die Gantt–Charts

Verfasser: Patrick Koehne

3.1 Die Gantt–Charts für das 1. und 2. Semester

Zwei andere Sichten auf den zeitlichen Ablauf eines Projektes bieten die sogenannten Gantt–Charts (entwickelt von Henry L. Gantt) [GJM91]. Es werden zwei Charts unterschieden. Das erste Gantt–Chart (Abbildung 3.1) stellt den zeitlichen Ablaufes des Projektes diesmal ohne kausale Zusammenhänge dar. Die einzelnen Balken spiegeln die verschiedenen Phasen wieder, wie sie schon in Abbildung 2.1 vorgestellt wurden. Diesmal jedoch wird dargestellt, wieviel Zeit für die einzelnen Phasen eingeplant wurde (gesamter Balken), wieviel Zeit vermutlich dafür verbraucht wird (weißer Abschnitt des Balkens) und wieviel Zeit bis zum definitiven Ende der Phase vielleicht nicht verbraucht wird (grauer Abschnitt des Balkens). Diese Unterteilung macht es z.B. sehr einfach möglich, im Falle eines Engpasses Arbeitskraft von einer Teilaufgabe abzuziehen, und der Teilaufgabe, die zu scheitern droht, zuzuweisen. In dem Fall der Abbildung 3.1 sieht man also, daß während der Einarbeitungsaufgabe keine Arbeitsressourcen zur Verfügung stehen, während die Implementierung des Abrechnungssystems eventuell frühzeitig abgeschlossen werden kann. Die geplanten Zeiten zum Ende des ersten Semsters konnten nicht ganz eingehalten werden, da die Implementierung des FDBS-Kerns länger als geplant gedauert hat.

Das zweite Gantt–Chart (Abbildung 3.2) dagegen stellt die zeitliche Einteilung der einzelnen Projektmitglieder dar. Der graue Balken sagt zunächst einmal aus, über welchen Zeitraum die Personen an dem Projekt beteiligt sind. Dies ist in unserem Fall natürlich der gesamte Zeitraum. Die weißen, darin liegenden Balken zeigen an, zu welcher Zeit die einzelnen Personen an welchen Projekten mitarbeiten und wann, in unserem Falle nicht eingetragen, sie eventuell im Urlaub oder nicht verfügbar sind. In einer solchen Übersicht kann man die einzelnen Aktivitäten der Projektmitglieder leichter überblicken. Wenn man dieses Chart schon bei der Einteilung der Leute vorbereitet und benutzt hat, so gestaltet sich dann die Vergabe der einzelnen Aufgaben sicherlich leichter.

3.2 Die Entwicklung

3.2.1 Im Januar 1997

Die Abbildung 3.3 stellt nun einen Vergleich zu dem ursprünglich erstellten Chart in Abbildung 3.1. Zu dem damaligen Zeitpunkt der Chart (8. Jan. 97) befand sich das Projekt schon nach dem zuvor angesetzten Termin für die Fertigstellung der Implementationen und hätte sich bereits bei der Integration befinden sollen. Das Ziel konnte zu dem damaligen Zeitpunkt nicht eingehalten werden, da der Aufwand zur Entwicklung des förderierten Systems unterschätzt wurde.

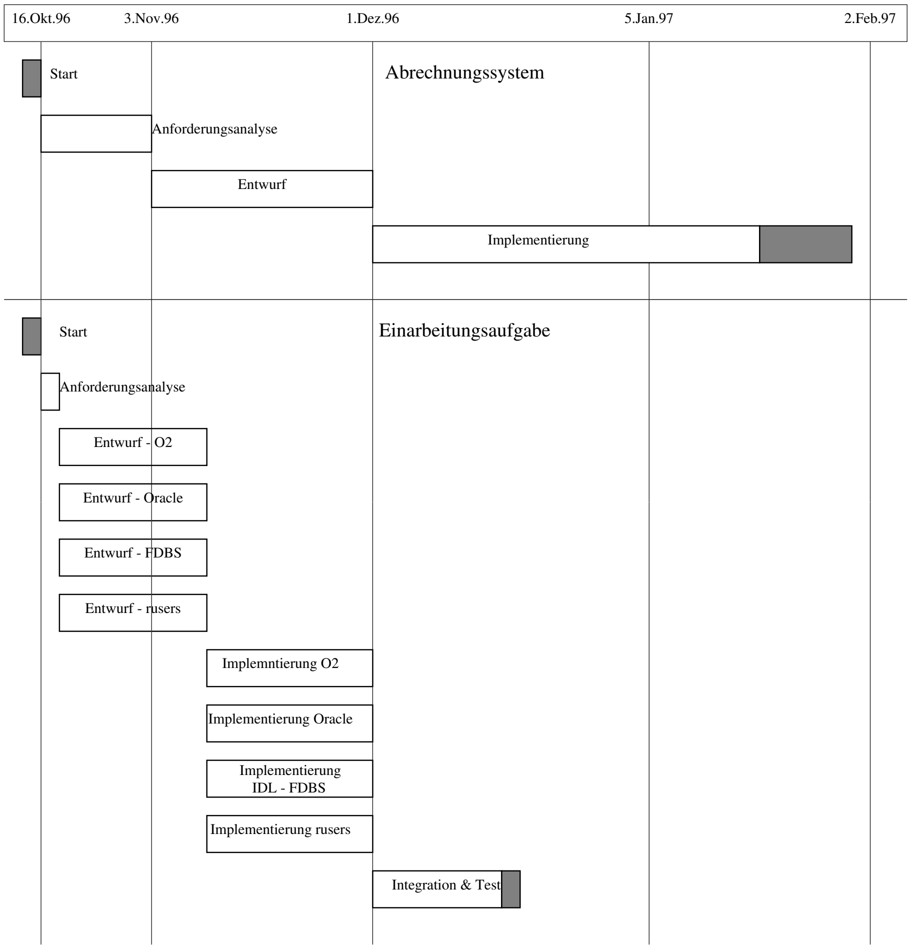


Abbildung 3.1: Das Gantt–Chart für die Planung des gesamten Projektes bis Ende des 1. Semesters.

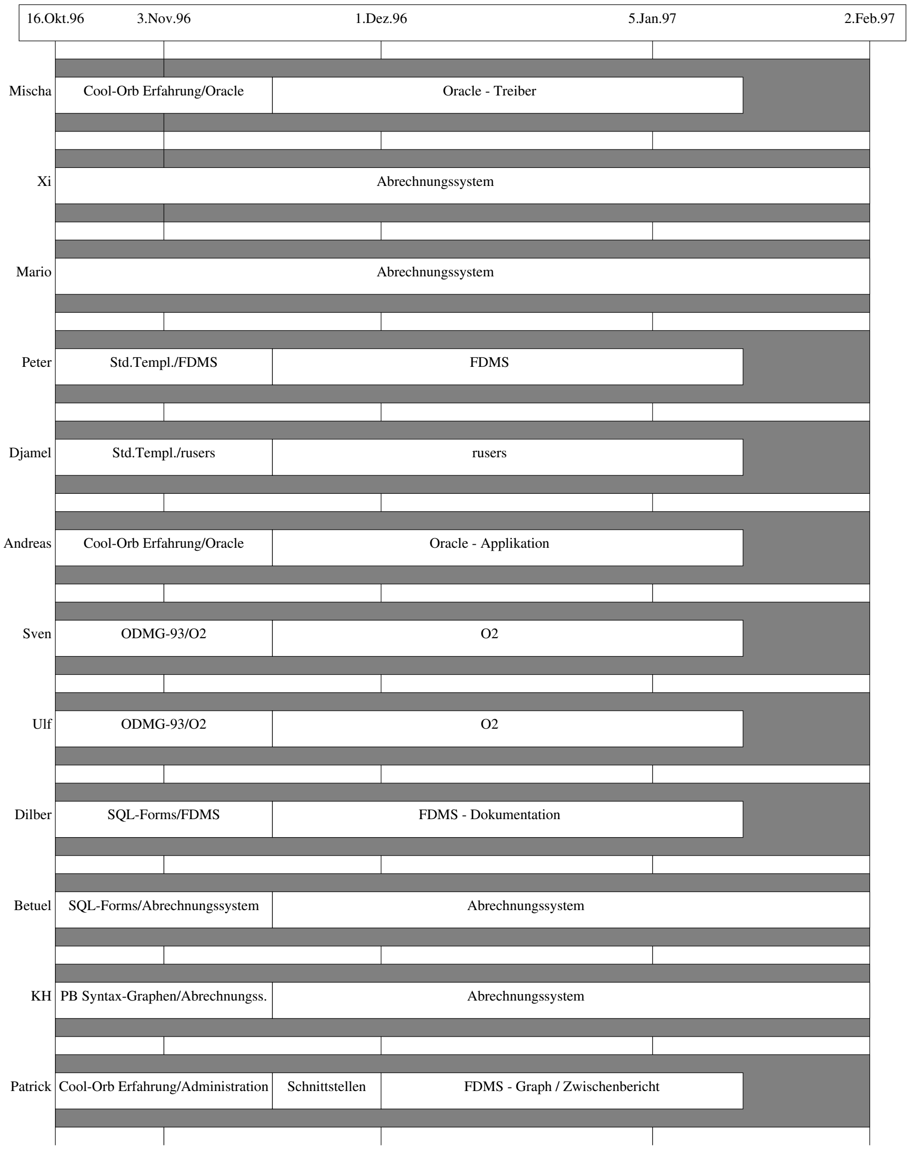


Abbildung 3.2: Das Gantt–Chart zur Aufgabeneinteilung der einzelnen Projektmitglieder (Planungsphase).

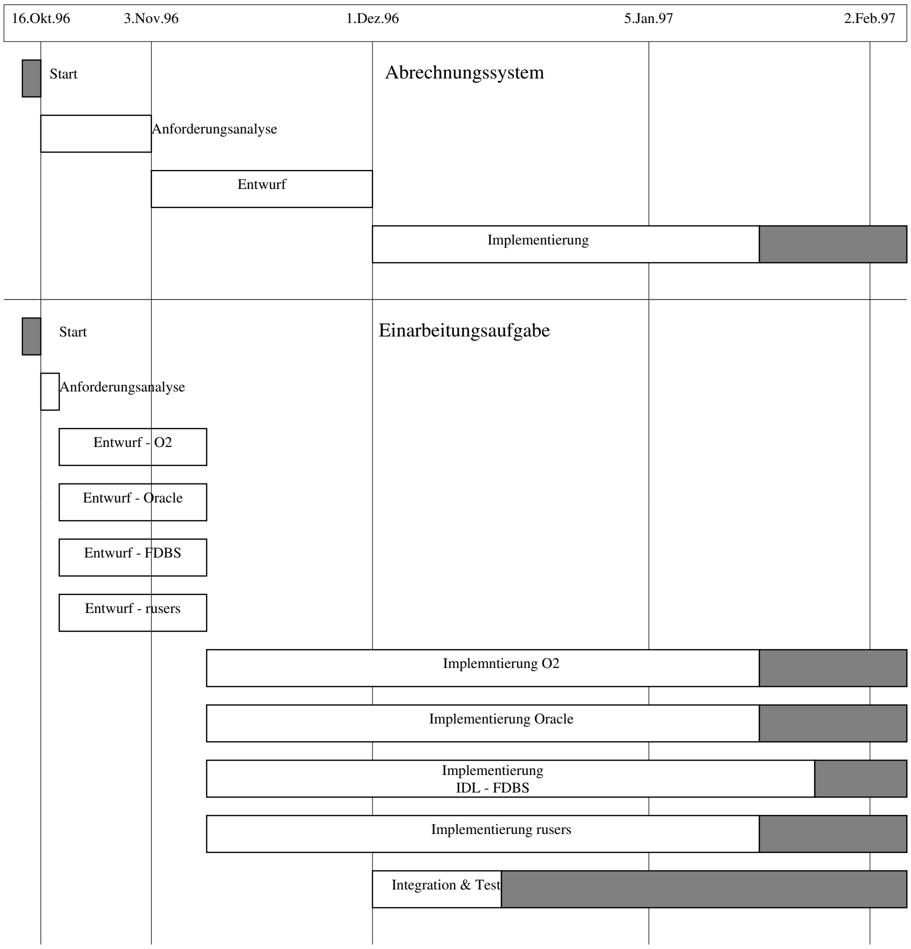


Abbildung 3.3: Das Projekt–Gantt–Chart im Januar 1997.

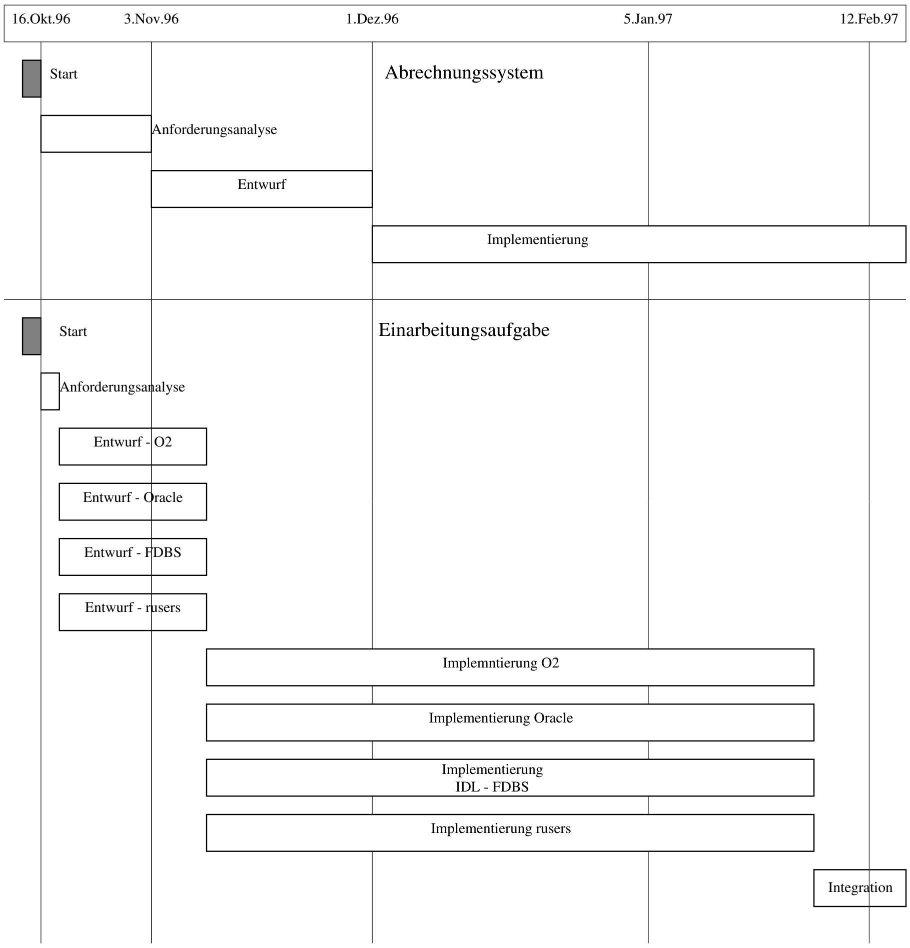


Abbildung 3.4: Das Projekt–Gantt–Chart zum Zwischenbericht.

Bis zum 1.2.97 wurde dann ein lauffähiges System implementiert, welches noch die föderierten Abbildungen hart-kodiert enthälten und zum Ende des ersten Semesters wurde die Einführungsaufgabe abgeschlossen. Knackpunkt war nach wie vor das FDBS.

3.2.2 Zum Zwischenbericht im Februar

Zum Ende des ersten Semester und zum Zeitpunkt des Zwischenberichtes ergibt sich nun das Gantt–Chart aus Abbildung 3.4. Die endgültige Ende-Deadline wurde nun auf den 12.2.97 gelegt, also dem Ende des ersten Semesters. Die Applikationen zu O_2, *Oracle* und rusers laufen integriert und das Abrechnungssystem wird vermutlich auch bis zu diesem Zeitpunkt fertiggestellt sein.

Das PG–Ziel, Beendigung der Einführungsaufgabe und Fertigstellung des Abrechnungssystems, wird also im groben eingehalten werden können. Natürlich wird es noch einige Weiterentwicklungen an dem Abrechnungssystem geben.

Alles in allem ist die Planung für das erste Semester zur Zufriedenheit aller Beteiligten abgelaufen. Theoretische Überlegungen für das FDBS im zweiten Semester sind angelaufen.

Teil IV

Die Einarbeitungsaufgabe

Kapitel 4

Die Aufgabenstellung

Verfasser: Willi Hasselbring, Klaus Alfert

Bevor die eigentliche Aufgabe — die Erstellung eines föderierten objektorientierten Krankenhausinformationssystem — gelöst wird, soll eine kleine Einarbeitungsaufgabe bearbeitet werden. Üblicherweise setzt man neue Techniken ja erst beim wiederholtem Mal richtig ein.

Aufgabe: Es soll ein Werkzeug zur Erfassung der Login-Daten der PG-Teilnehmer konstruiert werden. Ein Deamon-Prozeß soll in regelmäßigen (konfigurierbaren) Abständen auf einer konfigurierbaren Liste von Rechnern die aktuell eingeloggten PG-Teilnehmer erfassen und entsprechende Informationen persistent speichern.

Die benötigte Information wird aus der *Datenbank* "`rusers -l`" ausgelesen und soll durch ein Föderierungssystem in entsprechende Oracle- und O_2-Datenbanken gespeichert werden, wie in Abbildung 4.1 dargestellt.

Die einzelnen Applikationen sollen jeweils autonom sein. Keine Applikation darf aufgrund des Fehlens einer anderen Applikation nicht lauffähig sein. Die einzelnen Applikationen können auf verschiedenen Rechnern verteilt sein. Die Kommunikation zwischen den Applikationen, d.h. zwischen dem Föderierungssystem und den Applikationen soll mittels CORBA geschehen. Die Grobarchitektur für die Einarbeitungsaufgabe ist in Abbildung 4.2 dargestellt.

Dazu sind u.a. folgende Aufgaben zu erledigen:

- Definition eines Datenmodells für "rusers -l"
- Entwurf dazu geeigneter Schemata und Benutzungsschnittstellen für Oracle- und O_2
- Entwurf eines Föderierungssystems. Damit es als Prototyp für die eigentliche Anwendung dienen kann, ist wesentlich, daß das Föderierungssystem generisch und durch geeignete Import/Export-Tabellen steuerbar ist.

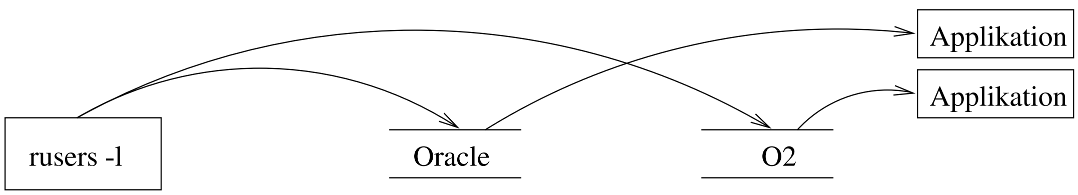


Abbildung 4.1: Grobübersicht über die Einarbeitungsaufgabe.

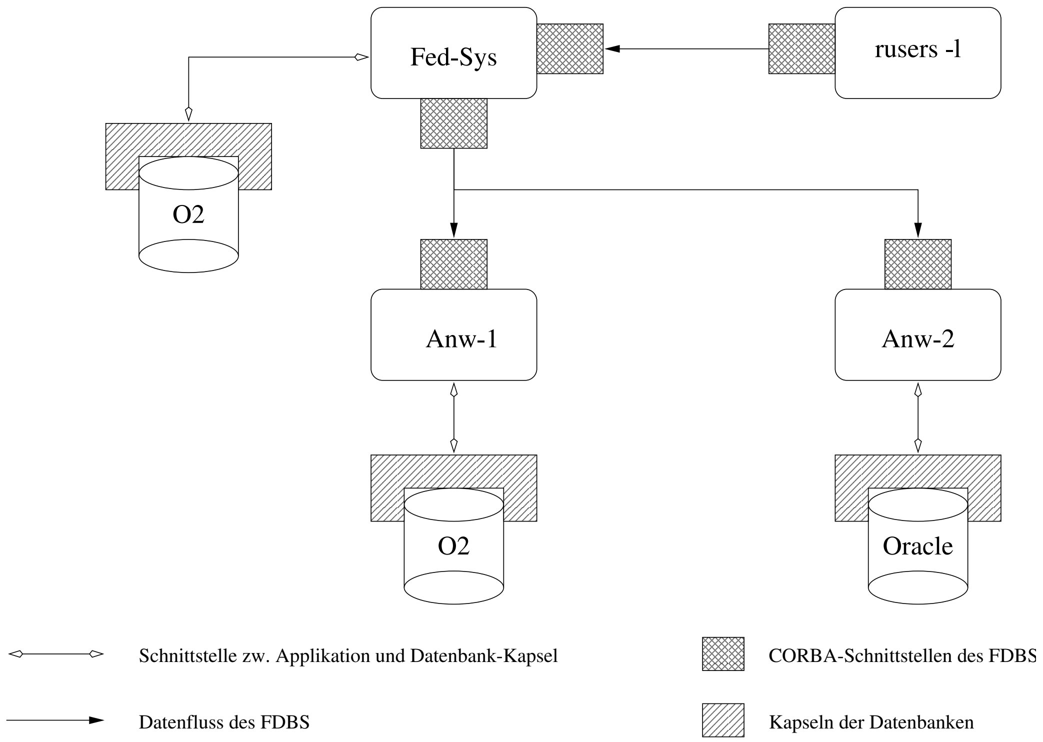


Abbildung 4.2: Grobarchitektur für die Einarbeitungsaufgabe.

Als Werkzeuge sollen verwendet werden:

- OMT mit *Rational Rose*, *xomt* oder *OOD* zur Spezifikation
- CORBA-IDL zur Beschreibung der Schnittstellen der einzelnen Komponenten zum Föderierungssystem sowie auch zur deren Realisierung
- Realisierung der Anwendungssysteme unter ORACLE mit FORMS sowie unter O_2-Look
- Sniff als Entwicklungsumgebung mit CVS oder RCS als Versionsverwaltung.

Die Implementierung ist nicht zu vergessen. Die Arbeit soll in Teilgruppen erledigt werden:

- *Einkapselung* von “rusers -l” und FDBS-Interface, 1 Person
- Oracle-Datenbank (Anwendung, Kapsel und FDBS-Interface), 2 Personen
- O_2-Datenbank (Anwendung, Kapsel und FDBS-Interface), 2 Personen
- Föderierungssystem (O_2, FDBS-Interace), 2 Personen
- Konfigurations-Manager (Sniff, Makefiles, Versions-Verwaltung, Koordination), 1 Person

Die Schnittstellen müssen geeignet definiert werden.
Geplante Deadlines:

1. Analyse (was genau soll's können): Auf dem Seminar in Haus Nordhelle
2. Entwurf (wie soll's gehen): 9.11.96 (3 Wochen)
3. Implementierung: 1.12.96 (3 Wochen)
4. Test: bis zur Scheinvergabe

Kapitel 5

Das Föderierungssystem

Verfasser: Peter Ziesche

5.1 Die Architektur

Das System, wie es im Rahmen der Projektgruppe entwickelt werden soll, besteht aus mehreren Komponenten. Den Kern bildet das Föderierungssystem (*FDBS*, siehe Kapitel 5.1.1). Für die an der Föderation beteiligten lokalen Datenbanken werden außerdem Adapter benötigt (siehe Kapitel 5.1.2).

Die Kommunikation zwischen diesen Komponenten, d.h. im wesentlichen der Austausch von Daten, wird über CORBA abgewickelt. Die Struktur der auszutauschenden Daten wird in allen Komponenten grundsätzlich die gleiche sein. Die CORBA-spezifischen Mechanismen sollen daher durch ein Kommunikationsinterface (*comi*) gekapselt werden, das den Komponenten eine einfach zu benutzende Schnittstelle bietet.

Um dies zu erreichen, werden alle Komponenten eine Schichtenarchitektur haben (siehe Abbildung 5.1). Die jeweils oberste Schicht bildet das *comi*.

5.1.1 Das Föderierungssystem

Das *FDBS* kontrolliert die Abhängigkeiten zwischen den an der Föderation beteiligten lokalen Datenbanken. Für jede Datenbank werden die Exportschemata gespeichert und miteinander verknüpft. Die Funktionsweise ergibt sich aus der folgenden Beschreibung der Schichten:

Kommunikations-Interface:
Siehe Kapitel 5.2.

Operationsverarbeitung:
Hier werden die von einer lokalen Datenbank eingehenden Operationen verarbeitet. Aus den Abhängigkeiten zwischen den Exportschemata werden resultierende Operationen für andere Datenbanken erzeugt und an die Adapter der lokalen Datenbanken geschickt.

Für die Einarbeitungsaufgabe stehen die beteiligten Datenbanken, die Exportschemata und die lokalen Datenmodelle bereits fest. Die Operationsverarbeitung wird daher fest codiert.

Föderierungs-Datenbank:
Die Föderierungs-Datenbank enthält die Exportschemata der angeschlossenen Datenbanken und verwaltet die Abhängigkeiten zwischen ihnen. Die dazu benötigten Daten werden in einer O_2-Datenbank gespeichert. Die Föderierungs-Datenbank stellt den Kern des föderierten Systems dar. Ihre Struktur wird deshalb in einem eigenen Kapitel genau beschrieben.

In der Einarbeitungsaufgabe wird das föderierte Schema nicht implementiert.

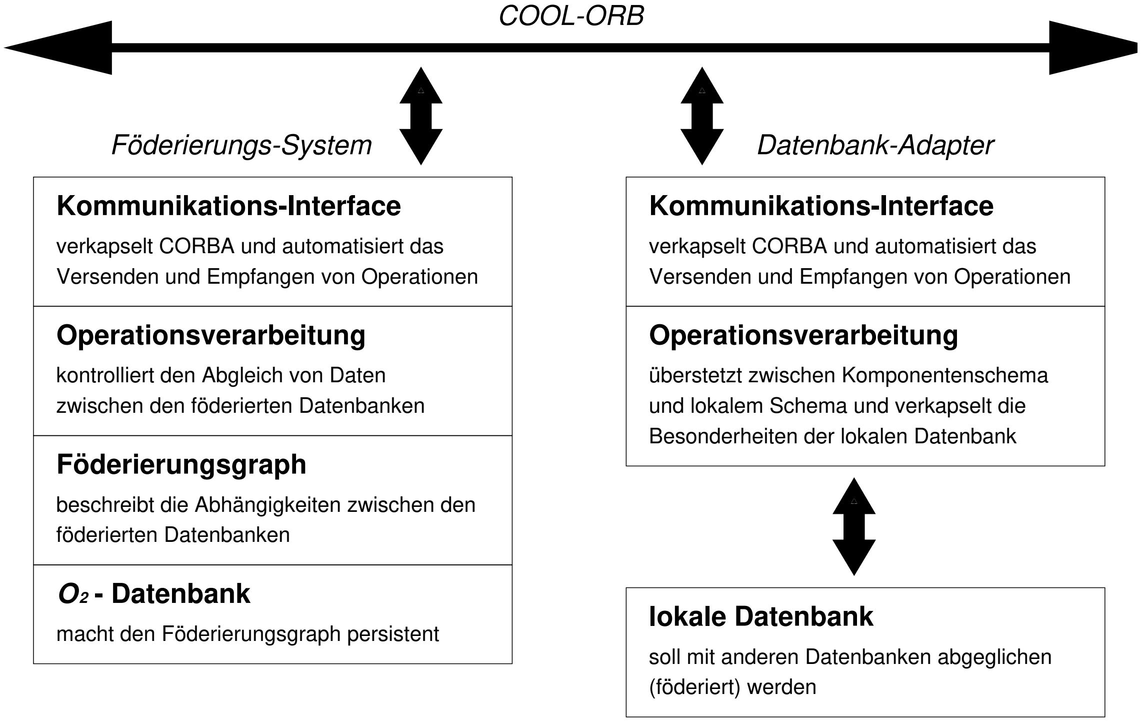


Abbildung 5.1: Architektur des *FDBS*

5.1.2 Die Datenbank-Adapter

Die Datenbank-Adapter verbinden die lokalen Datenbanken mit dem Föderierungssystem. Sie übernehmen auch die Konvertierungen zwischen dem kanonischen und dem lokalen Datenmodell. Datenbankoperationen (siehe Kapitel 5.1.3) werden auf die besonderen Gegebenheiten der jeweiligen lokalen Datenbank abgebildet.

Kommunikations-Interface:
Siehe Kapitel 5.2.

Operationsverarbeitung:
In diesem Teil des Adapters wird das Komponentenschema, bzw. das Exportschema auf das lokale Datenbankschema abgebildet. Datenbankoperationen vom/zum *FDBS* werden in Operationen der lokalen Datenbank konvertiert.

In der Einarbeitungsaufgabe wird diese Abbildung fest codiert sein.

Lokale Datenbank:
Die an der Föderation beteiligten, lokalen Datenbanken sollen soweit wie möglich autonom bleiben. Sie können sich sowohl durch das verwendete Datenbankmanagementsystem, als auch durch das Datenmodell unterscheiden.

5.1.3 Die Datenbankoperationen

Die Aufgabe des föderierten Systems besteht im wesentlichen aus dem Austausch von Daten zwischen lokalen Datenbanken. Ein Datentransfer wird durch eine Änderung in einer lokalen Datenbank ausgelöst. Eine solche Änderung erfolgt durch Datenbankoperationen, z.B. eine Update- oder Insert-Operation.

Der Datenaustausch zwischen den lokalen Datenbanken (über das *FDBS*) wird daher auf der Basis solcher Operationen erfolgen. Ein Datenbankadapter sendet eine Operation zusammen mit den benötigten Daten an das *FDBS*, das daraus Operationen für andere Datenbanken generiert. Das Kommunikationsinterface sorgt für eine einfache Handhabung der Operationen. *In der Einarbeitungsaufgabe werden nur Insert-Operationen vorkommen.*

5.2 Das Kommunikations-Interface

5.2.1 Architektur

Um den Datenaustausch für die Komponenten des föderierten Systems so einfach wie möglich zu gestalten, verwendet das *comi* zwei grundsätzliche Mechanismen:

Operationsobjekte:
Die Datenbank-Operationen (siehe Kapitel 5.1.3) werden in C++-Objekte gekapselt. Die benötigten Daten können in diese Objekte bequem hineingeschrieben und wieder ausgelesen werden. Für jede Art von Operation wird eine eigene Klasse gebildet. Alle Operations-Klassen erben von der abstrakten Klasse `COperation`, die einige, für alle Operationen benötigte, Routinen deklariert.

Operations-Handler:
Zur Verarbeitung einer Operation sind je nach Komponente und Art der Operation unterschiedliche Aufgaben zu erledigen. Eine Komponente, die Operationen eines bestimmten Typs empfangen will, muß daher für jeden Typ einen Handler bereitstellen. Ein Handler wird in Form einer C++-Klasse implementiert. Wenn das *comi* eine Operation empfängt, instanziiert es mit Hilfe einer *abstrakten Fabrik* [G+95] einen Handler und beauftragt ihn mit der Verarbeitung der Operation.

Für den Versand und den Empfang von Operationen stehen je eine Klasse zur Verfügung. `CSender` (siehe 5.2.2) versendet eine Operation an eine bestimmte Komponente des föderierten Systems, `CReceiver` (siehe 5.2.2) empfängt Operationen und ruft die entsprechenden Handler auf. Der Benutzer braucht sich dadurch keine Gedanken über den Kontrollfluß innerhalb des *comi* zu machen

Zur Benutzung des *comi* müssen einige Klassen durch den Benutzer geschrieben werden. Dabei handelt es sich um abgeleitete Klassen, die die komponentenspezifische Funktionalität in das Interface einbinden.

5.2.2 Die Klassen des Kommunikations-Interfaces

Die folgenden Klassen sind Bestandteil des Interfaces. Sie brauchen vom Benutzer nicht verändert zu werden und können über die Header-Datei `comi.H` eingebunden werden:

`CHandlerFactory`	Abschnitt 5.2.2, Seite 120
`CInsert`	Abschnitt 5.2.2, Seite 122
`CInsertHandler`	Abschnitt 5.2.2, Seite 121
`CReceiver`	Abschnitt 5.2.2, Seite 119
`CSender`	Abschnitt 5.2.2, Seite 119

Der Benutzer muß (je nach Anwendung nicht alle) die folgenden Klassen zur Verfügung stellen, um das Interface nutzen zu können, wobei *Concrete* als Platzhalter für einen anwendungsspezifischen Namen steht:

`C`*Concrete*`HandlerFactory`
`C`*Concrete*`InsertHandler`
`C`*Concrete*`SchemaOpHandler`

In Kapitel 5.2.3 finden sich Beispiel-Quelltexte zur Nutzung des *comi*.

Zusätzlich existieren noch einige weitere Klassen, die zur internen Funktionalität des *comi* beitragen. Sie sind für die Benutzung jedoch nicht wichtig und werden daher nicht beschrieben.

Die Klasse `CSender`

Beschreibung

Die Klasse dient als Schnittstelle für den Versand von Operationen. Jede Instanz dieser Klasse versendet Operationen an genau eine Komponente des föderierten Systems. Bevor zum ersten Mal eine Operation erzeugt wird, muß ein Objekt mit `Init()` initialisiert werden.

Mit den Create*Operation*-Methoden können die Operationen erzeugt und anschließend mit Daten gefüllt werden. Der Versand erfolgt durch den Aufruf von `Send()`. Der Benutzer ist für die Zerstörung des Objektes zustandig.

Alle Methoden, deren Rückgabewerte vom Typ `bool` sind, liefern im Falle der erfolgreichen Ausführung `true` als Ergebnis, sonst `false`. Nähere Informationen über den Fehler können anschließend über die Methode `GetLastError()` ermittelt werden. Ein von diesem Standard abweichendes Verhalten geht aus der Beschreibung einer Methode hervor.

Methoden:

`bool Init( const string i_sReceiverName, int argc, char* argv[] )`
: Initialisiert ein Sender-Objekt und legt den Empfänger für den Versand von Operationen fest. Die Kommandozeile der Komponente muß mit übergeben werden.

`bool Terminate()`
: Sorgt für ein ordnungsgemäßes Beenden des Senders.

`int GetLastError()`
: Die Routine liefert eine Fehlernummer, die dem Aufrufer Aufschluß über die Art eines Fehlers gibt, wenn eine Methode `false` geliefert hat. *In der Einarbeitungsaufgabe ist diese Methode noch nicht implementiert.*

`CInsert* CreateInsert()`
: Liefert einen Zeiger auf ein `CInsert` Objekt. Der Zeiger ist `NULL`, wenn kein Objekt erzeugt werden konnte.

`bool Send( COperation* i_poOperation )`
: Der Aufruf von Send versendet eine Operation an den bei der Initialisierung bestimmten Empfänger.

`bool Discard( COperation* i_poOperation )`
: Wenn eine Operation erzeugt wurde, jedoch aus irgendwelchen Gründen nicht versendet werden kann oder soll, so kann sie mit dem Aufruf von `Discard` wieder zurückgenommen werden.

Die Klasse `CReceiver`

Beschreibung:

Die Klasse dient als Schnittstelle für den Empfang von Operationen.

Eine Anwendung kann nur eine einzige Instanz dieser Klasse erzeugen (Entwurfsmuster Singleton [G+95]). Vor der Benutzung muß diese Instanz mit `Init()` initialisiert werden.

Anschließend wird mit `WaitForOperation` eine Warteschleife betreten. Von dort aus werden empfangene Operationen automatisch verarbeitet, indem je nach Typ der Operation ein entsprechender Handler erzeugt wird, der dann die Weiterverarbeitung übernimmt. *In der Einarbeitungsaufgabe kehrt diese Routine nie zurück. Die entsprechende Komponente muß manuell über die Shell terminiert werden.*

Alle Methoden, deren Rückgabewerte vom Typ `bool` sind, liefern im Falle der erfolgreichen Ausführung `true` als Ergebnis, sonst `false`. Nähere Informationen über den Fehler können anschließend über die Methode `GetLastError()` ermittelt werden. Ein von diesem Standard abweichendes Verhalten geht aus der Beschreibung einer Methode hervor.

Methoden:

`static CReceiver* Instance()`
: Die (*Klassen-*)Methode liefert einen Zeiger auf die einzige Instanz der Klasse `CReceiver`. Da sie statisch deklariert ist, muß der Aufruf die Form `CReceiver* poRec = CReceiver::Instance()` haben.

`static Destroy()`
: Da ein Objekt dieser Klasse vom Benutzer auf direktem Weg weder erzeugt noch zerstört werden kann, ist an Stelle eines `delete`-Aufrufes der Aufruf `CReceiver::Destroy()` erforderlich.

`bool Init( const string i_sReceiverName, CHandlerFactory* i_poFactory, int argc, char* argv[] )`
: Die Methode initialisiert den Empfänger. Der Parameter `i_sReceiverName` legt den Namen fest, unter dem der Empfänger im System angesprochen werden kann. In `i_poFactory` wird ein Zeiger auf das für die Erzeugung von Operationen-Handlern verantwortliche Fabrikobjekt übergeben (siehe auch Beschreibung in Abschnitt 5.2.2). Die Kommandozeile der Komponente muß mit übergeben werden.

`bool Terminate()`
: Sorgt für ein ordnungsgemäßes Beenden des Empfängers. *In der Einarbeitungsaufgabe ist diese Routine nicht implementiert.*

`int GetLastError()`
: Die Routine liefert eine Fehlernummer, die dem Aufrufer Aufschluß über die Art eines Fehlers gibt, wenn eine Methode `false` geliefert hat. *In der Einarbeitungsaufgabe ist diese Methode noch nicht implementiert.*

`void WaitForOperation()`
: Mit dieser Methode wird in eine Endlosschleife gesprungen, in der Operationen empfangen und automatisch verarbeitet werden (siehe auch Beschreibung von `CReceiver`). *In der Einarbeitungsaufgabe kehrt diese Routine nie zurück.*

Die Klasse `CHandlerFactory`

Beschreibung:

Diese *abstrakte* Klasse dient zur automatischen Erzeugung von Handlern für Operationen. Eine Anwendung muß in einer abgeleiteten Klasse die Methoden implementieren, in der Regel handelt es sich um eine einzige Anweisung je Methode. Der diesem Konzept zu Grunde liegende Gedanke ist das Entwurfsmuster *abstrakte Fabrik* [G+95].

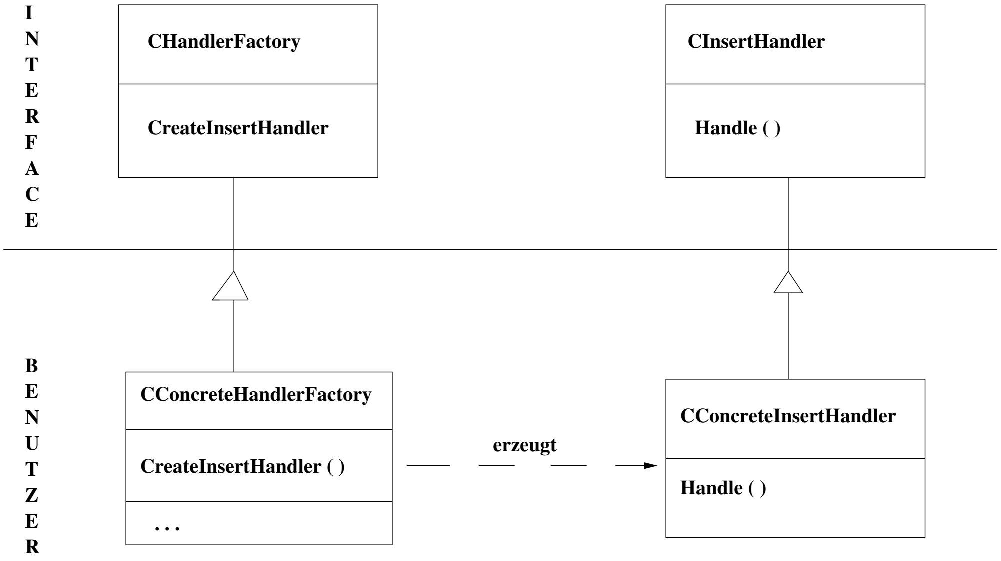


Abbildung 5.2: Prinzip der Operation-Handler.

Eine Anwendung muß von der abgeleiteten Klasse ein Objekt erzeugen und an die Initialisierungsfunktion der Klasse `CReceiver` übergeben. Weitere Informationen finden sich dort (Abschnitt 5.2.2, Seite 119).

Methoden:

Die folgenden Methoden erzeugen (in der vom Benutzer abgeleiteten Unterklasse von `CHandlerFactory`) jeweils Objekt der entsprechenden Klasse und liefern einen Zeiger darauf zurück.

Die Implementierung könnte wie folgt aussehen:

```
CInsertHandler* CConcreteHandlerFactory::CreateInsertHandler()
{
   return (CInsertHandler*) new CConcreteInsertHandler;
}
```

Die Klasse `CInsertHandler`

Beschreibung:

Die abstrakte Basisklasse dient für das *comi* als Schnittstelle für die Verarbeitung von Insert Operationen. Ein Datenbanktreiber muß in einer von `CInsertHandler` abgeleiteten Klasse die Methode `Handle()` überschreiben. Sie wird aufgerufen, wenn eine `Insert`-Operation empfangen wurde und verarbeitet werden soll.

Die Implementierung des Handlers muß die Daten des einzufügenden Objektes aus dem als Parameter übergebenen `CInsert`-Objekt auslesen. Anschließend muß das Objekt in die lokale Datenbank eingefügt und ein Schlüssel ermittelt werden. Dieser ist vom Handler in das Attribut `sLoid` des `CInsert`-Objektes einzutragen. Dadurch kann das *comi* diesen Wert an das *FDBS* melden.

Ein Handler-Objekt dient immer nur zur Bearbeitung einer einzigen Operation, d.h. es wird nach Verrichtung seiner Aufgabe wieder zerstört. Es sollten also innerhalb einer Komponente keine Zeiger auf Handler-Objekte gespeichert werden.

Methoden

`bool Handle( CInsert* poOp )`
: Diese Routine ist der Eintrittspunkt in das Handler-Objekt. Die Funktion muß `TRUE` liefern, wenn das Einfügen des Objektes erfolgreich war, die LOID also gültig ist. Schlägt die Operation fehl, muß `FALSE` zurückgegeben werden.

Die Klasse `COperation`

Beschreibung:

Dies ist die abstrakte Basisklasse für Operationen. Alle Operationen auf Datenbanken werden in Objekte verkapselt (siehe Abschnitt 5.1.3, Seite 117).

Methoden:

`bool IsComplete()`
: Mit dieser Funktion kann überprüft werden, ob alle Attribute eines Operations-Objektes mit Daten belegt wurden, die Operation also versandt werden kann. Eine inhaltliche Prüfung der Attributwerte auf Konsistenz findet *nicht* statt.

Die Klasse `CInsert`

Beschreibung:

Wenn in einer Datenbank ein neues Objekt eingefügt wurde, muß dies dem *FDBS* mitgeteilt werden. Dies wird mit der Insert-Operation erreicht.

Die Bedeutung des Attributes `sLoid` unterscheidet sich darin, ob ein Objekt eine zu versendende oder gerade empfangene Operation darstellt. Im ersten Fall ist die LOID der Schlüssel des Objektes in der Datenbank, in der es eingefügt wurde. Dieser Schlüssel muß dem *FDBS* mitgeteilt werden.

Handelt es sich jedoch um eine empfangene Operation, so teilt das *FDBS* dem Datenbanktreiber mit, daß ein neues Objekt in "seine" Datenbank einzufügen ist. Die LOID steht also zu diesem Zeitpunkt noch nicht fest. Der Datenbanktreiber — genauer gesagt: der zuständige Handler — muß ein neues Objekt einfügen und die neu vergebene LOID in das Attribut sLoid eintragen. Sie dient dann als Rückgabewert der Operation an das *FDBS*.

Der Datenbanktreiber muß sicherstellen, daß alle im Exportschema festgelegten Attribute der Klasse mit Werten belegt werden.

Attribute:

`string sSchema:`
: Der Name des Komponentenschemas, in das ein Objekt eingefügt werden soll, bzw. eingefügt wurde.

`string sClass:`
: Der Name der Klasse, von der ein Objekt instanziiert werden soll, bzw. instanziiert worden ist.

`string sLoid:`
: Der Schlüssel des einzufügenden bzw. eingefügten Objektes in der Datenbank.

Methoden:

`bool AddAttribute( const string i_sName, const string i_sValue)`
: Der erste Parameter gibt den Namen des Attributs an, der im Komponentenschema definiert wurde, der zweite bestimmt den Wert, den dieses Attribut erhalten hat.

`bool GetValue( const string i_sName, string* o_sValue )`
: Diese Methode liefert zu einem Attribut (spezifiziert durch den Namen des Attributes) den Wert.

5.2.3 Benutzung des Interfaces

Versenden von Operationen:

Das Hauptprogramm sollte in etwa die folgende Form haben:

```
#include "comi.H"

int main( int argc, char* argv[] )
{
   // Sender erzeugen und initialisieren
   CSender* poFdbs = new CSender;
   poFdbs->Init( "FOKIS_FDBS", argc, argv );

   // Eigener Initialisierungscode
   ...

   // Operation erzeugen
   CInsert* poInsOp;
   poInsOp = poFdbs->CreateInsert();

   // Operation mit Daten fuellen
   poInsOp->Schema = "Datenbankname";
   poInsOp->Class  = ... ;
   poInsOp->LOID   = ... ;
   poInsOp->AddAttribut( Name, Wert );
   ...
   // Operation versenden
   poFdbs->Send( poInsOp );

   // Programm beenden
   poFdbs->Terminate();
   delete poFdbs;
   delete poInsOp;

   return 0;
} //main
```

Empfangen von Operationen:

Das Hauptprogramm sollte in etwa die folgende Form haben:

```
#include "comi.H"
```

```
//Fabrikobjekt fuer Handler anlegen
COracleHandlerFactory g_oFactory;

int main( int argc, char* argv[] )
{
   // Empfaenger erzeugen und initialisieren
   CReceiver* poReceiver = CReceiver::Instance();
   poReceiver->Init( "ORACLE_DB",
                     (CHandlerFactory*) &g_oFactory,
                     argc, argv );

   // Eigener Initialisierungscode
   ...

   // Endlosschleife fuer Empfang
   poReceiver->WaitForOperation();

   // Programm beenden
   poReceiver->Terminate();
   CReceiver::Destroy();

   return 0;
} //main
```

Eine weitere Klasse, die der Benutzer selbst schreiben muß, ist die Fabrik für die Operationen-Handler. Für obiges Hauptprogramm könnte diese Fabrik etwa so aussehen:

```
class COracleHandlerFactory : public CHandlerFactory
{
   CInsertHandler* CreateInsertHandler()
   {
      return (CInsertHandler*) new COracleInsertHandler;
   }

   // Handler fuer weitere Operationen
   ...
};
```

Schließlich muß noch der eigentliche Handler für die Operation geschrieben werden. Er enthält den Code, der benötigt wird, um eine Operation effektiv auszuführen.

```
class COracleInsertHandler : public CInsertHandler
{
   bool HandleInsert( CInsert* poInsert )
   {
      // Der Code, um die Operation zu verarbeiten
   }

   // eventuell benoetigte lokale Routinen und Variablen
   ...
};
```

5.3 Starten des Föderierungssystems

Wie in Kapitel 5.1.1 beschrieben, ist die Operationsverarbeitung innerhalb des Föderierungssystems fest codiert. Auch das dynamische Konfigurieren des Systems ist erst im Planungsstadium (siehe Kapitel 10.1). Deshalb müsen zur Inbetriebnahme des Föderierungssystems bestimmte Prozesse in einer festgelegten Reihenfolge gestartet werden.

Der `rusers`-Daemon, die O_2- und die Oracle-Anwendung arbeiten unabhängig vom Föderierungssystem. Sie können also auch laufen, wenn das *FDBS* nicht aktiv ist.

Zuerst müssen die Datenbankadapter gestartet werden, d.h. es müssen CORBA-Server mit den Namen `O2`, `ORACLE` und `TESTMONITOR` beim Object-Request-Broker angemeldet werden. Da sie untereinander unabhängig sind, ist die Start-Reihenfolge beliebig. Der Testmonitor ist ein Adapter, der die empfangenen Daten auf dem Bildschirm ausgibt. Er dient zur Kontrolle. Für die O_2- und die *Oracle*-Datenbank stehen Testadapter zur Verfügung, die an Stelle der richtigen Adapter zu Testzwecken laufen können. Sie entsprechen in ihrer Funktionalität dem Testmonitor, machen also nur Ausgaben auf dem Bildschirm.

Anschließend kann das Föderierungssystem (CORBA-Name: `FDBS`) gestartet werden.

Zum Schluß wird der `rusers`-Agent gestartet. Voraussetzung dafür ist, das der `rusers`-Daemon bereits im Hintergrund läuft.

Das Herunterfahren des Systems muß in umgekehrter Reihenfolge erfolgen. Zuerst ist also der `rusers`-Agent zu beenden, anschließend das *FDBS* und zum Schluß die Datenbankadapter.

Kapitel 6

Das rusers-System

Verfasser: Djamel Kheldoun

6.1 Analyse von rusers

Der Aufruf von `rusers -l` liefert die folgenden Informationen über alle eingeloggten Benutzer:

Benutzername: Login-Name des Benutzers

Rechnername: Name des Rechners, auf dem der Benutzer eingeloggt ist.

Terminal: Terminal-Name

Login_Datum: Seit wann (Datum) ist der Benutzer eingeloggt.

Login_Zeit: Seit wann (Uhrzeit) ist der Benutzer eingeloggt.

Idle_Zeit: Zeitdauer, die der Benutzer nicht mehr gearbeitet hat, aber eingeloggt war.

Login_Rechnername: Der Rechnername, von dem aus sich der Benutzer eingeloggt hat.

```
Beispiel:
  frenzel      moskau:ttyp3              Nov 20 10:08          (peking)
  junker       udo:ttyp1                 Nov 19 09:13       16 (david)
```

In der Abbildung 6.1 ist die OMT-Modellierung der Ausgabe von `rusers -l` dargestellt.

6.2 Architektur des rusers-Systemes

Entsprechend der Konfigurationsdatei werden nur die benötigten Informationen über die PG-Teilnehmer selektiert und in die `Daten_Datei` gespeichert. Diese gespeicherten Daten werden vom Agent-Deamon gelesen und an das FDBS weitergegeben. In der Abbildung 6.2 ist die Architektur von `rusers` dargestellt.

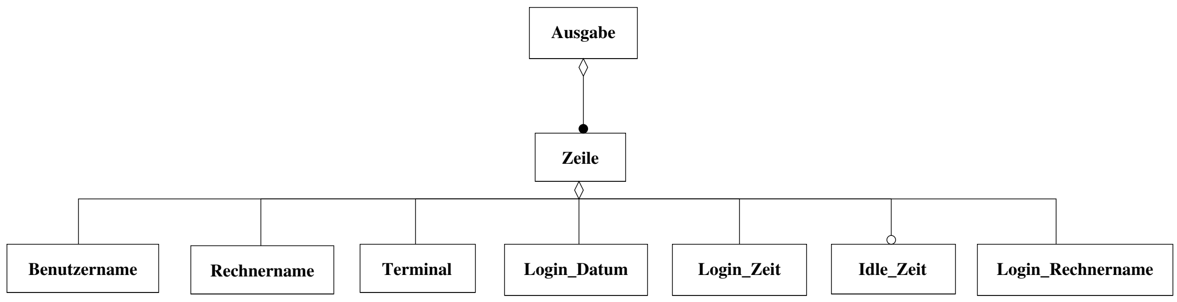


Abbildung 6.1: Die OMT-Modellierung der Ausgabe von `rusers -l`.

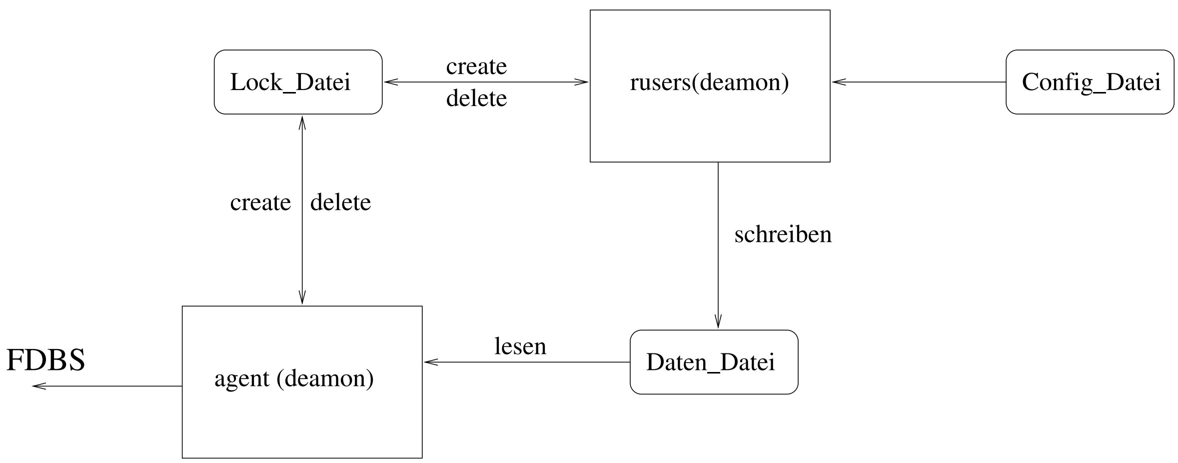


Abbildung 6.2: Die Architektur von `rusers`.

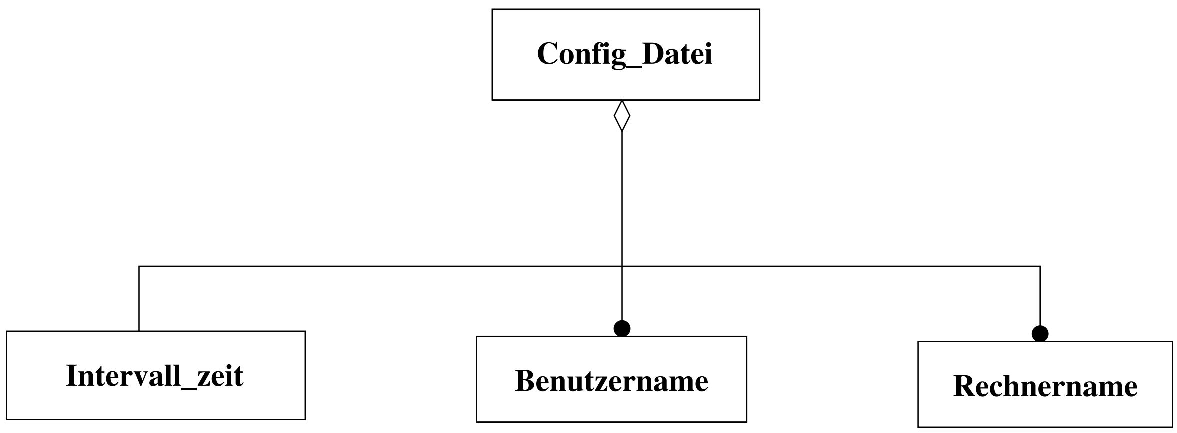


Abbildung 6.3: Das OMT-Objektmodell von `Config_Datei`.

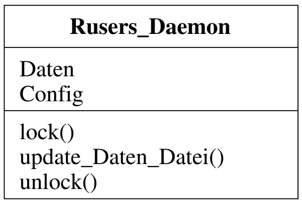


Abbildung 6.4: Das OMT-Objektmodell von `Rusers-Deamon`.

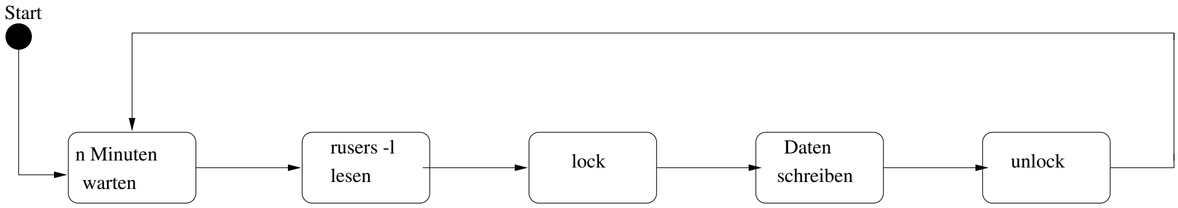


Abbildung 6.5: Das Zustandsübergangsdiagramm von `Rusers-Deamon`.

6.3 Die Konfigurationdatei:

In der Konfigurationdatei `Config_Datei` werden die `Benutzernamen`, `Rechnernamen` und `Intervall_Zeit` gespeichert. In der Abbildung 6.3 ist die OMT-Modellierung der `Config_Datei` dargestellt.

Die erste Zeile der `Config_Datei` enthält die `Intervall_Zeit` (in Minuten),die zweite Zeile die `Benutzernamen` und die dritte Zeile die `Rechnernamen`.
Ein Beispiel:

```
2
Otto Markus
london tokio kreta wien madrid
```

6.4 Der Rusers-Deamon

`Rusers` ist ein Prozeß, der als Deamon laufen soll. Er aktualisiert regelmäßig die `Daten_Datei` (z.B alle 2 Minuten). In der Abbildung 6.4 bzw. 6.5 ist das OMT-Objektmodell bzw. OMT-Dynamikmodell von rusers dargestellt.

6.5 Die Daten-Datei

Die Struktur der `Daten_Datei` wird durch die OMT-Modellierung In Abbildung 6.6 dargestellt.

Ein Beispiel:

```
Otto    (10:25   08.11.96) (madrid 8:00 08.11.96) (london 9:30 08.11.96)
Markus  (09:40   08.11.96)   (kreta 8:30  07.11.96)
```

6.6 Die Lock-Datei

Vor dem Schreiben bzw. Lesen `der Daten_Datei` wird von `Rusers` bzw. `Agent` eine `Lock_Datei` erzeugt, die danach wieder gelöscht wird.

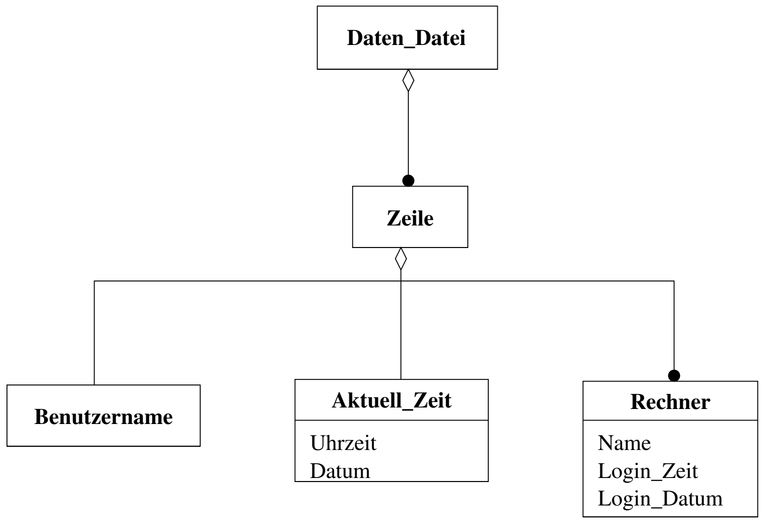


Abbildung 6.6: Das Objektmodell von `Daten_Datei`.

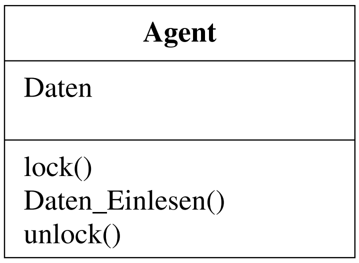


Abbildung 6.7: Das OMT-Objektmodell von `Agent-Deamon`.

6.7 Der Agent

`Agent` soll als Deamon laufen und liest regelmäßig die aktualisierten Daten aus der `Daten_Datei`. Die gelesenen Daten werden an die IDL-Schnittstelle des FDBS geschickt. In den Abbildungen 6.7 bzw. 6.8 ist Objekt- bzw. Dynamikmodell dargestellt.

6.8 Struktur der Daten (in C++):

```
class CDaten {
    // Die folgenden Daten werden an das FDBS geschickt.
    RWCString   sBenutzername; // Login-Name des Benutzers
    RWCString   sAktuellZeit;  // System-Zeit
    RWCString   sAktuellDatum; // System-datum

    RWCString   sRechnername;  // Der Rechner, an dem der Benutzer arbeitet
    RWCString   sLoginZeit;  // wann(Uhrzeit) hat sich der Benutzer eingeloggt
    RWCString   sLogindatum; //  wann(Datum) hat sich der Benutzer eingeloggt
 };
```

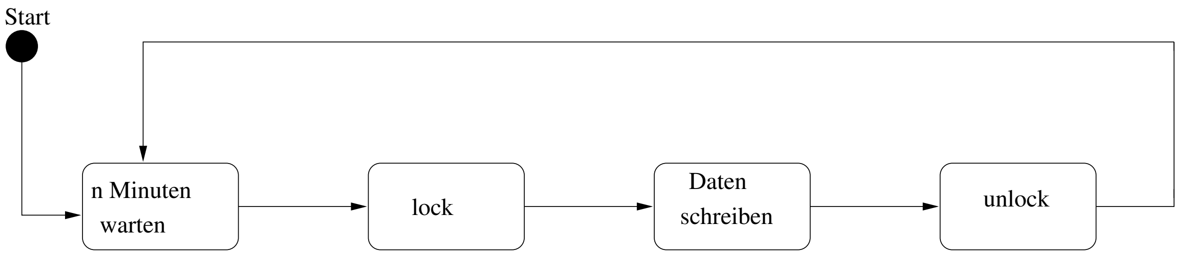


Abbildung 6.8: Das Zustandsübergangsdiagramm von `Agent_Deamon`.

Kapitel 7

Das O_2-System

Verfasser: Sven Gerding, Ulf Radmacher

7.1 Die Analyse

In der O_2-Datenbank sollen die von *rusers* gelieferten Informationen über die Benutzer gespeichert werden. Diese beinhalten Benutzername, Login-Zeit, aktuelle Zeit, Datum und Rechner, auf dem der Benutzer eingeloggt ist (siehe auch Abbildung 7.1).

7.2 Der Entwurf

Der Entwurf für die Datenspeicherung (Abbildung 7.2) mit O_2 umfaßt sieben Klassen und wurde mit *Rational Rose* [Cla95] erstellt. Die Klassen werden in der Booch-Notation [Boo94] durch Wolken repräsentiert, wobei der Klassenname sich über der Trennlinie befindet, die zur Klasse gehörenden Attribute und Methoden unterhalb. In der Abbildung 7.2 gibt es neben den normalen Klassen, die durch einfache Wolken repräsentiert werden, zwei weitere Arten von Klassen: Eine Parametrisierte Klasse und instantiierte Klassen. Die parametrisierte Klasse `Standardliste` wird durch eine Wolke mit einem gestrichelten Rechteck dargestellt. Eine parametrisierte Klasse (auch generische Klasse genannt) ist eine Klasse, die als Schablone für andere Klassen dient - eine Schablone, die durch andere Klassen, Objekte und/oder Operationen parametrisiert werden kann. Eine instantiierte Klasse entsteht aus einer parametrisierten Klasse, in dem man den formalen Parametern konkrete Werte zuweist. Sie wird durch eine Wolke mit durchgezogenem Rechteck dargestellt. Der gestrichelte Pfeil von einer instantiierten Klasse (z.B. `CRechnerliste`) zu einer parametrisierten Klasse (z.B. `Standardliste`) gibt die Instantiierungsbeziehung zwischen den beiden Klassen an. Weitere Beziehungen zwischen Klassen, die in der Abbildung vorkommen,sind die Benutzt-Beziehung, dargestellt durch eine einfache Linie, und die Besteht_Aus-Beziehung, dargestellt durch eine Linie mit einem Punkt an einem Ende. Die an den Enden der Beziehungslinien stehenden Konstanten geben die Kardinalität der Beziehung an. So besteht etwa eine `Loginliste` aus n `CLogin`.

Die Klasse `Standardliste` beschreibt die im O_2-System definierte Klasse `List`. Diese wird von den Klassen `CLoginlist`, `CRechnerlist` und `CPGMemberlist` benutzt und stellt alle Standardoperationen auf Listen zur Verfügung.

7.2.1 Die Klasse `CPGMember`

Die Klasse `CPGMember` besitzt die Attribute `m_sLoginname` vom Typ `String`, `m_sName` vom Typ `String` und `m_oCLoginlist` vom Typ `CLoginlist`. `m_sLoginname` beinhaltet den Login-Namen ei-

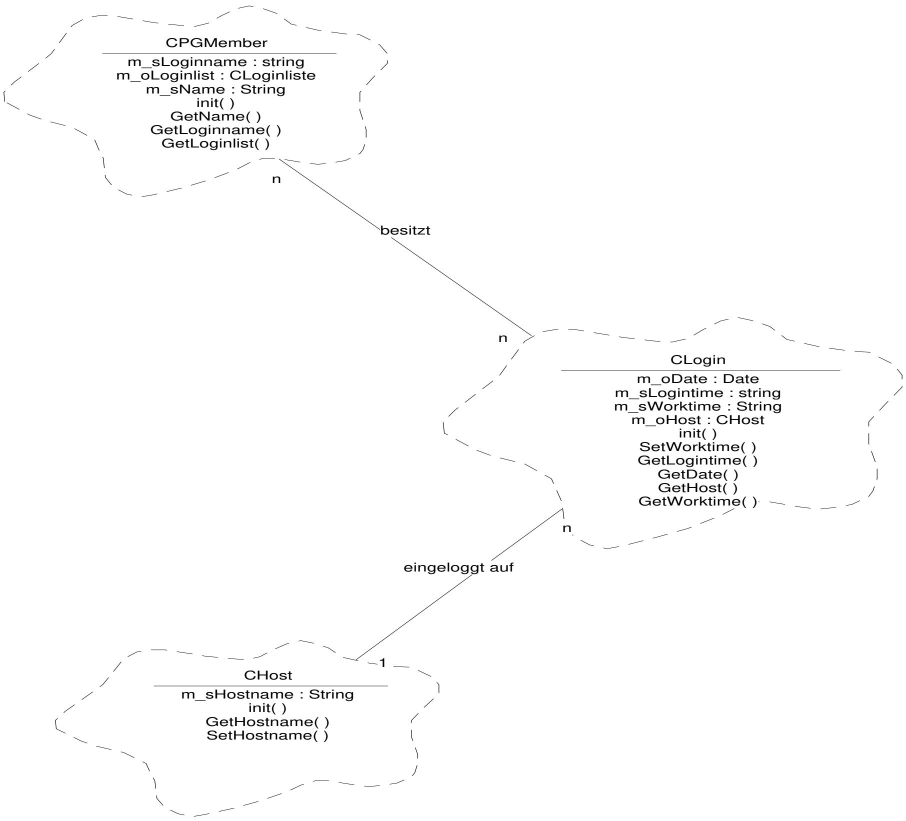


Abbildung 7.1: Analyse der zu erfassenden Daten

nes PG-Teilnehmers, `m_sName` den Namen des PG-Teilnehmers, bestehend aus Vor- und Nachname. In `m_oLoginlist` werden die einzelnen Rechner-Sitzungen des jeweiligen PG-Mitglieds festgehalten. Die Methoden der Klasse `CPGMember` sind [1]:

`GetLoginname` Die Methode `GetLoginame` hat keine Parameter. Sie liefert den Login-Namen, welcher vom Typ `String` ist.

`GetLoginlist` Diese Methode wird ebenfalls ohne Parameter aufgerufen und liefert die Liste mit den Logins des PG-Teilnehmer vom Typ `CLoginlist` zurück.

7.2.2 Die Klasse `CPGMemberlist`

In der Klasse `CPGMemberlist` werden die zwölf Mitglieder der PG-Gruppe verwaltet. Sie besitzt das Attribut `m_aMemberlist`, welches vom Typ `list(CPGMember)` ist und das Attribut `m_sPGName` vom Typ `String`. Auf dieser Klasse können die Methoden `Exists, Insert, Get` ausgeführt werden:

[1] Hier und auch bei den Beschreibungen der übrigen Klassen, soll auf die Erklärung der O_2–Constructor–Methode `init` verzichtet werden.

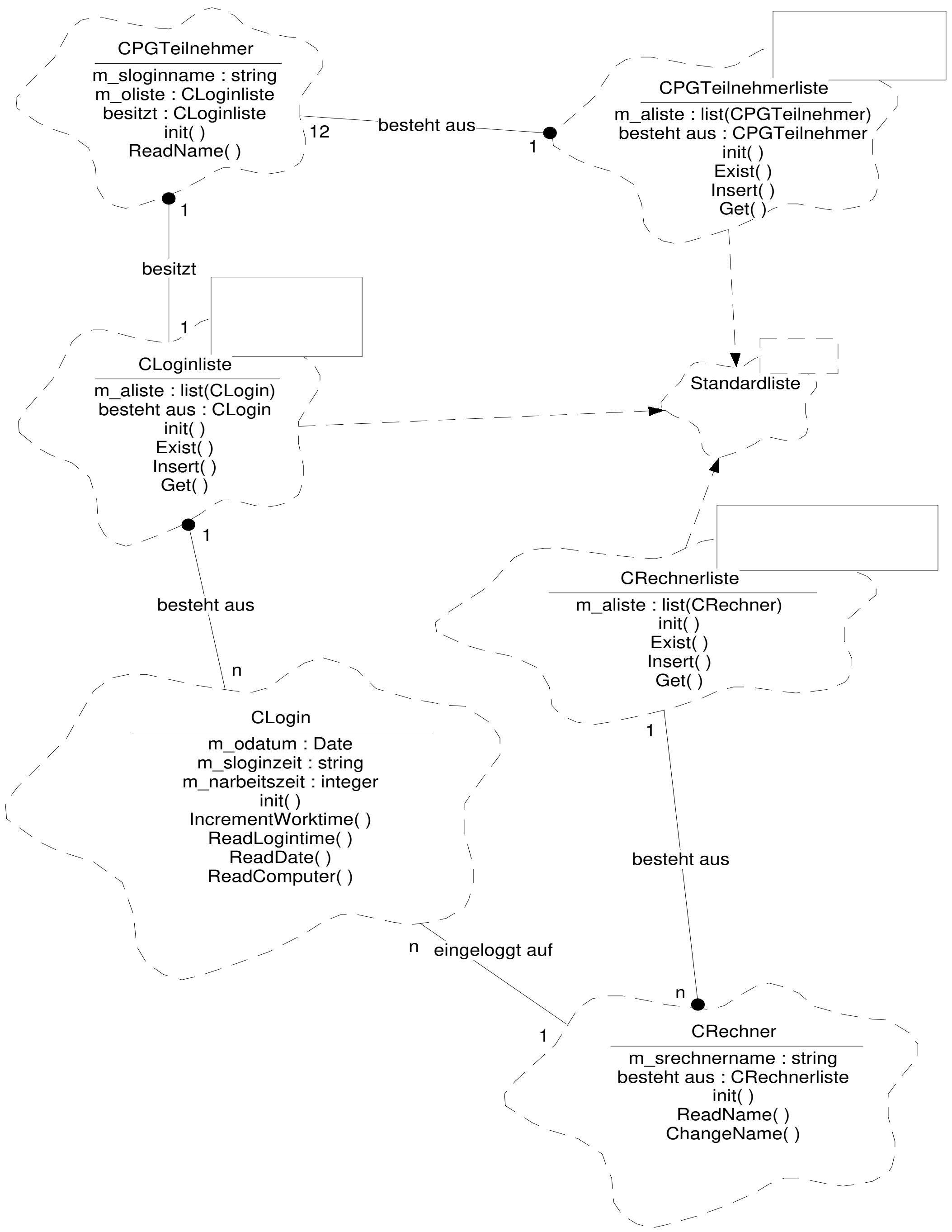


Abbildung 7.2: Modell für die Datenspeicherung mit O_2

Exists Die Methode wird mit einem Parameter `i_sLoginname` vom Typ `String` aufgerufen und liefert als Ergebnis einen Wert vom Typ `Boolean`. Dabei wird die interne Liste `m_aMemberlist` durchlaufen und die Loginnamen ihrer Elemente mit `i_sLoginname` verglichen. Stimmt der Loginname eines Elementes mit `i_sLoginname` überein, so wird `TRUE` zurückliefert, sonst `FALSE`.

Insert Aufruf hier mit einem Parameter `i_oMember` vom Typ `CPGMember`, kein Rückgabewert. Es wird davon ausgegangen, daß zuvor mit `Exists` geprüft wurde, ob `i_oMember` schon in der Liste vorhanden ist. `Insert` sollte nur aufgerufen werden, wenn `Exists FALSE` liefert.

Get Diese Methode wird ebenfalls mit einem Parameter `i_sLoginname` vom Typ `String` aufgerufen. Wie bei `Insert` wird davon ausgegangen, daß vorher mittels `Exists` die Existenz von `i_sLoginname` in der Liste geprüft wurde. Hat `Exists` einen positiven Wert geliefert, darf `Get` aufgerufen werden, wobei hier ein Zeiger auf das entsprechende Objekt zurückgegeben wird.

7.2.3 Die Klasse `CLogin`

Die Klasse `CLogin` besteht aus den Attributen `m_oDate` vom Typ `Date`, `m_sLogintime` vom Typ `String`, `m_sWorktime` vom Typ `String` und `m_oHost` vom Typ `CHost`. Die Attribute repräsentieren das Datum, die Loginzeit, die Arbeitsdauer und den Rechner des jeweiligen Logins. Die Klasse bietet die Methoden `SetWorktime, GetWorktime, GetDate, GetLogintime, GetHost` an:

SetWorktime Diese Methode wird mit dem Parameter `i_sWorktime` vom Typ `String` aufgerufen, welcher die von Rusers übermittelte Arbeitszeit darstellt. Die Methode setzt den Wert des Attribute `m_sWorktime` auf den Wert des übergebenen Parameters. Sie liefert keinen Wert zurück.

GetWorktime Die Methode `GetWorktime` wird ohne Parameter aufgerufen und liefert den Wert des Attributes `m_sWorktime` vom Typ `String` zurück.

GetDate Die Methode `GetDate` wird ohne Parameter aufgerufen. Sie liefert den Wert des Attributes `m_oDate` vom Typ `Date` zurück.

GetLogintime Die Methode `GetLogintime` wird ohne Parameter aufgerufen. Sie liefert den Wert des Attributes `m_sLogintime` vom Typ `String` zurück.

GetHost Diese Methode wird ohne Parameter aufgerufen und liefert den Wert des Attributes `m_oHost` vom Typ `CHost` zurück.

7.2.4 Die Klasse `CLoginlist`

Die Klasse `CLoginlist` dient zur Verwaltung der einzelnen Rechner-Sitzungen der PG-Teilnehmer. Ihr einziges Attribut ist `m_aLoginlist` vom Typ `list(CLogin)`. Analog zur Klasse `CPGMemberlist` gibt es auch hier die Methoden `Exists, Insert, Get`:

Exists Die Methode wird mit den Parametern `i_oDate` vom Typ `Date`, `i_sLoginTime` vom Typ `String` und `i_sHostname` vom Typ `String` aufgerufen. Als Ergebnis wird ein Wert vom Typ `Boolean` zurückgeliefert. Es wird anhand der Parameter `i_oDate, i_sLogintime` und `i_sHostname` geprüft, ob ein entsprechender Eintrag bereits in der Liste vorhanden ist. Falls ja, wird `TRUE` zuückgeliefert, sonst `FALSE`.

Insert Als Parameter besitzt diese Methode den Parameter `i_oLogin` vom Typ `CLogin`. Vor dem Aufruf sollte mit `Exists` sichergestellt worden sein, daß sich kein identischer Wert in der (Login)liste befindet.

Get Die Methode hat die gleichen Parameter wie `Exists`. Als Vorraussetzung für den Aufruf von `Get` sollte vorher mit `Exists` getestet werden, ob der bestimmte Eintrag auch in der Liste vorhanden ist. Der Returnwert ist ein Zeiger auf das entsprechende Objekt

7.2.5 Die Klasse `CHost`

Einziges Attribut der Klasse `CHost` ist `m_sHostname` vom Typ `String`. Es werden die Methoden `GetHostname` und `SetHostame` angeboten:

`GetHostname` Die Methode `GetHostname` wird ohne Parameter aufgerufen und liefert den Wert des Attributes `m_sHostname` vom Typ `String` zurück.

`SetHostname` Diese Methode wird mit dem Parameter `i_sHostname` vom Typ `String` aufgerufen und ersetzt den Wert des Attributes `m_sHostname` durch den Wert des Parameters.

7.2.6 Die Klasse `CHostlist`

In der Klasse `CPGHostlist` werden die Rechnernamen gespeichert, auf denen sich die PG-Teilnehmer einloggen können. Sie besitzt das Attribut `m_aHostlist` vom Typ `list(CHost)`. Wie bei der Klasse `CPGMemberlist` und `CLoginlist` stehen hier die Methoden `Exists, Insert` und `Get` zur Verfügung:

`Exists` Die Methode wird mit dem Parameter `i_sHostname` vom Typ `String` aufgerufen und liefert als Ergebnis einen Wert vom Typ `Boolean`. Sie überprüft, ob es in der Rechnerliste einen Rechner mit dem übergebenem Rechnernamen gibt. Trifft dies zu, so erhält der Rückgabewert den Wert `TRUE`, sonst `FALSE`.

`Insert` Disese Methode wird mit dem Parameter `i_oHost` vom Typ `CHost` aufgerufen. Dieser wird in `m_aHostlist` eingefügt. Vorher muß mit `Exists` sichergestellt worden sein, daß kein Rechner mit dem selben Rechnernamen bereits in der Rechnerliste existiert.

`Get` Die Methode `Get` wird mit dem Parameter `i_sHostname` aufgerufen und liefert einen Zeiger auf den Rechner zurück, der unter dem Rechnernamen `i_sHostname` geführt wird. Zuvor muß wieder sichergestellt worden sein, daß es diesen Rechner auch wirklich gibt.

7.3 Die Implementierung

Im Rahmen der Einarbeitungsaufgabe sind zwei Applikationen implementiert worden: *Rusers*, eine autonome Datenbankapplikation, die mit O_2C unter Verwendung von O_2*Look* realisiert worden ist und der im Entwurf vorgestellten Modellierung entspricht, und der mit *C++* implementierte O_2-Datenbank-Adapter.

7.3.1 Rusers

Die Applikation *Rusers* ermöglicht es einem Benutzer, die vom *FDBMS* über den O_2-Datenbank-Adapter in die Datenbank eingetragenen Daten zu sichten. Um eine graphische Oberfläche zu Verfügung zu stellen, wurde O_2*Look* verwendet. Mit diesem Hilfsmittel ist es sehr schnell möglich eine graphische, ausführbare Applikation zu erstellen, die als Grundlage die vorhandenen Datenstrukturen benutzt. Aufgrund der Arbeitsweise von O_2*Look* entsteht hierdurch allerdings eine große Anzahl von Fenstern, so daß die Handhabung der Applikation wenig benutzerfreundlich ist. Aufgerufen werden kann die Applikation auf zwei verschiedene Arten:

1. Über die O_2-Kommandozeile. Zuesrt muß die O_2-Kommandozeile mit `o2 -server bern -system iuvol` gestartet werden. Danach ist mit den Befehlen `set schema s_rusers` und `set base rusers` das Schema, bzw. die Base gesetzt werden. Die Befehle müssen jeweils mit `RETURN` und `Ctrl-D` abgeschlossen werden. Anschließend startet man die Applikation mit dem Aufruf `run application Rusers`.

2. Über die graphische Entwicklungsoberfläche. Hierzu ist O_2 mit dem Aufruf `o2 -server bern -system iuvol -toolsgraphic` zu starten. Anschließend müssen wieder Schema und Base im entsprechenden Untermenue gesetzt werden, bevor man im Untermenue *APPS* die Applikation `Rusers` auswählen und mittels `Test` ausführen kann.

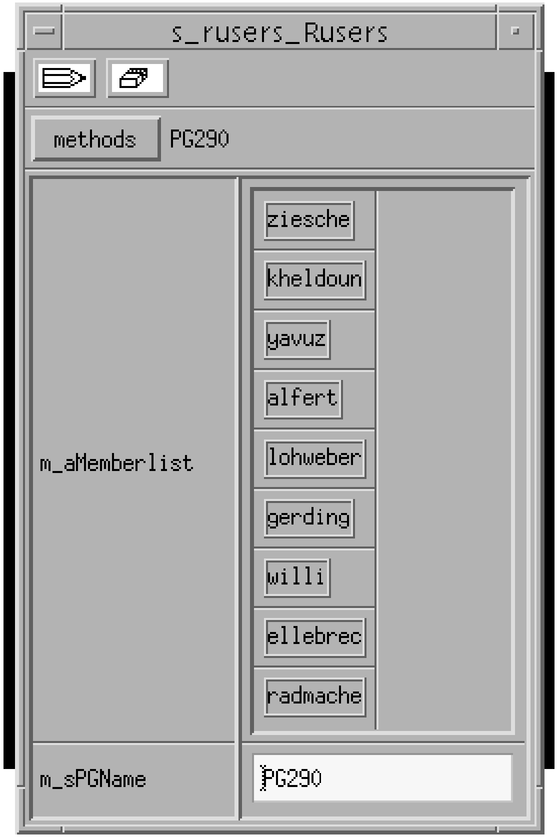


Abbildung 7.3: Screenshot der O_2-Applikation *Rusers*

Die Steuerung der Applikation erfolgt, ausgehend vom Startfenster, über die Auswahl diverser Menuepunkte, die mittels der rechten Maustaste angewählt werden können. Die Menuepunkte entsprechen den Methoden der dargestellten Objekte (siehe Abbildung 7.3 und einigen von O_2*Look* zur Verfügung gestellten Methoden .

Der Quellcode

```
class CPGMember inherit Object public type
    tuple(m_sLoginname: string,
          m_sName: string,
          m_oLoginlist: CLoginlist)


class CLogin inherit Object public type
    tuple(m_oDate: Date,
          m_sLogintime: string,
          m_sWorktime: string,
          m_oHost: CHost)
```

```
class CHost inherit Object public type
    tuple(m_sHostname: string)


class CLoginlist inherit Object public type
    tuple(m_aLoginlist: list(CLogin))


class CPGMemberlist inherit Object public type
    tuple(m_aMemberlist: list(CPGMember),
          m_sPGName: string)


class CHostlist inherit Object public type
    tuple(m_aHostlist: list(CHost))


application Rusers
    program
        public PGWaehlen,
        public PGAnlegen,
        public HOSTSAnlegen
end;
```

7.3.2 Der O_2-Datenbank-Adapter

Der Code für den O_2-Datenbank-Adapter ist im wesentlichen aus dem Schema von *Rusers* exportiert worden. Die von uns implementierten Klassen `CO2HandlerFactory, CO2InsertHandler` entsprechen den Vorgaben, die vom *FDBMS* für den Adapter festgelegt worden sind. Für die Einfügeoperationen werden die Methoden der jeweiligen Klassen der O_2Applikation *Rusers* benutzt, so daß der Code des Adapters recht kurz und übersichtlich ist.

Beim Aufruf des Adapters ist darauf zu achten, daß die Umgebungsvariablen `O2SERVER` und `O2SYSTEM` korrekt gesetzt sind. (Hier: `setenv O2SERVER bern` und `setenv O2SYSTEM iuvol`).

Der Quellcode

Der O_2InsertHandler:

```
#include "CO2InsertHandler.h"
#include <rw/cstring.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "o2lib_CC.hxx"
#include "o2template_CC.hxx"
#include "o2util_CC.hxx"
#include "CHost.hxx"
#include "CLogin.hxx"
#include "CPGMember.hxx"
```

```
#include "CHostlist.hxx"
#include "CLoginlist.hxx"
#include "CPGMemberlist.hxx"

   bool CO2InsertHandler::Handle(CInsert* poInsert)
   {
      RWCString sSchema, sClass, sLoid;
      RWCString sBenutzerName;
      RWCString sRechnerName;
      RWCString sAktuellZeit;
      RWCString sLoginZeit;
      RWCString sLoginDatum;

      bool result = FALSE;

      cout<<"Objekt wird empfangen:"<<endl;
      result =  poInsert->GetValue( "sBenutzerName", &sBenutzerName );
      result &= poInsert->GetValue( "sRechnerName", &sRechnerName );
      result &= poInsert->GetValue( "sAktuellZeit", &sAktuellZeit );
      result &= poInsert->GetValue( "sLoginZeit", &sLoginZeit );
      result &= poInsert->GetValue( "sLoginDatum", &sLoginDatum );

      if( result )
      {
         cout<<"Name: "<<sBenutzerName<<", Host: "<<sRechnerName
         <<", Zeit: "<<sAktuellZeit<<", Datum: "<<sLoginDatum<<endl;
         insert_login( sBenutzerName,
                       sLoginDatum,
                       sLoginZeit,
                       sAktuellZeit,
                       sRechnerName );
         cout<<"--------------------------------------------------"<<endl;
      }
      else
      {
         cout<<"Fehler bei Receive"<<endl;
         return false;
      }
      return true;
   }

/****************************************************************************/
/*  Funktion:   insert_member                                               */
/*  Parameter:  char * i_sName  -  Loginname des PG-Teilnehmers             */
/*  Returnwert: -                                                           */
/****************************************************************************/
   void CO2InsertHandler::insert_member( char *i_sName )
   {
      d_Transaction update;
      d_Ref<CPGMemberlist> oPG( "PGs" );
      d_Ref<CPGMember> oCurrent = new CPGMember( i_sName );

     update.begin();
     if ( !( oPG->Exists( i_sName ) ) )
```

```
    {
      cout<<"Neuer Teilnehmer wird eingefügt."<<endl;
      oPG->Insert( oCurrent );
    }
    update.validate();
  }


/*************************************************************************/
/*  Funktion:   insert_host                                              */
/*  Parameter:  char * i_sHost  -  Rechnername                           */
/*  Returnwert: -                                                        */
/*************************************************************************/
   void CO2InsertHandler::insert_host(char *i_sHost)
   {
      d_Transaction update;
      d_Ref<CHostlist> oHOST( "HOSTs" );
      d_Ref<CHost> oCurrent = new CHost( i_sHost );

      update.begin();
      if ( !( oHOST->Exists( i_sHost ) ) )
      {
         cout<<"Neuer Rechner wird eingefügt."<<endl;
         oHOST->Insert( oCurrent );
      }
      update.validate();
   }


/*************************************************************************/
/*  Funktion:   insert_login                                             */
/*  Parameter:  const char *i_sName      -  Loginname des PG-Teilnehmers  */
/*              const char *i_sLogindate -  Logindatum                   */
/*              const char *i_sLogintime -  Loginzeit                    */
/*              const char *i_sWorktime  -  aktuelle Zeit                */
/*              const char *i_sHost      -  Rechnername                  */
/*  Returnwert: -                                                        */
/*************************************************************************/
   void CO2InsertHandler::insert_login(
                                     const char *i_sName,
                                     const char *i_sLogindate,
                                     const char *i_sLogintime,
                                     const char *i_sWorktime,
                                     const char *i_sHost )
   {
      d_Transaction update;
      d_Ref<CPGMemberlist> oPG( "PGs" );
      d_Ref<CHostlist> oHOST( "HOSTs" );
      d_Ref<CPGMember> oCurrent;
      d_Ref<CLoginlist> oLoginlist;
      d_Ref<Date> New_Date;
      d_Ref<CHost> oHost;
      d_Ref<CLogin> oLogin;
```

```
//
// Datum konvertieren
//

int Day, Month, Year;
int found = 0;
int i;

char months[12][4] = {"Jan", "Feb", "Mar", "Apr", "May" ,"Jun",
                      "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

RWCString date_str;
RWCString s_month;
RWCString s_day;

date_str = i_sLogindate;
s_month = date_str( 0,3 );
if (date_str[4] == ' ')
   s_day = date_str( 5,1 );
else
   s_day = date_str( 4,2 );

for ( i=0; i<12; i++ )
{
   if( s_month == months[i] )
   {
      found = 1;
      break;
   }
}

if ( found )
{
   Day = atoi( s_day );
   Month = i + 1;
   Year = 1997;        // Wird einfach mal so festgelegt
   New_Date = new Date( Day, Month, Year );
}
else
{
   cout<<"Konnte Datum nicht konvertieren!"<<endl;
}

//
// Ende von Datum konvertieren
//

if ( !( oPG->Exists( i_sName ) ) )
{
   cout<<"insert_login: Teilnehmer nicht in Liste -> einfügen"<<endl;
   insert_member( i_sName );
}
if ( ! ( oHOST->Exists( i_sHost ) ) )
{
```

```
         cout<<"login_einfuegen: Rechner nicht in Liste -> einfügen"<<endl;
         insert_host( i_sHost );
      }
      oCurrent = oPG->Get( i_sName );
      oLoginlist = oCurrent->GetLoginlist();

      update.begin();
      if( !( oLoginlist->Exists ( i_sLogintime, New_Date, i_sHost ) ) )
      {
         cout<<"Neuer Login wird eingefügt."<<endl;
         oHost = oHOST->Get( i_sHost );
         oLogin = new CLogin( New_Date,
                              (char*) i_sLogintime,
                              (char*) i_sWorktime,
                              oHost );
         oLoginlist->Insert(oLogin);
      }
      else
      {
         cout<<"Login wird aktualisiert."<<endl;
         oLogin = oLoginlist->Get( i_sLogintime, New_Date, i_sHost );
         oLogin->SetWorktime( i_sWorktime );
      }
      update.commit();
   }
```

Das Hauptprogramm:

```
/*************************************************************************/
/*                                                                       */
/*  Datenbankadapter fuer O2-Rusers                                      */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include "o2lib_CC.hxx"
#include "o2template_CC.hxx"
#include "o2util_CC.hxx"
#include <comi.H>
#include "CO2HandlerFactory.h"

/*************************************************************************/
/*  Das Hauptprogramm                                                    */
/*  Returnwert: int                                                      */
/*************************************************************************/
int main(int argc, char **argv)
{
   d_Session session;
   d_Database database;
   d_Transaction update;

   CO2HandlerFactory g_oFactory;
```

```
    session.set_default_env();
    if ( session.begin ( argc, argv ) )
    {
      cerr << "Something wrong to start o2" << endl;
      exit( EXIT_FAILURE );
    }

    cout<<"Datenbank öffnen."<<endl;
    database.open( "rusers" );

    CReceiver* poReceiver = CReceiver::Instance();
    poReceiver->Init( "O2_DB", (CHandlerFactory*) &g_oFactory, argc, argv);

    cout<<"DB-Adapter arbeitet."<<endl;
    poReceiver->WaitForOperation();

    poReceiver->Terminate();
    CReceiver::Destroy();
    poReceiver = NULL;

    cout<<"Datenbank schließen."<<endl;
    database.close();

    session.end();
    cout<<"Das Ende!"<<endl;
    return( EXIT_SUCCESS );
}
```

Kapitel 8

Die Entwicklung der Oracle Applikation

Verfasser: Mischa Lohweber, Andreas Dinsch

8.1 Anforderungen

In der Anwendung sollen die von `rusers` gelieferten Daten über Benutzer gespeichert werden. Diese Daten sollen mit Hilfe einer graphischen Benutzungsoberfläche angezeigt und verändert werden können.

8.2 Der Oracle Grobentwurf

Das ER-Diagramm wie in Abbildung 8.1, auf Seite 144, stellt das Datenschema da, in welcher die Benutzerdaten gespeichert werden sollen.

8.3 Der Feinentwurf

Die Daten werden in zwei Tabellen gespeichert. Die Tabelle `USERS` beinhaltet, wie in Tabelle 8.1 dargestellt, den Namen des Benutzers und einen Schlüssel (`IDUSER`). Das Kürzel 'PK' meint einen "PrimaryKey", also einen Schlüssel, der jeden Datensatz in dieser Tabelle identifiezeiert. Da dieser Schüssel eindeutig (Unique) ist, ist auch die Identifizierung eines Datensatzes eindeutig. Das Kürzel 'Not Null' sagt aus, das dieses Feld niemals einen Nullwert (`NULL`) haben darf. Die Tabelle `HOSTS` enthält, wie in Tabelle 8.2 dargestellt, die Namen der Rechner und ebenfalls einen Schlüssel (`IDHOST`).

Zwischen diesen beiden Tabellen herscht eine n:m Beziehung. Ein Benutzer kann auf beliebig vielen Rechnern eingeloggt sein und auf einem Rechner können beliebig viele Benutzer sein. Diese Beziehung wird in relationalen Datenbanken mit Hilfe einer Tabelle dargestellt. Die dritte Tabelle, welche in Tabelle 8.3 zu sehen ist, stellt also die Relation zwischen der `USERS` und der `HOSTS` Tabelle dar. Die

USERS		NOT NULL	PK/FK	Unique
<u>IDUser</u>	NUMBER	x	PK	x
Name	CHAR	x		

Tabelle 8.1: Tabelle USERS.

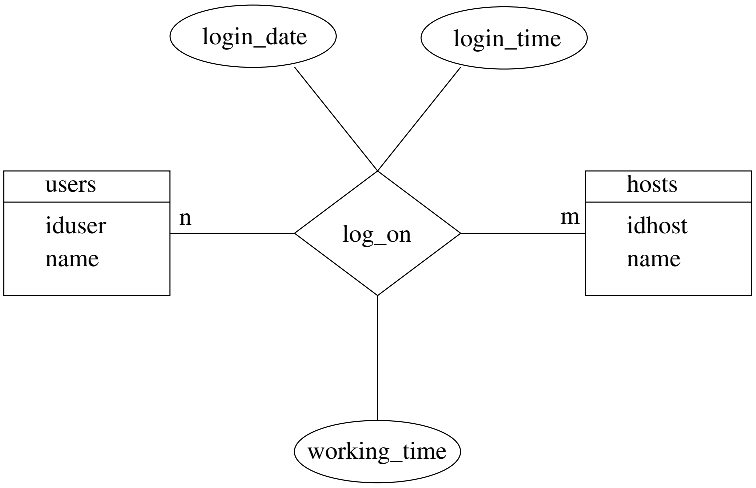


Abbildung 8.1: ER-Diagramm des Oracle Grobentwurfs.

HOSTS		NOT NULL	PK/FK	Unique
IDHost	NUMBER	x	PK	x
Name	CHAR	x		

Tabelle 8.2: Tabelle HOSTS.

Tabelle `LOG_ON` enthält die Felder `IDUser, IDHost, Login_Time, Login_Date` und `Working_Time`. Der Schlüssel setzt sich aus dem Fremdschlüssel (ForeignKey) `IDUser` und `Login_Date` zusammen. Es wäre hier überflüssig einen eigenen Primärschlüssel (PK) anzulegen, da die Verwendung der beiden Fremdschlüssel (FK) `IDUser` und `Login_Date` zu einem eindeutigen Primärschlüssel führen. Es wird also für jeden Benutzer gespeichert, wann er wie lange auf einem bestimmten Rechner eingeloggt war oder ist. Es ensteht dabei eine Art Historie, da der Schüssel sich aus der UserID und den Datum des letzten Logins zusammensetzt.

Für die Realisierung der Oberfläche soll OracleForms verwendet werden. Die Initialisierung der Tabellen wird in SQLPLUS [Ora94a] vorgenommen. SQLPLUS bietet eine shellähnliche Oberfläche, in welcher unter zu Hilfenahme von SQL Befehlen Tabellen erstellt, Daten abgefragt, gelöscht und geändert werden können.

8.4 Der Entwurf des Agenten

Der Agent dient zur Kommunikation zwischem dem föderiertem Datenbank System, von nun an FDBS genannt, und der Oracle-Datenbank. Er soll automatisch Informationen vom FDBS empfangen, diese analysieren und dann in die Datenbank eintragen. In einer späteren Version sollen Daten, die

LOG_ON		NOT NULL	PK/FK	Unique
IDUser	NUMBER	x	PK/FK	
Login_Date	DATE	x	PK	
IDHost	NUMBER	x	FK	
Login_Time	CHAR			
Working_Time	CHAR			

Tabelle 8.3: Tabelle LOG_ON.

unabhängig vom FDBS, in der Datenbank eingefügt oder geändert wurden, an das FDBS gesendet werden.

8.4.1 Die übermittelten Daten

Der Agent erhält Daten über die Benutzer in unserer Projektgruppe. Das FDBS übermittelt, welcher Benutzer auf welchem Rechner zu welcher Zeit eingelogt ist, die aktuelle Zeit und das aktuelle Datum der Übertragung. Es wird also der Tupel ('BenutzerName', 'RechnerName', 'AktuelleZeit', 'LoginZeit', 'LoginDatum') übermittelt. Beispiele für eine Übermittlung:

```
lohweber london 12:01 08:00 Jan 27
gao tripolis 12:01 10:12 Jan 27
gao ottawa 12:01 10:00 Jan 27
gerding ottawa 12:01 08:15 Jan 27
```

8.4.2 Die Realisierung

Der Agent ist ein C++ Programm, das als Hintergrundprozeß, ein sogenannter 'Deamon', läuft. Das Programm besteht aus mehreren Teilen. Der Hauptteil (`main.c`) ist dafür verantwortlich den *Cool–Orb* anzusprechen und somit die Verbindung zum Föderierungssystem herzustellen und diesem die Routinen zum Einfügen der Daten in die Datenbank zur Verfügung zu stellen. Die Anbindung an *Cool–Orb* wird durch das Kommunikationsinterface bereitgestellt (`comi.c`) und wird in Kapitel 5 beschrieben. Die Erstellung der Routine des Handlers zum Einfügen der Daten und die Routine zum eigentlichen Einfügen in die Datenbank müssen für das jeweilige Datenbanksystem, in unserem Fall Oracle, angepaßt werden. Diese Routinen sind die `OracleHandlerFactory (OracleHandlerFactory.c)` und der `OracleInsertHandler (OracleInsertHandler.pc)`. Die `OracleHandlerFactory` stellt dem Kommunikationsinterface eine Möglichkeit zur Verfügung einen `InsertHandler` zu erzeugen, der das Einfügen der Daten in die Datenbank vornimmt.In unserem Fall wird ein `InsertHandler` anhand der Klasse `COracleInsertHandler` definiert.
Die Datei `OracleHandlerFactory.c` enthält folgenden C++ Code:

```
#include "OracleHandlerFactory.h"
CInsertHandler* COracleHandlerFactory::CreateInsertHandler()
{
 return (CInsertHandler*) new COracleInsertHandler;
};
```

Der OracleInsertHandler stellt nun die Funktionalität zum konkreten Einfügen der Daten in die Oracle Datenbank zur Verfügung. Der OracleInsertHandler ist in *Pro*C* [Ora94b] geschrieben. Durch *Pro*C* werden oraclespezifische Variablen und Konstanten zur Verfügung gestellt. Ausschnitt aus dem Deklarationsteil von `OracleInsertHandler.pc`:

```
[...]
EXEC SQL BEGIN DECLARE SECTION;
#define     UNAME_LEN      20
#define     PWD_LEN        40
#define     EINGABE_LEN    40
#define     SQL_LEN        400

VARCHAR username[UNAME_LEN];  /* VARCHAR is an ORACLE supplied struct */
varchar password[PWD_LEN];    /* varchar can be in lower case also */
VARCHAR sqlstmt[SQL_LEN];
```

```
VARCHAR host_name[EINGABE_LEN];
VARCHAR user_name[EINGABE_LEN];
VARCHAR stunden[EINGABE_LEN];
varchar login_time[EINGABE_LEN];
varchar login_date[EINGABE_LEN];
EXEC SQL END DECLARE SECTION;
[...]
```

Vor der eigentlichen Erstellung des Programms muß der *Pro*C*-Code von einem Precompiler in ein C++ Programm umgewandelt werden. Die Verwendung von *Pro*C*-Code an Stelle von reinem C++ Code hat den Vorteil, daß 'Embeded SQL' verwendet werden kann. Es ist also möglich direkt im C++ Code SQL Anweisungen zu schreiben. Hier ein Ausschnitt aus `OracleInsertHandler.pc` in welchen die Daten per SQL Anweisung in die Datenbank geschrieben werden. Der komplette Quellcode ist in Kapitel 8.4.4 zu finden. Der folgende Ausschnitt zeigt die *Pro*C* Anweisung, welche die Anmeldung an die Oracledatenbank darstellt.

```
[...]
EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error:");
oraca.orastxtf = ORASTFERR;
username.len = strlen(strcpy((char *)username.arr, "PG290"));
password.len = strlen(strcpy((char *)password.arr, "PG290"));
EXEC SQL CONNECT :username IDENTIFIED BY :password;
cout<<"\nConnected to ORACLE."<<endl;
[...]
```

Die folgenden Programmzeilen zeigen die eigentliche SQL Anweisung, welche die Eintragungen in die Datenbank vornimmt. Zur genaueren Beschreibung siehe Kapitel 8.4.3. Das SQL Statement wird in eine Variable vom Typ `VARCHAR` kopiert. Diese Variablentypen werden durch die Oracle Programmbibliotheken zur verfügung gestellt. Alle SQL Anweisungen oder Variablen welche an SQL Statements übergeben werden, müssen von diesem Typ sein, da sonst Fehler beim Aufruf des Precompliers auftreten.

```
[...]
strcpy((char *)sqlstmt.arr, \
"INSERT INTO log_on (IDUSER, IDHOST, WORKING_TIME, LOGIN_TIME, LOGIN_DATE)
SELECT USERS.IDUser, HOSTS.IDHost, :v1, :v2, TO_DATE(:v3, 'DD-MM-YY')
FROM USERS, HOSTS
WHERE USERS.Name = :v4 AND HOSTS.Name = :v5 AND NOT EXISTS
(SELECT LOGIN_TIME, LOGIN_DATE, IDUSER, IDHOST FROM LOG_ON
 WHERE LOGIN_TIME = :v2 AND LOGIN_DATE = TO_DATE(:v3, 'DD-MM-YY') AND
 IDUSER = (Select IDUser from USERS Where Name = :v4) AND
 IDHost = (Select IDHost From HOSTS Where Name = :v5))");

sqlstmt.len = strlen((char *)sqlstmt.arr);

EXEC SQL PREPARE S FROM :sqlstmt;
EXEC SQL EXECUTE S USING :stunden, :login_time,:login_date,:user_name,:host_name;
EXEC SQL COMMIT RELEASE;
[...]
```

Diese Anweisungen werden nach dem Durchlauf des Preocompilers in Aufrufe von Oracle-Systembibliotheken umgewandelt, so daß das nun entstandene C++ Programm mit dem CC-Compiler übersetzt und mit dem Hauptprogramm verknüpft werden kann. Der nun entstandene C++ Code ist nicht mehr leserlich und wird deshalb hier nicht abgebildet.

Die hier verwendeten SQL Anweisungen beschränken sich auf SQL Methode 2. Die Methode 2 bedeutet, daß keine Abragen, also Querys, als Ergebnis einer Anweisung möglich sind und das nur mit einer vorher bekannte Anzahl von Variablen im SQL Statement gearbeitet werden kann. Die SQL Anweisung (mit einer beaknnten Anzahl von Variablen) wird durch den Befehl " EXEC SQL PREPARE <symbolischer Name> FROM <SQl Anweisung> " zur Ausführung vorbereitet. Der Befehl "EXEC SQL EXECUTE <symbolischer Name> Variable1 Variable2 ..." führt dann die Anweisung aus und fügt die Werte der übergebenen Variablen in das vorher präparierte SQL Statement ein.

8.4.3 Erläuterung des SQL Statements an Hand eines Beispiels

Der eigentliche Teil, welcher die Eintragungen in der Datenbank vornimmt, ist die SQL Anweisung in der Datei OracleInsertHandler.pc. Die Anweisung in reinem SQl lautet:

```
INSERT INTO LOG_ON (IDUSER, IDHOST, WORKING_TIME, LOGIN_TIME, LOGIN_DATE)
SELECT USERS.IDUser, HOSTS.IDHost,<Zeit1>, <Zeit2>, TO_DATE(<Datum>, 'DD-MM-YY')
FROM USERS, HOSTS WHERE USERS.Name = <BenutzerName> AND HOSTS.Name = <RechnerName>
AND NOT EXISTS(
SELECT LOGIN_TIME, LOGIN_DATE, IDUSER, IDHOST FROM LOG_ON
WHERE LOGIN_TIME = <Zeit1> AND LOGIN_DATE = TO_DATE(<Datum>, 'DD-MM-YY')
AND IDUSER = (SELECT IDUser FROM USERS WHERE Name = <BenutzerName>) AND
IDHost = (SELECT IDHost FROM HOSTS WHERE Name = <RechnerName>));
```

Die Kürzel <Zeit1>,<Zeit2>,<Datum>,<BenutzerName>,<RechnerName> stehen für Variablen vom Typ VARCHAR und beinhalten die Daten welche vom Föderierungssystem geliefert wurden. Die Daten sind in Form von Strings vorhanden. Die Zeile INSERT INTO LOG_ON (IDUSER,.....,LOGIN_DATE) liefert die Anweisung einen Datensatz in die Tabelle LOG_ON einzufügen. Die Anweisung SELECT USERS.IDUser... dient dazu die ID des Namens des Benutzer und des Rechners zu liefern. Diese Abfrage ist nötig, da das Föderierungssystem den Namen des Beutzers und des Rechners in Stringform liefert (z.B. lohweber, london). An Hand dieses Namens muss dann aus der jeweiligen Tabelle die zugehörige ID ermittelt werden. Würde der Name direkt gespeichert, entständen redundante Daten. Die letzte WHERE Klausel (AND NOT EXISTS(SELECT LOGIN_TIME...) überprüft ob der einzufügende Datensatz schon verhanden ist. Dies wird so realisiert, in dem eine Abfrage mit den vom Föderierungssystem gelieferten Daten formuliert wird. Liefert diese Abfrage kein Ergebnis (NOT EXISTS) ist der Datensatz noch nicht vorhanden und der Eintrag wird vorgenommen. Ist der Datensatz vorhanden, wird keine Eintragung vollzogen.

8.4.4 Implementierung

Die Datei main.C:

```
/*
 * Name:   main.c
 *
 * Autor:  Mischa Lohweber
 *
 * Typ:    SourceDatei
 *
 * Beschreibung:
 *  Das eigentliche Hauptprogramm. Der Empfaenger wird initialisiert,
 *  geht danach in eine Endlosschleife und wartet auf Operationen.
 *
 *
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <comi.H>
#include "OracleHandlerFactory.h"

int main ( int argc, char* argv[] )
{
  COracleHandlerFactory g_oFactory;

  //Empfaenger erzeugen und initialisieren
  CReceiver* poReceiver = CReceiver::Instance();
  poReceiver->Init("ORACLE_DB", (CHandlerFactory*) &g_oFactory, argc, argv );

  cout<<"\nOracle Agent initialisiert."<<endl;
  cout<<"Verbindung zu FDBS hergestellt."<<endl;

  //Endlosschleife fuer Empfang
  poReceiver->WaitForOperation();

  //Programm beenden
  poReceiver->Terminate();
  CReceiver::Destroy();
  poReceiver = NULL;

  return 0;

}
```

Die Dateien `OracleHandlerFactory.h` und `OracleHandlerFactory.c`:

```
/*
 * Name:   OracleHandlerFactory.h
 *
 * Typ:    Headerdatei
 *
 * Autor:  Mischa Lohweber
 *
 * Beschreibung:
 *   Diese Klasse dient zur Erstellung des Operationshandlers.
 *
 */
#include <comi.H>
#include "OracleInsertHandler.h"

class COracleHandlerFactory:public CHandlerFactory {
  public:
  CInsertHandler* CreateInsertHandler();
};



/*
 * Name:   OracleHandlerFactory.c
```

```
 *
 * Autor:  Mischa Lohweber
 *
 * Beschreibung:
 *   Diese Klasse dient zur Erstellung des Operationshandlers.
 *
 *
 */

#include "OracleHandlerFactory.h"

CInsertHandler* COracleHandlerFactory::CreateInsertHandler()
{
  return (CInsertHandler*) new COracleInsertHandler;
};
```

Die Dateien `OracleInsertHandler.h` und `OracleInsertHandler.pc`:

```
/*
 * Name:   OracleInsertHandler.h
 *
 * Typ:    Headerdatei
 *
 * Autor:  Mischa Lohweber
 *
 *
 */

#include <comi.H>

class COracleInsertHandler : public CInsertHandler {
  public: virtual bool Handle(CInsert* poInsert);
};



/*
 * Name:   OracleInsertHandler.pc
 *
 * Typ:    PRO*C SourceDatei - Muss erst durch den Precompiler laufen!
 *
 * Autor:  Mischa Lohweber
 *
 * Beschreibung:
 *   Diese Funktion enthaelt den eingentlichen Handler fuer die Operation.
 *   Hier wird das Objekt das via Cool-Orb gesendet wurde ausgewertet, in
 *   einzelnen Komponenten zerlegt und in die Datenbank eingetragen.
 *
 *
 */
#include "OracleInsertHandler.h"
#include <rw/cstring.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include <iostream.h>
#include <sqlca.h>
#include <oraca.h>


/* Declare error handling function. */
void sql_error(char *msg);


bool COracleInsertHandler::Handle(CInsert* poInsert)
{

EXEC ORACLE OPTION (ORACA=YES);

EXEC SQL BEGIN DECLARE SECTION;
#define     UNAME_LEN      20
#define     PWD_LEN        40
#define     EINGABE_LEN    40
#define     SQL_LEN        400

VARCHAR username[UNAME_LEN];  /* VARCHAR is an ORACLE supplied struct */
varchar password[PWD_LEN];    /* varchar can be in lower case also */
VARCHAR  sqlstmt[SQL_LEN];
VARCHAR  host_name[EINGABE_LEN];
VARCHAR  user_name[EINGABE_LEN];
VARCHAR  stunden[EINGABE_LEN];
varchar  login_time[EINGABE_LEN];
varchar  login_date[EINGABE_LEN];
EXEC SQL END DECLARE SECTION;


RWCString sSchema, sClass, sLoid;
RWCString sBenutzerName;
RWCString sRechnerName;
RWCString sAktuellZeit;
RWCString sAktuellDatum;
RWCString sLoginZeit;
RWCString sLoginDatum;

bool result = FALSE;

// Hole Daten via Orb
result = poInsert->GetValue("sBenutzerName", &sBenutzerName);
result &=  poInsert->GetValue("sRechnerName", &sRechnerName);
result &=  poInsert->GetValue("sAktuellZeit", &sAktuellZeit);
result &=  poInsert->GetValue("sLoginZeit", &sLoginZeit);
result &=  poInsert->GetValue("sLoginDatum", &sLoginDatum);

if (result)
  {
  RWCString month;
  RWCString day;
  RWCString dest;
```

```
// Den erhaltenen Datumsstring der Form "MON DD" konvertieren nach "DD.MM.YY"
// Als Jahr wird das gerade aktuelle Jahr genommen

month  = sLoginDatum( 0, 3 );
day    = sLoginDatum( 4, 2 );
dest = day;
dest += ".";

if( month == "Jan" )
  dest += "01.";
if( month == "Feb" )
  dest += "02.";
if( month == "Mar" )
  dest += "03.";
if( month == "Apr" )
  dest += "04.";
if( month == "May" )
  dest += "05.";
if( month == "Jun" )
  dest += "06.";
if( month == "Jul" )
  dest += "07.";
if( month == "Aug" )
  dest += "08.";
if( month == "Sep" )
  dest += "09.";
if( month == "Okt" )
  dest += "10.";
if( month == "Nov" )
  dest += "11.";
if( month == "Dec" )
  dest += "12.";

dest += "97";

sLoginDatum = dest;

  // Holen der Daten erfolgreich, Daten in  Datenbank eintragen
  EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error:");
  oraca.orastxtf = ORASTFERR;
  username.len = strlen(strcpy((char *)username.arr, "PG290"));
  password.len = strlen(strcpy((char *)password.arr, "PG290"));
  EXEC SQL CONNECT :username IDENTIFIED BY :password;
  cout<<"\nConnected to ORACLE."<<endl;

  host_name.len = strlen(strcpy((char *)host_name.arr, sRechnerName));
  cout<<"RechnerName="<<sRechnerName<<endl;

  user_name.len = strlen(strcpy((char *)user_name.arr, sBenutzerName));
  cout<<"BenutzerName="<<sBenutzerName <<endl;

  login_date.len = strlen(strcpy((char *)login_date.arr, sLoginDatum));
  cout<<"LoginDatum="<<sLoginDatum<<endl;
```

```
    stunden.len = strlen(strcpy((char *)stunden.arr, sAktuellZeit));
    cout<<"AktuelleZeit="<<sAktuellZeit<<endl;

    login_time.len = strlen(strcpy((char *)login_time.arr, sLoginZeit));
    cout<<"LoginZeit="<<sLoginZeit<<endl;

    strcpy((char *)sqlstmt.arr,
    "INSERT INTO log_on (IDUSER, IDHOST, WORKING_TIME, LOGIN_TIME, LOGIN_DATE)
    SELECT USERS.IDUser, HOSTS.IDHost, :v1, :v2, TO_DATE(:v3, 'DD-MM-YY')
    FROM USERS, HOSTS
    WHERE USERS.Name = :v4 AND HOSTS.Name = :v5 AND NOT EXISTS
    (SELECT LOGIN_TIME, LOGIN_DATE, IDUSER, IDHOST FROM LOG_ON WHERE LOGIN_TIME = :v2
    AND LOGIN_DATE = TO_DATE(:v3, 'DD-MM-YY') AND IDUSER = (Select IDUser from USERS
    Where Name = :v4) AND IDHost = (Select IDHost From HOSTS Where Name = :v5))");

    sqlstmt.len = strlen((char *)sqlstmt.arr);

    EXEC SQL PREPARE S FROM :sqlstmt;
    EXEC SQL EXECUTE S USING :stunden, :login_time, :login_date, :user_name, :host_name;
    EXEC SQL COMMIT RELEASE;
    cout<<"Commiting Transaction..."<<endl;

  }
  else
    {
      // Holen der Daten nicht erfolgreich, Fehler ausgeben
      cout<<"Fehler beim holen der Daten!"<<endl;
      return false;
    }

  return true;
}

void sql_error(char *msg)
{
/* This is the ORACLE error handler.
 * Print diagnostic text containing error message,
 * current SQL statement, and location of error.
 */
    printf("\n%s", msg);
    printf("\n%.*s\n",
        sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in \"%.*s...\"\n",
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("on line %d of %.*s.\n\n",
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

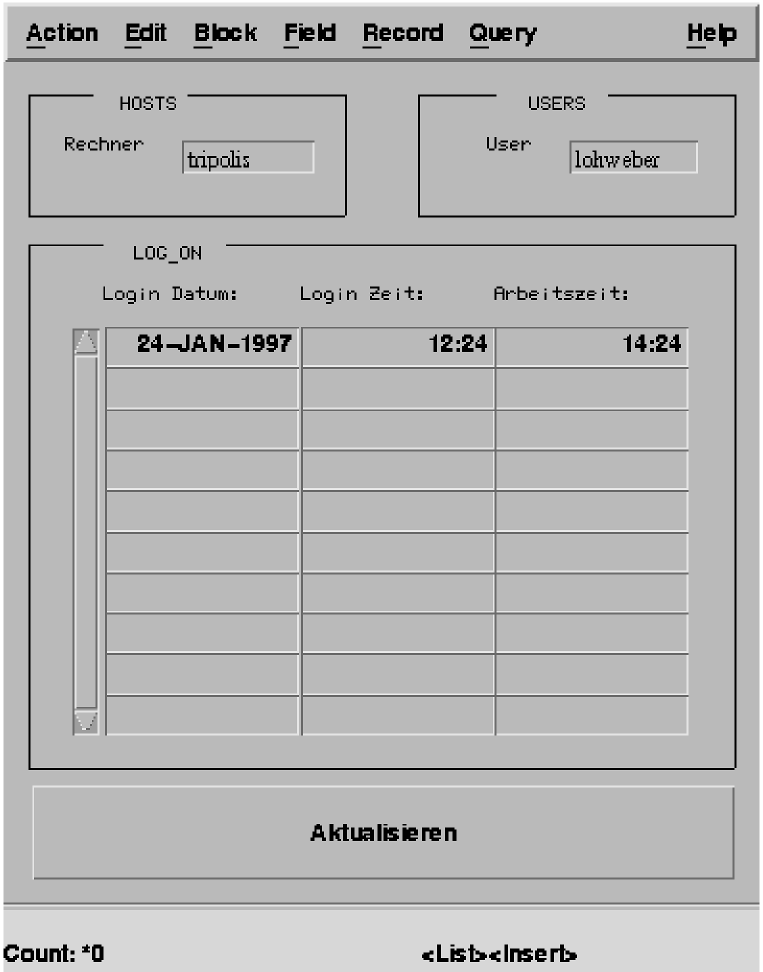


Abbildung 8.2: Die Oberfläche der Applikation.

8.5 Der Entwurf der Oberfläche

Die Oberfläche zum Darstellen der von "rusers" gelieferten und in der Datenbank abgespeicherten Daten wurde mit *Oracle Forms* erstellt. Abbildung 8.2 zeigt den Fensteraufbau der Applikation. Über eine "List of Values" (LOV) können Rechnername und User ausgewählt werden. Diese LOV's werden mit einem Mausklick in das jeweilige Textfeld für Rechnername und User geöffnet. Nach dem Schließen der LOV kann auch ein anderer Wert per Hand eingetragen werden. Hat man nun Rechner und User ausgewählt, wird die Liste der Login-Daten durch einen Mausklick auf den Push-Button "Aktualisieren" auf den neuesten Stand gebracht. Der dazu benötigte PL/SQL-Code ist in Abbildung 8.3 dargestellt. Da *Oracle Forms* die Darstellung von Tabellen alleine durch Display-Items anscheinend nicht unterstützt, wird auch das Feld 'idhost' aus der Tabelle 'log_on' selektiert und in ein Text-Item geschrieben, welches aber, mit einer dargestellten Breite von 0 Punkten, nicht mit angezeigt wird.

```
declare
  cursor log_on_zeilen is select log_on.idhost,
                                 log_on.login_date,
                                 log_on.login_time,
                                 log_on.working_time
                          from log_on, hosts, users
                          where (     (hosts.name = :hosts.name)
                                  and (users.name = :users.name)
                                  and (log_on.idhost = hosts.idhost)
                                  and (log_on.iduser = users.iduser) );

begin

go_block('log_on');

-- Zuerst alle vorhandenen Eintraege loeschen
loop
  first_record;
  delete_record;
  if :system.last_record = 'TRUE' then
    delete_record;
    exit;
  end if;
end loop;

-- Cursorvariable oeffnen
open log_on_zeilen;

-- Neue Eintraege in Tabelle schreiben
loop
  fetch log_on_zeilen into :log_on.idhost,
                           :log_on.login_date,
                           :log_on.login_time,
                           :log_on.working_time;
  if log_on_zeilen%found then
    next_record;
  else
    exit;
  end if;
end loop;

-- Cursorvariable schliessen
close log_on_zeilen;

commit;

end;
```

Abbildung 8.3: Der PL/SQL-Code zum Trigger “Aktualisieren”.

Kapitel 9

Locking für Daemon-Prozesse

Verfasser: Dilber Yavuz

9.1 Benutzung

Es soll eine Klasse realisiert werden, der überprüft, ob ein Daemon-Prozeß bereits gestartet wurde. Die Klasse heißt `CLock` und hat folgende Methoden:

`int Init()`
: Die Funktion initialisiert die Lock-Datei.

`void Exit()`
: Die Funktion schließt die geöffnete Lock-Datei.

`int CreateLock( char* ulock )`
: Die Funktion versucht die Lock-Datei zu sperren. Es liefert `TRUE`,wenn gesperrt werden konnte, sonst `FALSE`.

`int CloseLock( void )`
: Die Funktion hebt die Sperre auf, und liefert `TRUE`.

Die Klasse `CLock` (siehe Abbildung 9.2) wird folgendermaßen verwendet (siehe Abbildung 9.1): Die hier zu sperrende Datei heißt `testfile`. Wenn diese Datei gesperrt werden konnte, soll `Frage: 1` ausgegeben werden. Sonst soll ein String `name` ausgegeben werden.

9.2 Entwurf

Es wird eine Lock-Datei zum Lesen und Schreiben geöffnet (siehe Abbildung 9.3) und versucht diese zu sperren. Wenn gesperrt werden konnte, wird ein Positionszeiger für den nächsten `write`-Aufruf gesetzt und der String in die geöffnete Datei geschrieben. Sonst wird aus der geöffneten Datei die Prozeß-ID, der Benutzer- und Rechnername gelesen, der die Lock-Datei bereits gesperrt hat, und ausgegeben (siehe Abbildung 9.5). Um die Lock-Datei zu schließen wird ein Positionszeiger für den nächsten `write`-Aufruf gesetzt, die Sperre aufgehoben und die Datei geschlossen (siehe Abbildung 9.6).

9.3 Implementierung

Die Implementierung des Lockings für Daemon-Prozesse sieht wie folgt aus:

```
main()
{
   int d;
   int i;
   CLock testfile;
   char name[80];
   testfile.Init();
   if( testfile.CreateLock( name )==TRUE )
     {
        printf("\nFrage:\n%d",1);
        scanf("%d",d);
        i=testfile.CloseLock();
        testfile.Exit();
     }
   else
        printf("\n%60s\n",name);
}
```

Abbildung 9.1: Anwendungsbeispiel.

```
#ifndef _SPERRE_H
#define _SPERRE_H

class CLock
{
public:
    int Init();
    void Exit();
    int CreateLock( char* ulock );
    int CloseLock(void);
private:
int fd;
};
#endif
```

Abbildung 9.2: Die Klasse CLock.

```
#include <iostream.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/file.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/uio.h>
extern "C" {
  extern int gethostname(char *name, int namelen);
}

char* FILESPERRE="/home/pg/pg290/Zugriff";

int CLock::Init()
{
  fd = open( FILESPERRE, O_RDWR );//Datei zum Lesen und Schreiben "offnen
  if( fd == -1 )
    // Datei konnte nicht ge"offnet werden, Fehler
    return( FALSE );
  else
    // Datei konnte geoeffnet werden, alles OK
    return( TRUE );
}
```

Abbildung 9.3: Die Methode `Init`.

```
void CLock::Exit()
{
  close( fd );//Datei schlie"sen
}
```

Abbildung 9.4: Die Methode `Exit`.

```
int CLock::CreateLock( char* ulock )
{
  char BRstring[80],Benutz_Nam[L_cuserid],Rech_Nam[60];

  // BRstring mit Leerzeichen fuellen
  int i;
  for ( i = 0; i < 79;i++ )
     BRstring[i]=32;

   // Dateieintrag vorbereiten
  gethostname(Rech_Nam,60);
  cuserid(Benutz_Nam);

  //den String ausgeben
  sprintf( BRstring,"%20s%20s%10d",Benutz_Nam,getuid(),Rech_Nam,getpid() );

  // versuchen, die Lock-Datei zu sperren
  if( lockf( fd,F_TLOCK,60 ) == NULL )//Datei sperren, wenn nicht gesperrt ist

      {
        // Datei konnte gelockt werden
        lseek( fd,0,SEEK_SET );//Positionszeiger fuer den naechsten
                               //write-Aufruf setzen
        write( fd,BRstring,80 );//den String in die geoeffnete Datei schreiben
        return( TRUE );
      }
   else
      {
        // Datei konnte nicht gelockt werden
        lseek( fd,0,SEEK_SET );//Positionszeiger fuer den naechsten
                               //read-Aufruf setzen
        read( fd,BRstring,60 );//den String aus der geoeffneten Datei
                               //in den Puffer lesen
        sprintf( ulock,"%60s",BRstring );//gibt den Benutzer- und Rechnernamen
                                         //aus, der die Datei gesperrt hat
        return( FALSE );
      }
}
```

Abbildung 9.5: Die Methode `CreatLock`.

```
int CLock::CloseLock( void )
{
   char BRstring[80];
   int i;
   for( i = 0; i < 79; i++ )
      BRstring[i]=32;
   lseek( fd,0,SEEK_SET );//Positionszeiger fuer den naechsten
                          //write-Aufruf setzen
   write( fd,BRstring,60 );//den String, auf die BRstring zeigt,
                           //in die geoeffnete Datei schreiben
   lockf( fd,F_ULOCK,60 );//Sperre aufheben
   return( TRUE );
}
```

Abbildung 9.6: Die Methode `CloseLock`.

Kapitel 10

Evaluation

10.1 Kommunikationsinterface

Verfasser: Peter Ziesche

Das Kommunikationsinterface als vereinfachende Schicht zum Datenaustausch hat sich weitgehend bewährt. Die Entwicklung des einzelnen Komponenten des Föderierungssystems gestaltete sich dadurch einfacher. Während des Projektes sollte eine Einarbeitung in die Datenbanken O_2 und *Oracle* und die CORBA-Implementierung COOL stattfinden. Durch das *comi* konnte die Entwicklung der Adapter vollständig in einen datenbank- und einen kommunikationsspezifischen Teil aufgeteilt werden. Die Verwendung von Handlern für Operationen hat dabei zu einer hohen Lokalität von Funktionalitäten geführt.

Das *comi* kann für das Hauptprojekt übernommen werden. Es soll im Verlauf des zweiten Semesters in folgenden Punkten weiterentwickelt werden:

Dynamische Konfiguration
Das Föderierungssystem soll dynamisch konfigurierbar werden. Es sollen nicht immer alle Datenbank-Adapter laufen müssen. Ein Protokoll zwischen der Föderierungs-Datenbank und den Adaptern soll das An- und Abmelden von Adaptern erlauben. Insbesondere sollen die einzelnen Komponenten geordnet heruntergefahren werden können.

Unabhängigkeit von Operationen
Durch die Verwendung einer abstrakten Fabrik [G+95] zur Erzeugung von Handlern sind die möglichen Operationen mehr oder weniger fest codiert (siehe Beschreibung in 5.2.1). Diese Abhängigkeit ist unnötig, da das *comi* im Prinzip unabhängig davon ist. Um diesen Mangel zu beheben, muß ein Mechanismus gefunden werden, der das bisher erfolgreiche Konzept der Handler unterstützt, gleichzeitig aber die geforderte Unabhängigkeit bietet.

Vereinfachung des Datenaustausches
Zur Zeit werden Operationen in eine Menge von Strings serialisiert, die dann über den ORB verschickt und vom Empfänger wieder zu Operationen zusammengesetzt werden. Leider ist es so, daß ein Operationsobjekt sich selbst in einen STL-Container schreibt, der Versand aber über eine CORBA-Sequence erfolgt. Es sind also Konvertierungen der Form *Operation* → *STL-Container* → *CORBA-Sequence* → *STL-Container* → *Operation* erforderlich. Hier sollte unbedingt über Vereinfachungen nachgedacht werden.

Multithreading
Die Implementierung von CORBA setzt voraus, daß ein auf Nachrichten wartender Prozeß die Kontrolle an CORBA abgibt. Dazu begibt er sich in eine Warteschleife. Diese wird erst wieder

verlassen, wenn der Prozeß den Nachrichtenempfang beenden möchte. Dies könnte zu Problemen führen, wenn ein Datenbank-Adapter ein ähnliches Verfahren bei seiner lokalen Datenbank anwenden muß. Sollte dies der Fall sein, muß der Adapter wahrscheinlich zwei Threads erzeugen. Es ist jedoch noch nicht bekannt, ob dieser Schritt wirklich notwendig ist.

10.2 Das O_2-System

Verfasser: Ulf Radmacher, Sven Gerding

Die Entwicklung der autonomen Datenbankapplikation *Rusers* mit den von O_2 zu Verfügung gestellten Hilfsmitteln O_2C, O_2*Tools* und O_2*Look* hat sich als relativ einfach erwiesen. Es war in kurzer Zeit möglich, eine lauffähige Anwendung zu erstellen, welche die Anforderungen der Einarbeitungsaufgabe erfüllen konnte. Aufgrund der Arbeitsweise von O_2*Look* ist es allerdings nur mit großem Aufwand möglich, eine benutzerfreundlichere Arbeitsoberfläche zu Verfügung zu stellen. Um beispielsweise die Anzahl der sich öffnenden Fenster während eines Applikationsdurchlaufes zu reduzieren, müsste man die gesamte Klassenstruktur ändern ("verflachen"). Im Rahmen der Einarbeitungsaufgabe haben wir darauf verzichtet.

Der Export nach *C++* der von uns mit O_2C im Datenbankschema entwickelten Klassen hat sich als einfach erwiesen. O_2 hat hierbei für jede exportierte Klasse die entsprechenden Header- und Implementation-Files generiert, welche bei der Implementation des O_2-Datenbank-Adapter einfach eingebunden werden konnten. Probleme hat es nur gegeben, wenn Änderungen am Datenbankschema vorgenommen werden mussten, da nun der gesamte Exportvorgang erneut durchgeführt werden mußte. Alternativ hätte man das Datenbankschema auch über das *C++*-Binding erstellen können, wodurch der umständliche Export-Vorgang eingespart werden könnte, man würde allerdings auf den von O_2 erzeugten Rahmen verzichten.

Der Einsatz von Sniff bei der Implementierung des Datenbank-Adapters hat sich als problematisch erwiesen, so daß schließlich auf den Einsatz von Sniff ganz verzichtet wurde. Die Probleme bestanden konkret darin, daß für O_2 über das Tool `o2makegen` aus einer Konfigurationsdatei ein Makefile erzeugt wird. Dieses muß dann umständlich in das von Sniff vorgegebene Makefile eingebaut werden, wobei zusätzliche Modifikationen in beiden Makefiles nötig sind. Des weiteren setzt das O_2-Makefile voraus, daß sich der Quellcode sämtlicher zu compilierender Module in einem Verzeichnis befindet, was bedeutet das die Quellen von `comi` per Hand in das entsprechende O_2-Verzeichnis kopiert werden mußten. Sniff bietet im Zusammenhang mit der Anwendungsentwicklung unter C++ für O_2 keinerlei Arbeitserleichterung, sondern erfordert mehr Aufwand.

10.3 *Oracle*

Verfasser: Mischa Lohweber, Andreas Dinsch

10.3.1 Oracle als Datenbanksystem

Das Datenbanksystem von Oracle hat sich als zuverlässig und schnell erwiesen. Die Anbindung an C++, bzw. die Integration von 'Embedded SQL' in C++ Programmen stellt kein Problem dar. Der Precompiler, welcher *Pro*C* Code in C++ Code umwandelt, arbeitet einwandfrei. Zur initialen Erstellung der Tabellen stand nur die Shelloberfläche SQLPlus zur Verfügung. Hier wäre eine graphische, bzw. Menüorientierte Oberfläche wünschenswert. Die elektronische Dokumentation war nicht überzeugend, da das Programm fehleranfällig ist, unerwartet abstürzt und somit ein effektives Arbeiten schwer macht.

10.3.2 *Oracle Forms* als Eingabeoberfläche

Die Arbeit mit *Oracle Forms* erwies sich anfänglich als problematisch, da mit der vorhandenen Online-Dokumentation ein flüssiges Arbeiten nahezu unmöglich ist, was zum Einen an der instabilen Installation, zum Anderen an der schlechten Übersichtlichkeit der Oracle-Online-Help liegt. Hat man sich jedoch erst einmal an die z.T. recht umständliche Bedienung von *Oracle Forms* gewöhnt, läßt sich auch hiermit einigermaßen effektiv arbeiten. Sehr unübersichtlich allerdings ist die Struktur einer *Oracle*-Anwendung, da die verschiedenen Programmteile in `Triggern` und `Program Units` über die gesamte Form verteilt sind. Den Debug-Mode habe ich bis heute noch nicht verstanden, somit war natürlich die Fehlersuche nur mit recht großem Aufwand und einer Menge Glück verbunden. Die Tatsache, daß die Reihenfolge von UND-verknüpften Bedingungen in der Where-Klausel eines Select-Statements im PL/SQL-Code eines Triggers Einfluß auf das Ergebnis haben könnte, ließ mich fast an der Existenzberechtigung von *Oracle Forms* zweifeln.

Fazit: Mit *Oracle Forms* zu arbeiten ist zwar keine Freude, aber es geht.

Teil V

Das Abrechnungssystem

Kapitel 11

Die Anforderungsanalyse

Verfasser: Betül Ilikli, Xi Gao, Mario Ellebrecht, K.-H. Schulte

Es soll ein Abrechnungssystem für die Angiologie erstellt werden. Für die medizinische Sicht steht ein Therapiekontrollsystem zur Verfügung. Dieses Therapiekontrollsystem wurde im Rahmen der Diplomarbeit von Ingo Ullrich [Ull96] für die Radiologieabteilung des Krankenhauses in Wuppertal-Barmen entwickelt. Das Abrechnungssystem soll für die gleiche Umgebung ausgelegt sein und mit dem bestehenden Angiographiesystem kommunizieren.

11.1 Beschreibung der Systemumgebung

Das System soll auf *Sun*-Workstations unter *Solaris 2.5* laufen. Zur Datenhaltung soll die Datenbank *Oracle* [Ora94c] verwendet werden.

Es wird vorausgesetzt, daß ein Therapiekontrollsystem für die Angiographie parallel existiert, gemäß [Ull96]. Hier wird der komplette Behandlungsverlauf eines jeden Patienten in einer autonomen O_2-Datenbank [O296a] erfaßt.

Eine Anwendung zur Patientenaufnahme und Entlassung kann **nicht** als existent vorausgesetzt werden!

Als Abrechnungsverfahren soll das seit Januar 1996 gesetzlich vorgeschriebene Verfahren verwendet werden. Etwaige Vereinfachungen oder implizite Annahmen werden die Autoren gegebenenfalls im folgenden Text explizit angeben.

Die dem Angiographie-Therapiekontrollsystem zugrundeliegende O_2-Datenbank und die dem zu erstellenden Abrechnungssystem zugrundeliegende *Oracle*-Datenbank sollen später föderiert werden.

11.2 Das zu modellierende Szenario

Das Abrechnungssystem soll genau in ein Szenario, oder in der Sprechweise von *Jacobson* [JCJO92] in einen Use-Case, involviert sein. Dieses, bzw. dieser, soll im folgenden beschrieben werden.

Bezüglich des Problembereichs setzen die Autoren voraus, daß das Therapiekontrollsystem in einer eigenen Krankenhausabteilung, der Angiologie, existiert. Davon unabhängig gibt es die Verwaltung, in der das zu erstellende Abrechnungssystem eingesetzt werden soll.

Da in der Verwaltung kein Stammdatenverwaltungsprogramm existiert, haben sich die Autoren entschlossen die Aufnahme und Entlassung mit in der Abrechnungsanwendung zu realisieren.

11.2.1 Der Patientenfluß im Szenario

Der Patientenfluß verläuft in unserem Szenario folgendermassen:

1. Aufnahme des Patienten im Krankenhaus
2. Überweisung des Patienten zur Angiographie
3. Behandlung des Patienten
4. Entlassung des Patienten

11.2.2 Anforderungen an das Abrechnungssystem

Das Abrechnungssystem soll folgende Anforderungen erfüllen:

- Erfassung der Stammdaten des Patienten bei der Aufnahme und Speicherung dieser in der *Oracle*-Datenbank soll möglich sein.
- Die Kostenübernahmebestätigung (positiv oder negativ) der Krankenkasse soll relativ zum Patienten vermerkt werden können.
- Die Entlassung eines Patienten muß erfaßt werden können.
- Auf Wunsch soll bei der Entlassung eine Patientenrechnung erstellt werden können. Diese wird zu diesem Zeitpunkt allerdings noch nicht an die Krankenkasse weitergeleitet!
- Die Erstellung der (Sammel-) Rechnungen an die Kostenträger zum Stichtag muß möglich sein.
- Die Rechnungserstellung soll vollautomatisch geschehen. Dazu muß lokal eine Tabelle aller möglichen Fallpauschalpreise angelegt werden und auch die aktuellen Stationspflegesätze müssen lokal abgelegt sein.
- Für die interne Controllingabteilung des Krankenhauses soll zusätzlich eine Realkostenabrechnung erstellt werden können. Diese soll patientenbezogen aber auch stichtagsbezogen berechnet werden können.
- Solange das Programm ohne Anbindung an das Angiographiesystem läuft, soll es möglich sein, alle medizinischen Daten mit diesem System zu erfassen. Dies sind im einzelnen:
  - Der Materialverbrauch,
  - die Diagnose und
  - die verwendete Therapie.

  Für eine Diagnose soll des weiteren spezifizierbar sein, ob es sich um die Hauptkrankheit oder eine Nebenkrankheit handelt.

11.2.3 Die Benutzungsschnittstelle

Aus den Anforderungen ergibt sich unmittelbar der semantische Aufbau der Benutzungsschnittstelle. Das Abrechnungssystem soll eine Maske mit folgenden Menüpunkten als Hauptmenü anbieten:

- Aufnehmen eines Patienten
- Entlassen eines Patienten
- Erfassen der Kostenübernahmeerklärung

- Erfassen der medizinischen Daten
- Erstellung einer Abrechnung

Bei Anwahl des letzten Unterpunktes soll ein Menü gemäß folgendem Aufbau erscheinen:

- Erstellen der externen Stichtagsabrechnung
- Erstellen einer externen Patientenabrechnung
- Erstellen der internen Stichtagsabrechnung
- Erstellen einer internen Patientenabrechnung

Die Begriffe *intern* und *extern* sollen an der Stelle verwendet werden um die Benutzerfreundlichkeit zu steigern. Eine korrekte Beschreibung der Form -*abrechnung gemäß gesetzlicher Grundlage* beziehungsweise -*abrechnung gemäß Realkosten* würde die Übersichtlichkeit des Menüs zerstören. Darüberhinaus sind die verwendeten Begriffe deutlich besser für die Alltagssprache des Benutzers geeignet.

11.2.4 Einschränkungen des Abrechnungssystems

Im Abrechnungssystem werden die Autoren folgendes **nicht** modellieren:

- Abrechnungen für andere Stationen außer der Angiologie.
- Abrechnung von Nebenkrankheiten, solange die zugehörige Hauptkrankheit auf einer anderen Station behandelt wird.
- Abrechnung von Nebenkrankheiten, solange diese auf einer anderen Station behandelt werden muß.
- Zwischenabrechnungen mit dem Kostenträger vor der Entlassung.
- Abrechnung ambulanter Patienten.
- Abrechnung von Wahlleistungen durch Sonderentgelte.
- Die Absendung eines Kostenübernahmegesuches an die Krankenkasse des jeweiligen Patienten.

11.3 Der „grobe“ Programmablauf

Bei der **Anmeldung** eines Patienten werden die Stammdaten entweder neu aufgenommen oder bereits vorhandene Stammdaten ausgewählt. Die weitere Abhandlung des Anwendungsfalles geschieht mit Hilfe des Angiographiesystems.

Solange das System als unabhängige Anwendung laufen soll, muß es zu jedem Zeitpunkt möglich sein, **medizinische Daten** einzugeben. Die Diagnose soll mit Hilfe des ICD [icd96], die Therapie mit Hilfe des ICPM [icp96] eingegeben werden können. Die Materialbezeichnung kann natürlichsprachlich eingegeben werden, dazu auch der Preis und die Menge.

Sobald die **Kostenübernahmeerklärung** eintrifft, wird das Ergebnis in der Datenbank erfaßt. Ist diese negativ, so ist der Kostenträger der Patient selbst, sonst seine Krankenkasse.

Nach der Erfassung der **Entlassung** des Patienten *kann* eine Berechnung der Kosten für die gesamte Verweildauer des Patienten erstellt werden. Dies kann gemäß der gesetzlichen Grundlagen oder auch gemäß der im Krankenhaus anfallenden Realkosten geschehen.

Zu jedem **Stichtag** wird für jeden Kostenträger eine Rechnung über die erfaßten Leistungen für alle entlassenen und nicht bereits abgerechneten Patienten erstellt. Der Kostenträger kann entweder der Patient selber sein oder eine Krankenkasse. Für jede Krankenkasse soll zu jedem Stichtag nur eine Sammelabrechnung erstellt werden. Der sogenannte Stichtag ist genau dann, wenn der Benutzer den entsprechenden Menüpunkt anwählt. Zu diesem Tag kann ebenfalls eine interne Abrechnung gemäß Realkosten erstellt werden.

11.3.1 Der Algorithmus für die externe Abrechnung

Abbildung 11.1 beschreibt den Algorithmus für die Berechnung der externen Rechnungssumme pro Patient. Als Darstellungsform wurde ein *Nassi-Shneiderman*-Diagramm gewählt (im folgenden auch Struktogramm).

Die Stichtagsabrechnung ist eine Auflistung aller Rechnungen aller entlassenen Patienten, die bis zu diesem Tage noch nicht abgerechnet wurden.

11.3.2 Der Algorithmus für die interne Abrechnung

Die Berechnung aller Leistungen für einen Patienten geschieht folgendermaßen:

Der Rechnungsbetrag setzt sich aus der Summe der Preise aller Materialien (Katheter, Drähte, ...), aller Medikamente und dem Produkt aus der Verweildauer in Tagen und der internen Tagesbehandlungspauschale zusammen. Diese definiert die Controllingabteilung des Krankenhauses gemäß der internen Kostenanalyse.

Die Stichtagsabrechnung ist eine Auflistung aller Rechnungen aller entlassenen Patienten, die bis zu diesem Tage noch nicht abgerechnet sind.

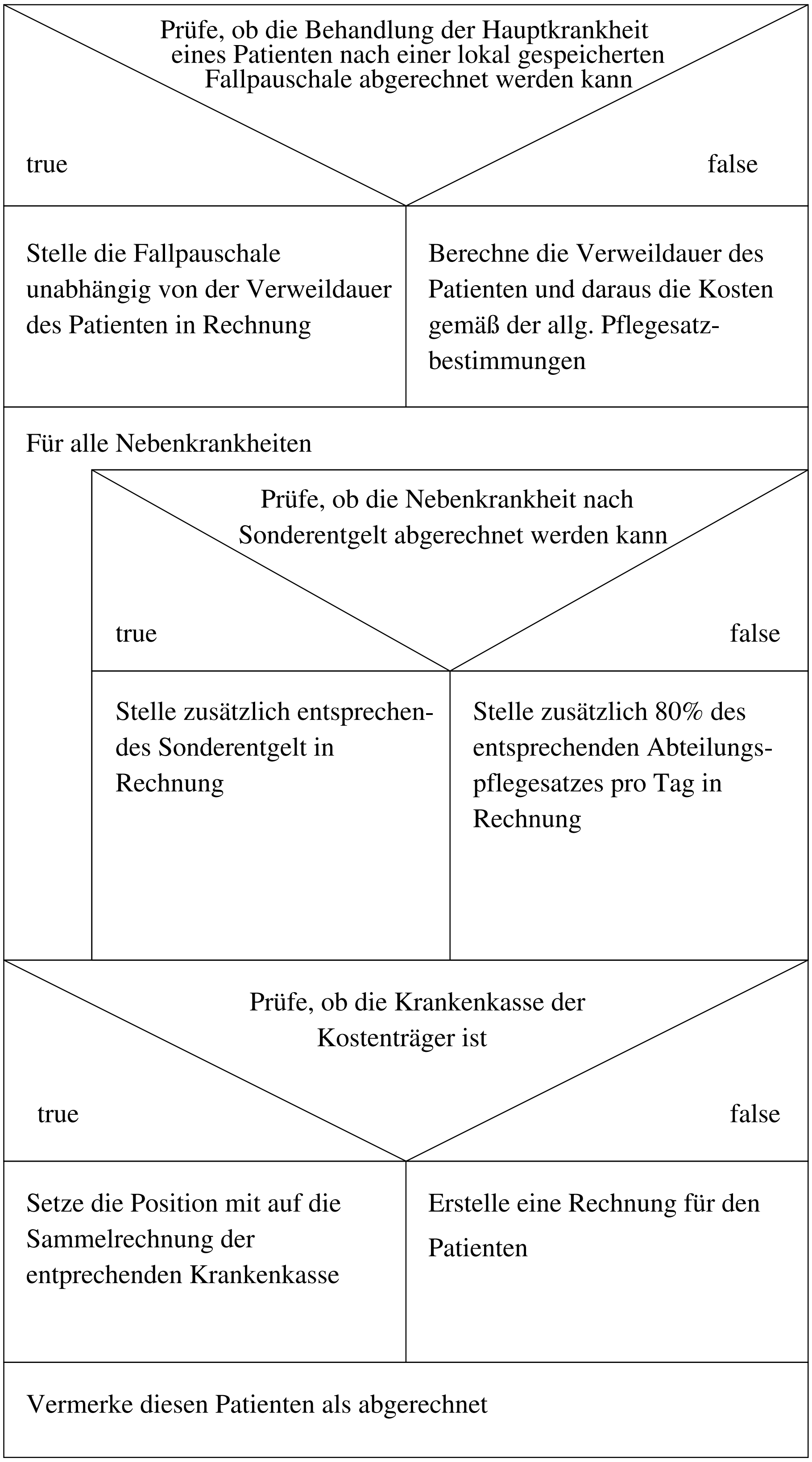


Abbildung 11.1: Das Struktogramm für die externe Abrechnung

Kapitel 12

Der Entwurf

Verfasser: Betül Ilikli, Xi Gao, Mario Ellebrecht, K.-H. Schulte

Den Entwurf beginnen die Autoren mit einem ER-Diagramm. Es soll die Datenhaltungsarchitektur des Systems darstellen. Im folgenden stellen die Autoren die Umsetzung der Architektur in relationale Datenbanktabellen in Form von graphisch dargestellten Tabellen dar. Anschließend soll noch ein Datenflußdiagramm der Veranschaulichung der Applikationsarchitektur dienen.

Für die dynamische Sicht der Applikation wird hier kein gesondertes Diagramm erstellt. Die Anwendung verweilt stets, unter Anzeige des Hauptmenüs, in einem Ruhezustand. Wird dann ein Menüpunkt angewählt, so wird die entsprechende dahinterliegende Prozedur ausgeführt. Darum soll für die dynamische Sicht die Bearbeitung der einzelnen Prozeduren in Form von *Nassi-Shneiderman*-Diagrammen [Bal96] ausreichen.

12.1 Das ER-Diagramm

Im ER-Diagramm haben die Autoren die statische Sicht auf die Struktur festgelegt. Für die Datenhaltung ergeben sich folgende Entitäten:

- Patientenstammdaten

  Hier sollen alle Patientendaten gespeichert werden, die längerfristig von Interesse sein werden, i. e. die Daten, die nicht an eine Krankheit gebunden sind.

- Abrechnungsdaten

  Hier sind alle Daten bezüglich einer Haupt- oder Nebenkrankheit gespeichert. Hier soll auch vermerkt werden, ob diese Behandlung bereits abgerechnet worden ist.

- ICD (international coding of diseases)

  Hier sollen alle ICD-Schlüssel gespeichert werden, sowie eine Krankheitsbezeichnung und gegebenenfalls eine Fallpauschalnummer, falls vorhanden.

- ICPM (international coding of procederes in medicine)

  Hier sollen alle ICPM-Schlüssel gespeichert werden, sowie eine Therapiebezeichnung.

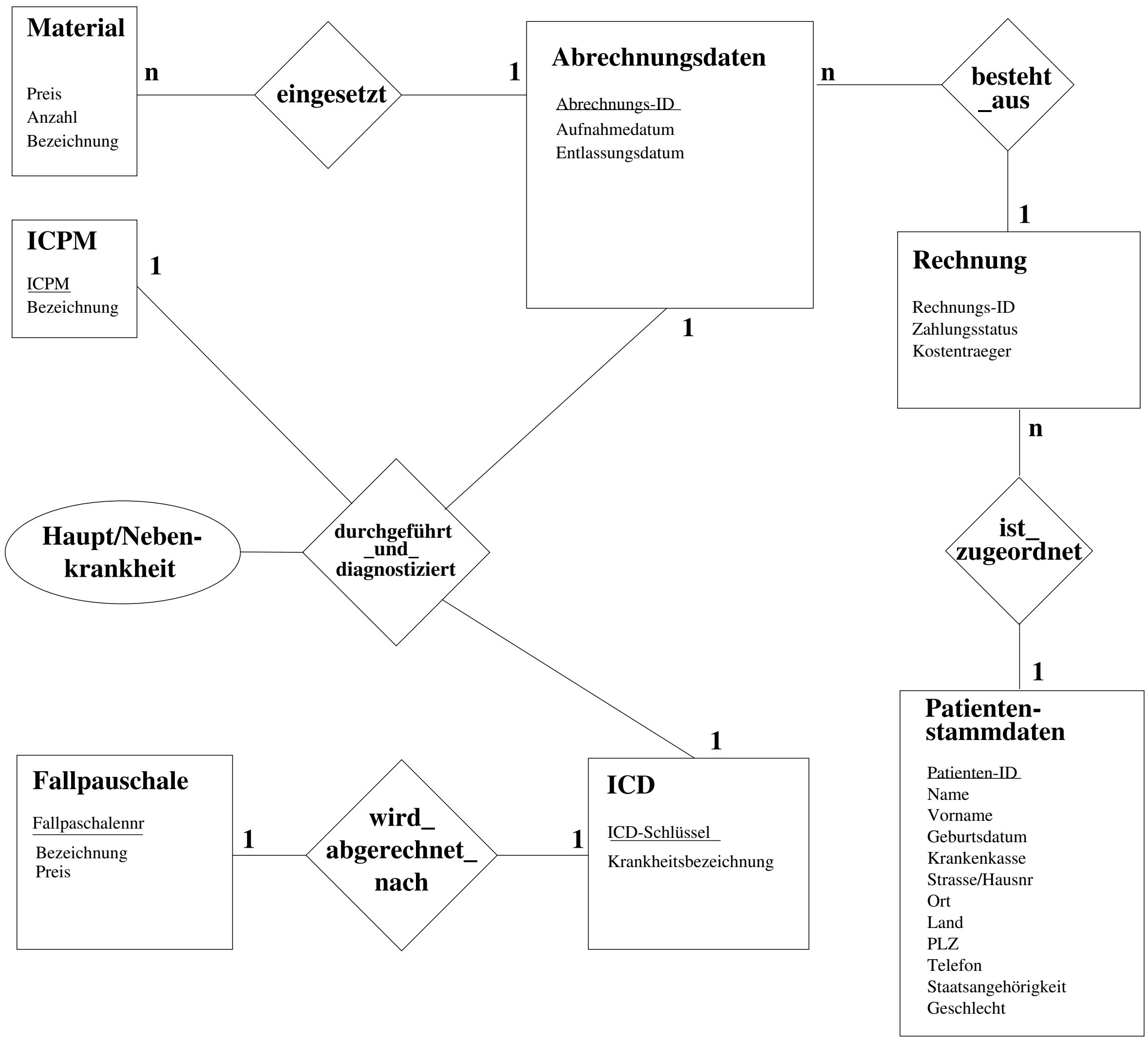
- Fallpauschale

  Hier soll jeder Fallpauschalnummer eine Bezeichnung und ein Preis gemäß gesetzlicher Grundlage zugeordnet werden. Unsere Terminologie umfaßt hier sowohl Fallpauschalen als auch Sonderentgelte.

- Material

  Unter Materialien sind an dieser Stelle alle die Dinge zu verstehen, die bei der Behandlung des Patienten verbraucht werden, i. e. also sowohl Katheter, Drähte, usw. als auch jegliche Form von Medikamenten. Hier sollen alle nötigen Daten bezüglich des Materials gespeichert werden, die für die Abrechnung eines Krankheitsfalles nötig sind.

Die graphische Veranschaulichung zeigt Abbildung 12.1.



Legende:

- Die Notation der Entitäten ist der OMT-Notation [RBP+91] angelehnt.
- Die unterstrichenen Attribute stellen die Schlüsselattribute dar.
- Das Attribut „Haupt/Nebenkrankheit“ an der ternären Relation gibt an, ob die Relation mit der Haupt- oder einer Nebenkrankheit verbunden ist. Bei der Modellierung muß beachtet werden, daß es immer genau eine Hauptkrankheit geben muß.

Abbildung 12.1: Das ER-Diagramm

12.2 Die Datenbanktabellen

Im folgenden legen die Autoren die verschiedenen Tabellen, gemäß des ER-Diagramms, der Datenbank fest. Siehe dazu die folgenden Abbildungen:

Patientenstammdaten		**NOT NULL**	**PK/FK**	**Unique**
Patienten-ID	NUMBER	x	PK	x
Name	VARCHAR2	x		
Vorname	VARCHAR2	x		
Geburtsdatum	DATE	x		
Geschlecht	CHAR(1)			
Staatsangehörigkeit	VARCHAR2			
Straße	VARCHAR2			
Hausnummer	VARCHAR2			
PLZ	NUMBER			
Ort	VARCHAR2			
Land	VARCHAR2			
Telefon	VARCHAR2			
Telefax	VARCHAR2			
Krankenkasse	VARCHAR2	x		

Abrechnungsdaten		**NOT NULL**	**PK/FK**	**Unique**
Abrechnungs-ID	NUMBER	x	PK	x
Patienten-ID	NUMBER	x	FK	
Aufnahmedatum	DATE			
Entlassungsdatum	DATE			
Kostenübernahme	CHAR(1)			
Abrechnungsstatus	CHAR(1)			

Material		**NOT NULL**	**PK/FK**	**Unique**
Abrechnungs-ID	NUMBER	x	FK	
Bezeichnung	VARCHAR	x		
Preis	FLOAT	x		
Anzahl	NUMBER	x		

Abbildung 12.2: Die Datenbanktabellen (1 von 2)

ICD		NOT NULL	PK/FK	Unique
ICD	VARCHAR2	x	PK	x
Bezeichnung	VARCHAR2	x		x

ICPM		NOT NULL	PK/FK	Unique
ICPM	VARCHAR2	x	PK	x
Bezeichnung	VARCHAR2	x		x

Fallpauschale		NOT NULL	PK/FK	Unique
Fallpauschalennummer	NUMBER	x	PK	x
Bezeichnung	VARCHAR2	x		x
Preis	FLOAT			
ICD	VARCHAR2	x	FK	
FP_or_SE	CHAR(1)			

HK_NK_Zuordnung		NOT NULL	PK/FK	Unique
Abrechnungs-ID	NUMBER	x	FK	x
ICPM	VARCHAR2	x	FK	
Fallpauschalennummer	VARCHAR2	x	FK	
HK_or_NK	CHAR(1)			

Legende:

- Die Bezeichnung im oberen linken Feld entspricht dem Namen der Datenbanktabelle.
- Darunter sind jeweils die Attributnamen angegeben, deren Datentypen sowie eventuelle Einschränkungen angekreuzt.
- Die Abkürzungen PK und FK stehen für `Primary Key` bzw. `Foreign Key`.
- Die Semantik der Begriffe `NOT NULL`,`Primary Key`, `Foreign Key` und `Unique` entspricht der jeweiligen aus dem relationalen Datenmodell.

Abbildung 12.3: Die Datenbanktabellen (2 von 2)

Einige Attribute dieser Tabellen unterliegen allerdings Bedingungen oder stehen in Beziehung zu anderen. Diese sollen, wie auch die Semantik der nicht intuitiv klaren Attribute, im folgenden stichpunktartig erläutert werden:

- Die Schlüsselattribute (unterstrichen in den Tabellen) müssen jeweils mit eindeutigen Werten belegt werden.
- Zu jeder `PatientenID` einer beliebigen Tabelle muß ein `Patientenstammdaten`-satz existieren, i. e. jeder Patient muß im Krankenhaus aufgenommen sein (referentielle Integrität).

- Zu jeder `PatientenID` muß genau ein Eintrag in der Tabelle `HK_NK_Zuordnung` existieren mit dem Attribut `HK_or_NK` gleich `true`.
- Zur Tabelle `Abrechnungsdaten`
  - Das `Aufnahmedatum` und das `Entlassungsdatum` müssen in chronologischer Reihenfolge gemäß der intuitiven Bedeutung der Begriffe liegen.
  - Die `Kostenübernahme` und der `Abrechnungsstatus` sollen per default auf `false` gesetzt werden.
  - `Abrechnungsstatus` gleich `true` bedeute daß diese Krankheit bereits abgerechnet wurde.
  - `Kostenübernahme` gleich `true` bedeute daß die Krankenkasse des Patienten diese Behandlung bezahlen werde.
- Zur Tabelle `Material`
  - Für jede verwandte `Abrechnungs-ID` muß ein entsprechender Eintrag in der Tabelle `Abrechnungsdaten` existieren.
- Zur Tabelle `FP_or_SE`
  - Jede Fallpauschale bzw. jeder Sonderentgelt ist genau einem `ICD` zugeordnet (referentielle Integrität durch den Fremdschlüssel `ICD`).
- Zur Tabelle `HK_NK_Zuordnung`
  - Für jede verwandte `Fallpauschalennummer` muß ein entsprechender Eintrag in der Tabelle `FP_or_SE` existieren.
  - Für jeden verwandten `ICPM` muß ein entsprechender Eintrag in der Tabelle `ICPM` existieren.
  - Für jede verwandte `Abrechnungs-ID` muß ein entsprechender Eintrag in der Tabelle `Abrechnungsdaten` existieren.
  - Ist `HK_or_NK` gleich `true`, so bedeute dies, daß sich diese Zuordnung auf die Hauptkrankheit beziehe. Ein `false` drückt die Zuordnung zu einer Nebenkrankheit aus.

12.3 Das Datenflußdiagramm

Das Datenflußdiagramm in Abbildung 12.4 stellt die funktionale Sicht auf das System dar. Hier sollen die „Daten"-Aspekte erklärt werden.

Jede Tabelle wird hier als Datenspeicher dargestellt. Die angegebenen Prozesse stellen im wesentlichen die Funktionalität für die entsprechenden Menüpunkte zur Verfügung.

Die Speicher „externe Rechnung" und „interne Rechnung" sind lediglich Hilfsspeicher, um den Datenaustausch zwischen den beteiligten Funktionen im Datenflußdiagramm darstellen zu können. Die Stichtagsabrechnungen sollen jeweils die Patientenrechnungen benutzen.

Auf die explizite Bezeichnung der fließenden Daten wurde in diesem Diagramm der Übersichtlichkeit halber verzichtet. Durch die genaue Spezifizierung der in den einzelnen Tabellen enthaltenen Attribute und deren sinngebende Namen sollte der notwendige Datenfluß durch die Beschreibung der Prozesse klar werden.

Terminologie- und Notationserklärung zum Datenflußdiagramm:

- Schnittstellen zur Umwelt durch die Daten in das Modell hineinfließen, werden als *Quellen* bezeichnet.
- Schnittstellen zur Umwelt durch die Daten aus dem Modell herausfließen, werden als *Senken* bezeichnet.

- *Quellen* und *Senken* werden durch Rechtecke dargestellt.
- Die funktionalen Elemente des Modells werden mit *Prozeß* bezeichnet.
- *Prozesse* werden durch Ellipsen dargestellt.
- Jegliche Form von Daten, die im Diagramm explizit dargestellt werden muß, wird als *Datenspeicher* bezeichnet.
- *Datenspeicher* werden durch zwei parallele Linien dargestellt.
- Fließende Daten werden als Pfeile notiert.

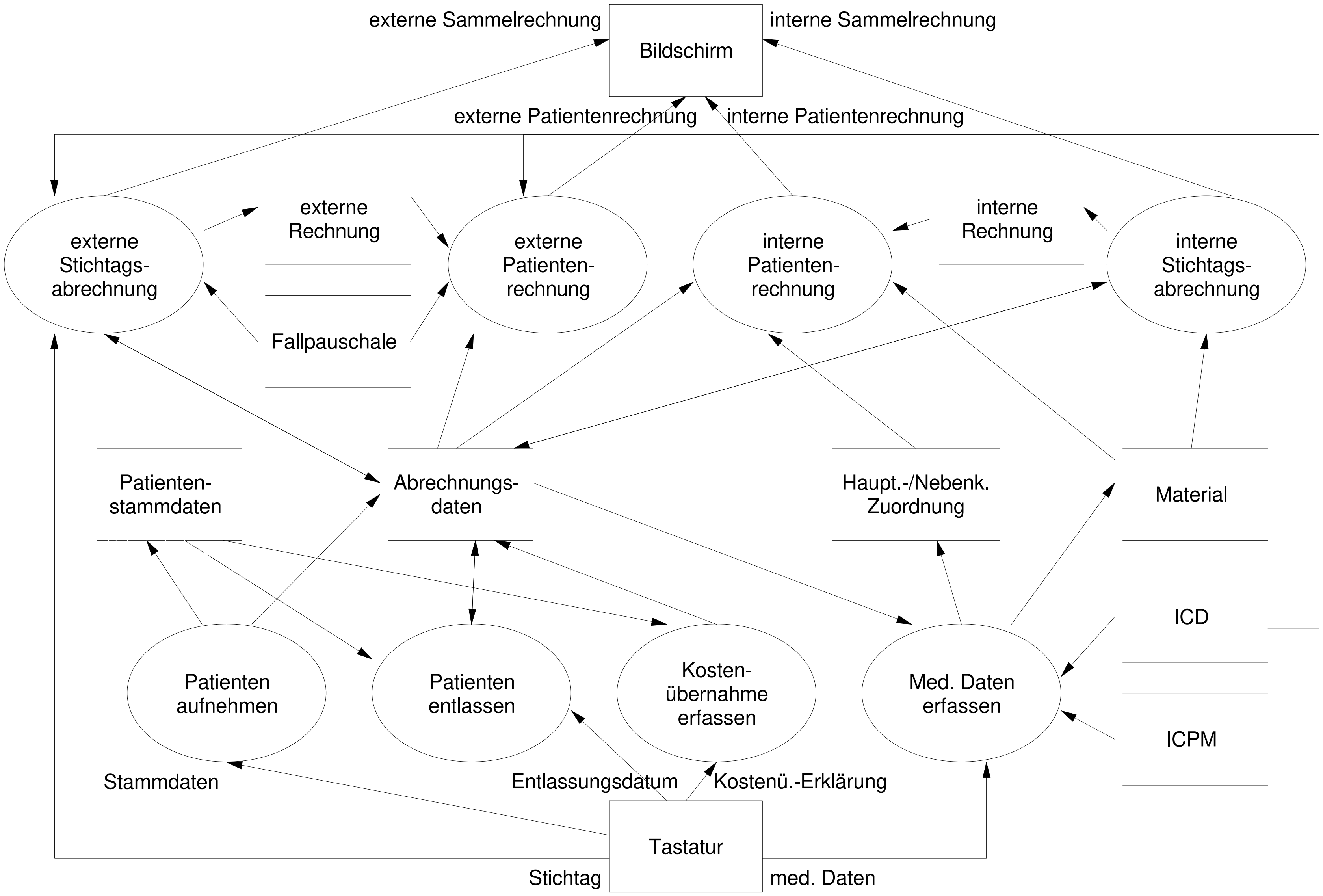


Abbildung 12.4: Das Datenflußdiagramm

12.4 Die Struktogramme

Der Feinentwurf beschränkt sich auf die Darstellung der Algorithmen der einzelnen Prozesse. Zur Darstellung werden Struktogramme in der Form von *Nassi-Shneiderman*-Diagrammen [Bal96] verwendet.

12.4.1 Die Patientenaufnahme

Die Modellierung des Prozesses „Patienten aufnehmen“ in der Funktion „PatientenAufnehmen“.

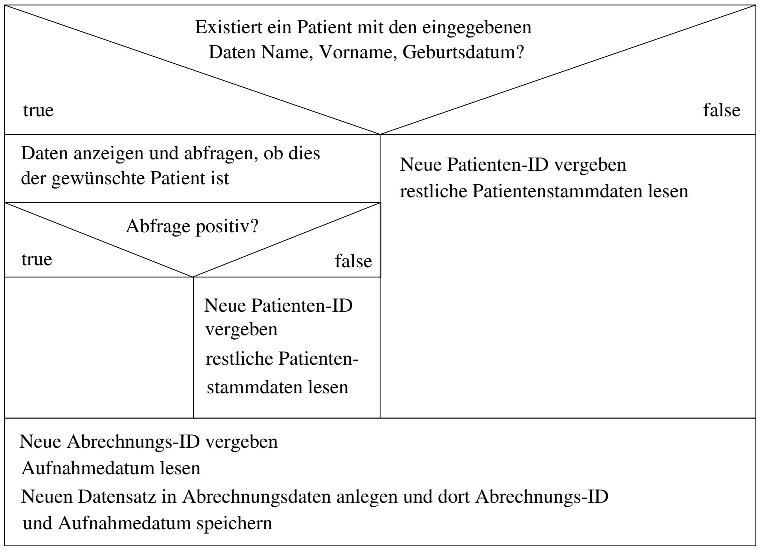


Abbildung 12.5: Das Struktogramm „Patienten aufnehmen“

12.4.2 Die Patientenentlassung

Die Modellierung des Prozesses „Patienten entlassen“ in der Funktion „PatientenEntlassen“.

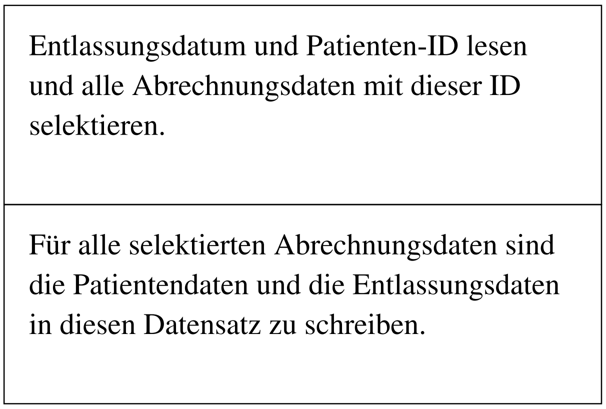


Abbildung 12.6: Das Struktogramm „Patienten entlassen“

12.4.3 Die Erfassung der Kostenübernahmeerklärung

Die Modellierung des Prozesses „Kostenübernahme erfassen“ in der Funktion „KostenuebernahmeErfassen“. Hier soll, sobald der entsprechende Brief der Kasse eintrifft, erfasst werden, daß die Krankenkasse des Patienten die Kosten für seine Behandlung übernimmt.

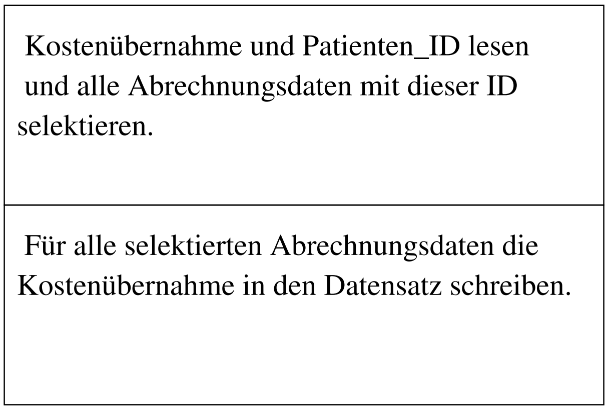


Abbildung 12.7: Das Struktogramm „Kostenübernahme erfassen“

12.4.4 Die Erfassung der medizinischen Daten

Die Modellierung des Prozesses „med.Daten erfassen“ in der Funktion „MedizinischeDatenErfassen“.

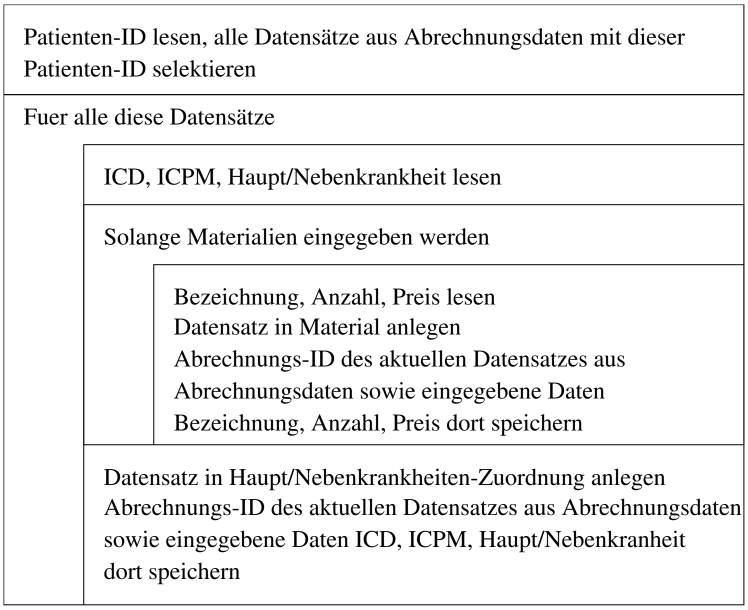


Abbildung 12.8: Das Struktogramm „medizinische Daten erfassen“

12.4.5 Die externe Patientenrechnung

Die Modellierung des Prozesses „externe Patientenrechnung“ in der Funktion „ExternePatientenrechnungErstellen“.

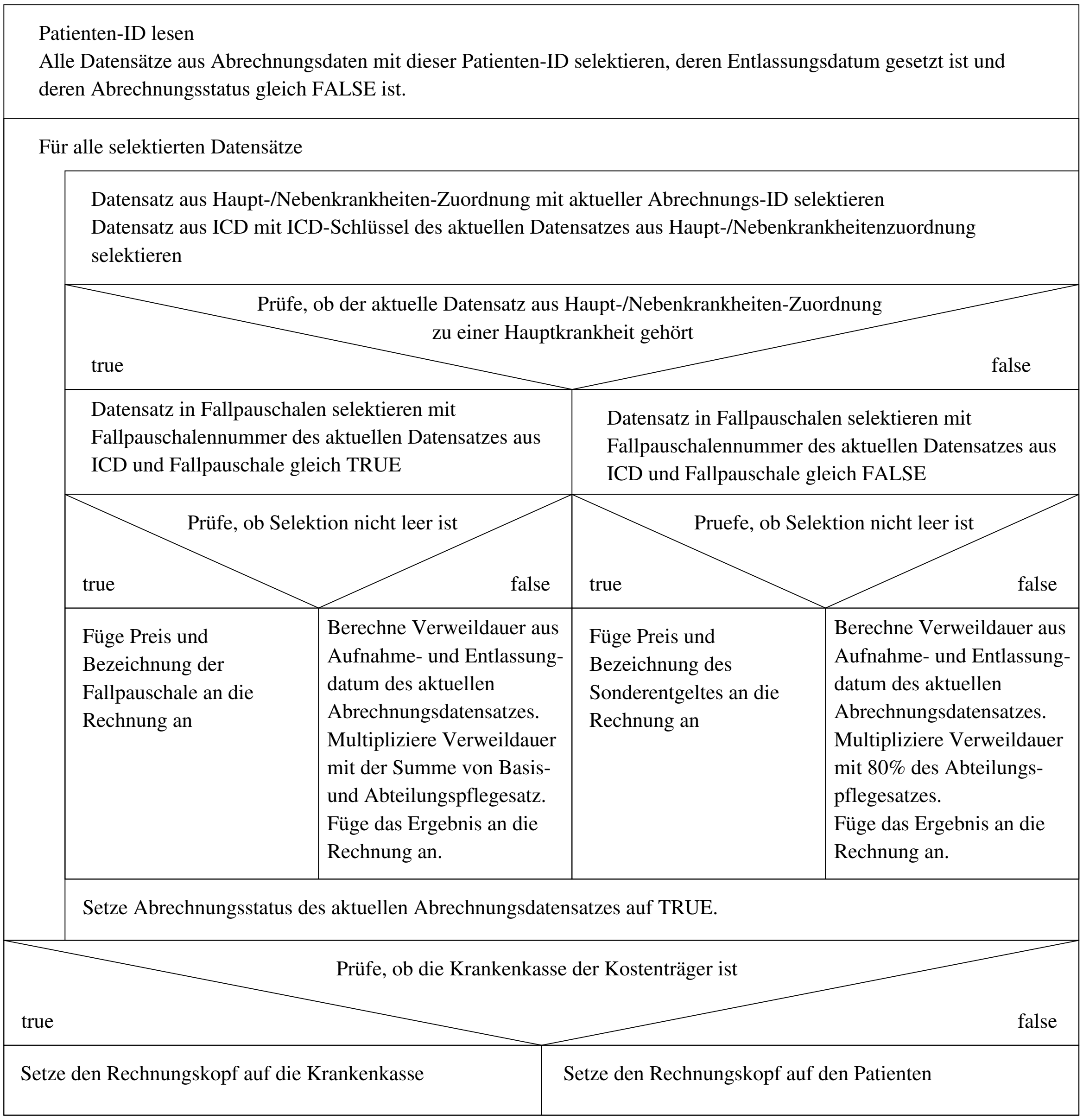


Abbildung 12.9: Das Struktogramm „externe Patientenrechnung“

12.4.6 Die externe Stichtagsabrechnung

Die Modellierung des Prozesses „externe Stichtagsabrechnung“ in der Funktion „ExterneStichtagsabrechnungErstellen“.

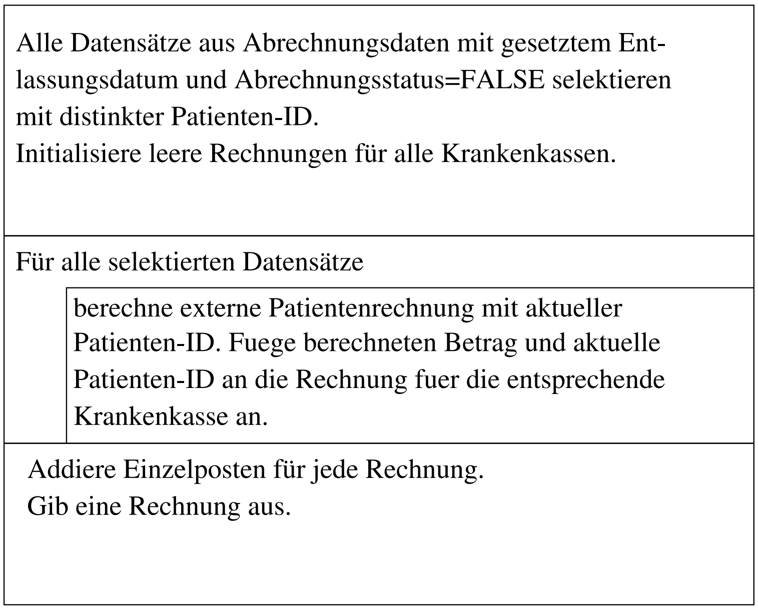


Abbildung 12.10: Das Struktogramm „externe Stichtagsabrechnung“

12.4.7 Die interne Patientenrechnung

Die Modellierung des Prozesses „interne Patientenrechnung“ in der Funktion „InternePatientenrechnungErstellen“.

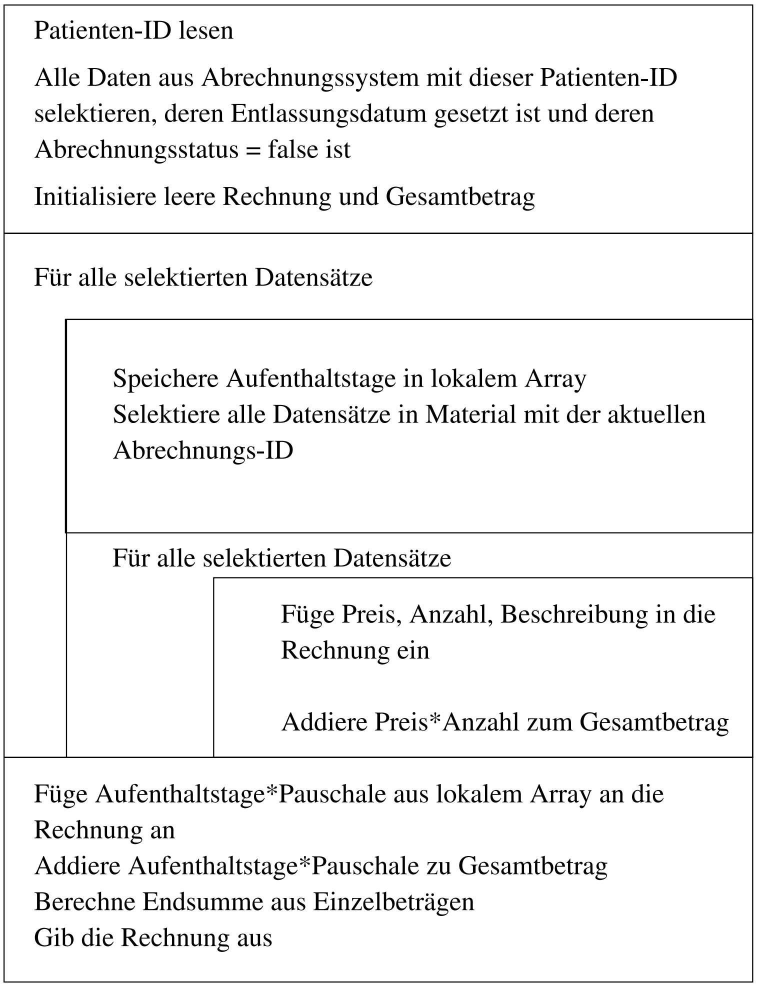


Abbildung 12.11: Das Struktogramm „interne Patientenrechnung“

12.4.8 Die interne Stichtagsabrechnung

Die Modellierung des Prozesses „interne Stichtagsabrechnung“ in der Funktion „InterneStichtagsabrechnungErstellen“.

Alle Datensätze aus Abrechnungsdaten mit gesetztem Entlassungsdatum und Abrechnungsstatus=FALSE selektieren mit distinkter Patienten-ID.
Initialisiere leere Rechnung.

Für alle selektierten Datensaetze

berechne interne Patientenrechnung mit aktueller Patienten-ID. Füge berechneten Betrag und aktuelle Patienten-ID an die Rechnung an.

Addiere Einzelposten.
Gib eine Rechnung aus.

Abbildung 12.12: Das Struktogramm „interne Stichtagsabrechnung“

Kapitel 13

Die Implementierung

Verfasser: Betül Ilikli, Xi Gao, Mario Ellebrecht, K.-H. Schulte

In diesem Kapitel wollen die Autoren den gesamten Prozeß der Implementierung dokumentieren. Es soll insbesondere der prinzipielle Weg bis zum fertigen Produkt aufgezeigt werden, wobei auf technische Details weitestgehend verzichtet werden soll.

Sollten aus implementierungstechnischen Gründen Anpassungen am Entwurf vorgenommen werden müssen, so sollen diese im vorliegenden Kapitel dokumentiert werden. Die Verfasser wollen dadurch sicherstellen, einen werkzeug- und produktunabhängigen Entwurf im gleichnamigen Kapitel beizubehalten.

13.1 Die Implementierumgebung

In der Analyse wurde bereits die Datenbank *Oracle* [Ora94c] als Basis für die Datenhaltung festgelegt. Auf dieser Basis wurde entschieden zur Implementierung das Produkt *Oracle Forms* [Ora94c] zu verwenden.

Oracle Forms arbeitet auf *Oracle*-Datenbanken und erlaubt das halbautomatische Erzeugen von grafischen Oberflächen. Diese Oberflächen setzen sich dann insbesondere aus Spalten der zugrundeliegenden Datenbanktabelle zusammen. Mit Hilfe der Sprache *PL/SQL* [Ora94c], einer Weiterentwicklung von *SQL*, können für die verschiedenen Elemente der Oberflächen zum Beispiel Trigger und Joins definiert werden. Auch das Gestalten von grafischen Menüs, unabhängig von Datenbankspalten, ist möglich. Für weitere Informationen über die verwendeten Produkte der Firma *Oracle Corporation* wird auf das ausführliche Handbuch [Ora94c] verwiesen.

13.2 Anlegen der Datenbanktabellen

Die Datenbanktabellen wurden gemäß dem Entwurf erzeugt, jedoch wurde der Datentyp BOOL durch einen CHAR(1) substituiert. In der Realisierung haben die Autoren jeweils eine zweiwertige Menge als Universum festgelegt wobei die konkreten Werte kontextsensitiv gewählt worden sind.

Ergänzend zu den Tabellen sind aus implementierungstechnischen Gründen Sequenzen angelegt worden. Dieses sind Elemente die *Oracle* zur Verfügung stellt um einen Wert vom Typ NUMBER fortlaufend zu vergeben. Die Vergabe ist somit automatisiert und bedarf keiner weiteren Kontrolle mehr.

Im folgenden ist das SQL-Skript, mit Hilfe dessen die Tabellen und Sequenzen angelegt worden sind, abgedruckt.

```
create sequence PatientenSequence
       start with 1 increment by 1 nomaxvalue minvalue 1 nocycle;

create sequence AbrechnungsSequence
       start with 1 increment by 1 nomaxvalue minvalue 1 nocycle;

create sequence FallpauschalenSequence
       start with 1 increment by 1 nomaxvalue minvalue 1 nocycle;

create table Patientenstammdaten (
       PatientenID NUMBER CONSTRAINT pk_PID PRIMARY KEY,
       Name VARCHAR2(100) CONSTRAINT nn_Name NOT NULL,
       Vorname VARCHAR2(100) CONSTRAINT nn_Vorname NOT NULL,
       Geburtsdatum DATE DEFAULT SYSDATE CONSTRAINT nn_Datum NOT NULL,
       Geschlecht CHAR(1) DEFAULT 'w' CONSTRAINT ch_Gesch CHECK (Geschlecht IN ('m','w')),
       Staatsangehoerigkeit VARCHAR2(100),
       Strasse VARCHAR2(100) DEFAULT '',
       Hausnummer VARCHAR2(100) DEFAULT '',
       PLZ NUMBER DEFAULT 0,
       Ort VARCHAR2(100) DEFAULT '',
       Land VARCHAR2(100) DEFAULT 'Deutschland',
       Telefon VARCHAR2(100) DEFAULT '',
       Telefax VARCHAR2(100) DEFAULT '',
       Krankenkasse VARCHAR2(100) DEFAULT 'unbekannt' CONSTRAINT nn_Krank NOT NULL
       );

create table Abrechnungsdaten (
       AbrechnungsID NUMBER CONSTRAINT pk_AID PRIMARY KEY,
       PatientenID NUMBER CONSTRAINT fk_PID REFERENCES Patientenstammdaten
                   CONSTRAINT nn_PID NOT NULL,
       Aufnahmedatum DATE CONSTRAINT nn_ADatum NOT NULL,
       Entlassungsdatum DATE DEFAULT SYSDATE,
       Kostenuebernahme CHAR(1) DEFAULT 'f'
                   CONSTRAINT ch_Kost CHECK (Kostenuebernahme IN ('t','f')),
       Abrechnungsstatus CHAR(1) DEFAULT 'f'
                   CONSTRAINT ch_Abr CHECK (Abrechnungsstatus IN ('t','f'))
       );

create table Material (
       AbrechnungsID NUMBER CONSTRAINT fk_AID REFERENCES Abrechnungsdaten
                   CONSTRAINT nn_AID NOT NULL,
       Bezeichnung VARCHAR2(100) DEFAULT '' CONSTRAINT nn_Bez NOT NULL,
       Preis NUMBER(10,2) DEFAULT 0 CONSTRAINT nn_Preis NOT NULL,
       Anzahl NUMBER DEFAULT 0 CONSTRAINT nn_Anza NOT NULL
       );

create table ICD (
       ICD varchar(100) CONSTRAINT pk_ICD PRIMARY KEY,
       Bezeichnung varchar(300) CONSTRAINT nn_ICD_bez NOT NULL
       );

create table ICPM (
       ICPM varchar(100) CONSTRAINT pk_ICPM PRIMARY KEY,
       Bezeichnung varchar(300) CONSTRAINT nn_ICPM_bez NOT NULL
```

```
        );

create table Fallpauschale (
        Fallpauschalennummer varchar(100) CONSTRAINT pk_FP PRIMARY KEY,
        Bezeichnung varchar(300) CONSTRAINT nn_FP_bez NOT NULL,
        Preis FLOAT,
        ICD varchar(100) REFERENCES ICD CONSTRAINT fpicdnn NOT NULL,
        FP_or_SE char(1) CONSTRAINT nn_F_or_S CHECK ( FP_or_SE IN ( 'h', 'n' ) ),
        );

create table HK_NK_Zuordnung (
        AbrechnungsID number REFERENCES Abrechnungsdaten CONSTRAINT nn_AIDHNK NOT NULL,
        ICPM varchar(100) REFERENCES ICPM CONSTRAINT nn_ICPMHNK NOT NULL,
        HK_or_NK char(1) DEFAULT 'h'
                    CONSTRAINT nn_HK_or_NK CHECK ( HK_or_NK IN ( 'h', 'n' ) ),
        Fallpauschalennummer varchar(100) REFERENCES Fallpauschale
                    CONSTRAINT nn_FPHNK NOT NULL
        );
```

13.2.1 Initialisieren der Datenbanktabellen

Einige Datenbanktabellen sind von den Autoren unabhängig von der Applikation „per Hand“ mit Datensätzen gefüllt worden, da die Eingabe dieser Daten nicht zu den Aufgabenbereichen eines Abrechnungssystems zählt. Dies trifft für die Tabellen *ICD*, *ICPM* und *Fallpauschale* zu. Die zugrundeliegenden Informationen entstammen dem WWW-Server der Universität München, [icd96], [icp96], [fps96].

Die Umwandlung der Daten wurde mit Hilfe von *Suchen- und Ersetzen- Funktionen* eines Texteditors vorgenommen. Im einzelnen wurden also die Formatierbefehle der Listen in INSERT-Befehle, gemäß SQL, gewandelt.

13.3 Die Applikation

In diesem Unterkapitel soll die Applikation vorgestellt werden, i. e. die Erscheinung gegenüber dem Benutzer und nicht die Funktionalität als solches.

Um zum einen den ergonomischen Anforderungen der Leser und zum anderen auch der Art und Weise der Erstellung gerecht zu werden, soll sich die Beschreibung stets an den grafischen Oberflächen, i. e. den Eingabemasken, orientieren. Dies soll insbesondere durch Screenshots ermöglicht werden.

13.3.1 Unterschiede zwischen Implementierung und Entwurf

Die Autoren konnten aus zeitlichen Gründen die im Entwurf beschriebene interne und externe Stichtagsabrechnung nicht implementieren.

13.3.2 Die Layoutkonventionen

Um das Produkt in einem einheitlichen Look-and-Feel erscheinen zu lassen, haben die Autoren an dieser Stelle eine Menge von Layoutkonventionen vorgenommen. Diese sollen für alle Fenster der Applikation gelten.

- Jedes Fenster trägt den Namen `Abrechnungssystem`.
- Am unteren Rand der Fenster befinden sich jeweils drei Buttons mit der Beschriftung `OK`, `Hauptmenü` bzw. `Cancel`.
- Am oberen Rand der Fenster befindet sich jeweils der Titel des Menüpunktes in einem Textfenster mit Hintergrundfarbe `white`.
- Die Hintergrundfarbe des Fensters ist `gray`.
- Die Hintergrundfarbe der Items ist `white`.
- Die Schriftart im gesamten Fenster ist `new century schoolbook`.
- Die Schriftfarbe im gesamten Fenster ist `black`.
- Die Schriftgröße im gesamten Fenster, ausgenommen die der Überschrift, beträgt `10 pt`.
- Die Schriftgröße der Überschrift beträgt `20 pt`.
- Items, die nur vom System ausgefüllt werden können und somit dem Benutzer nur zur Information dienen, werden mit einem abgerundeten Rechteck umrandet.
- Die Größe des Fensters sollte genau auf die Größe der Form abgestimmt sein. Sollte das bei sehr großen Forms nicht machbar sein, so ist das Fenster mit einem Scrollbar zu versehen. Die Größe der Fenster braucht somit vom Benutzer nicht mehr verändert werden können.

13.3.3 Die Menüs

Zunächst wollen die Autoren die Menüs vorstellen. Da die Semantik der einzelnen Menüpunkte bereits im Entwurf geklärt wurde und sich die Implementierung genau daran hält, soll an dieser Stelle nicht mehr auf die Semantik der Unterpunkte eingegangen werden. Hier soll lediglich das grafische Layout präsentiert, sowie an einem Beispiel gezeigt werden, wie ein solcher Menüaufruf technisch realisiert wurde.

Das Hauptmenü

Der folgende Screenshot zeigt das Hauptmenü, so wie es bei Start des Programms unmittelbar erscheint.

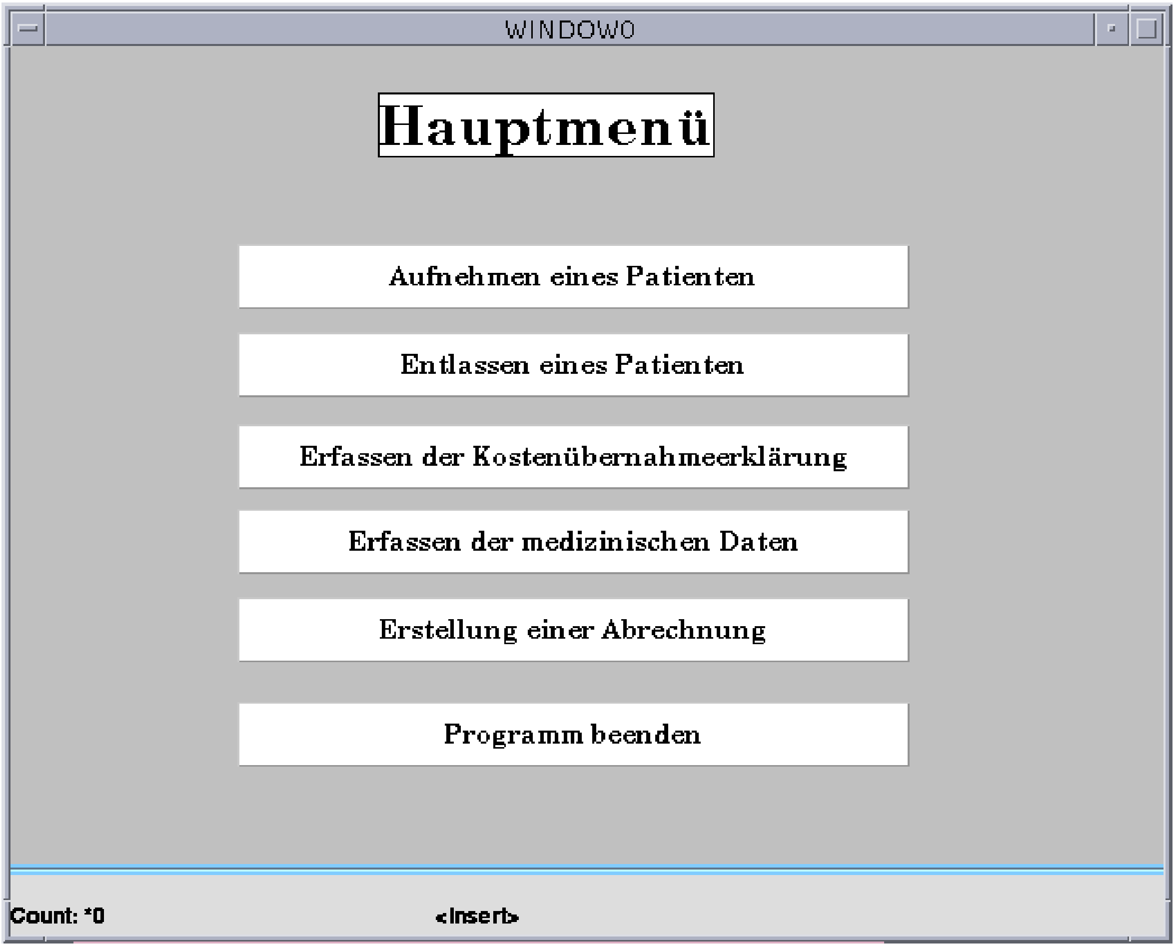


Abbildung 13.1: Das Hauptmenü

Durch Anklicken der einzelnen Buttons erreicht man die entsprechenden Eingabemasken, so wie sie im Unterkapitel „Die Eingabemasken“ (Kapitel 13.3.4) vorgestellt werden. Technisch realisiert wurde dieser Form-Wechsel durch ein PL/SQL-Statement in dem jeweiligen `WHEN-BUTTON-PRESSED-TRIGGER`. Ein solcher befindet sich „hinter“ jedem Button des Menüs. Der Knopf „Programm beenden“ schließt die gesamte Applikation.

Das Untermenü zur Rechnungserstellung

Drückt der Benutzer im Hauptmenü den Button „Erstellung einer Abrechnung“, so erscheint folgendes Untermenü.

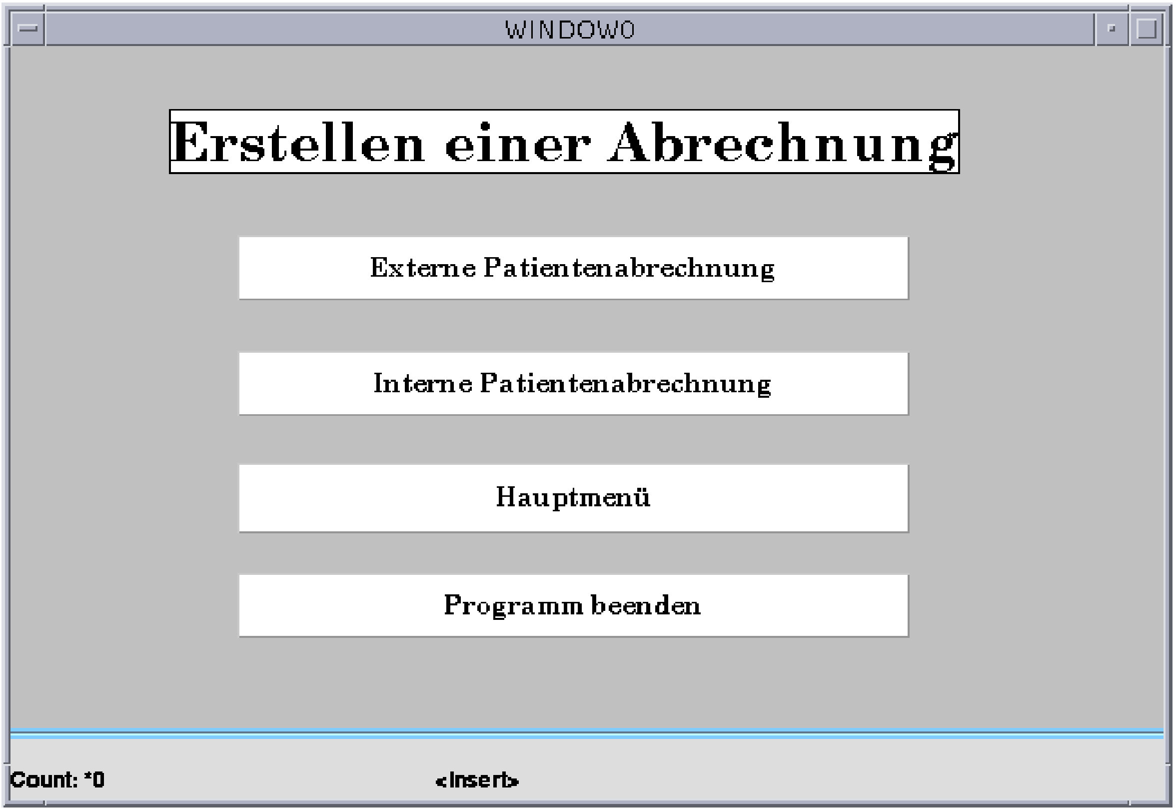


Abbildung 13.2: Das Untermenü zur Erstellung einer Abrechnung

Durch Anklicken der einzelnen Buttons erreicht man die entsprechenden Eingabemasken, so wie sie im Unterkapitel „Die Eingabemasken“ (Kapitel 13.3.4) vorgestellt werden. Technisch realisiert wurde dieser Form-Wechsel durch ein PL/SQL-Statement in dem jeweiligen `WHEN-BUTTON-PRESSED-TRIGGER`.

13.3.4 Die Eingabemasken

Hier soll anhand der Eingabemasken nicht nur das Layout dargestellt werden, sondern auch der Ablauf der Applikation aus technischer Sicht geschildert werden. Dies soll geschehen, indem der Quellcode hinter einigen Items und Buttons, Trigger genannt in der Sprache von Oracle, vorgestellt wird.

Patientensuche

Diese Maske bietet lediglich eine Hilfsfunktion, welche von einigen anderen Funktionen benötigt wird.

Um diese Hilfsfunktion zu erreichen, muß jeweils der Button *Patienten suchen* gedrückt werden. Dieser führt zu einer Übersichtstabelle, welche alle Patienten, geordnet nach Patienten-ID, auflistet. Diese Liste läßt sich nun durch Suchkriterien einschränken. Diese Kriterien können zum Beispiel Teile eines Namens oder der Anschrift sein. Diese Suchmaske zeigt der folgende Screenshot.

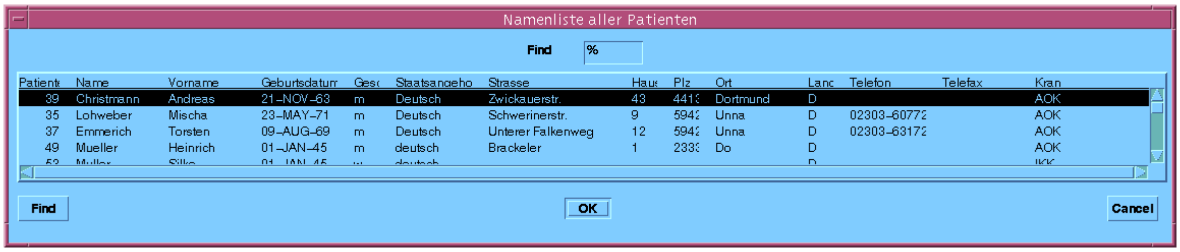


Abbildung 13.3: Die Patientenauflistung in der Suchfunktion

In der angezeigten Liste der Patienten kann der Benutzer nun den gewünschten Patienten mit einem Doppelklick auswählen. Die entsprechende Patienten-ID wird dann automatisch in die Entlassungsmaske übernommen.

Aufnehmen eines Patienten

Die Aufnahme von Patienten wird dem Benutzer mit Hilfe folgender Eingabemaske ermöglicht.

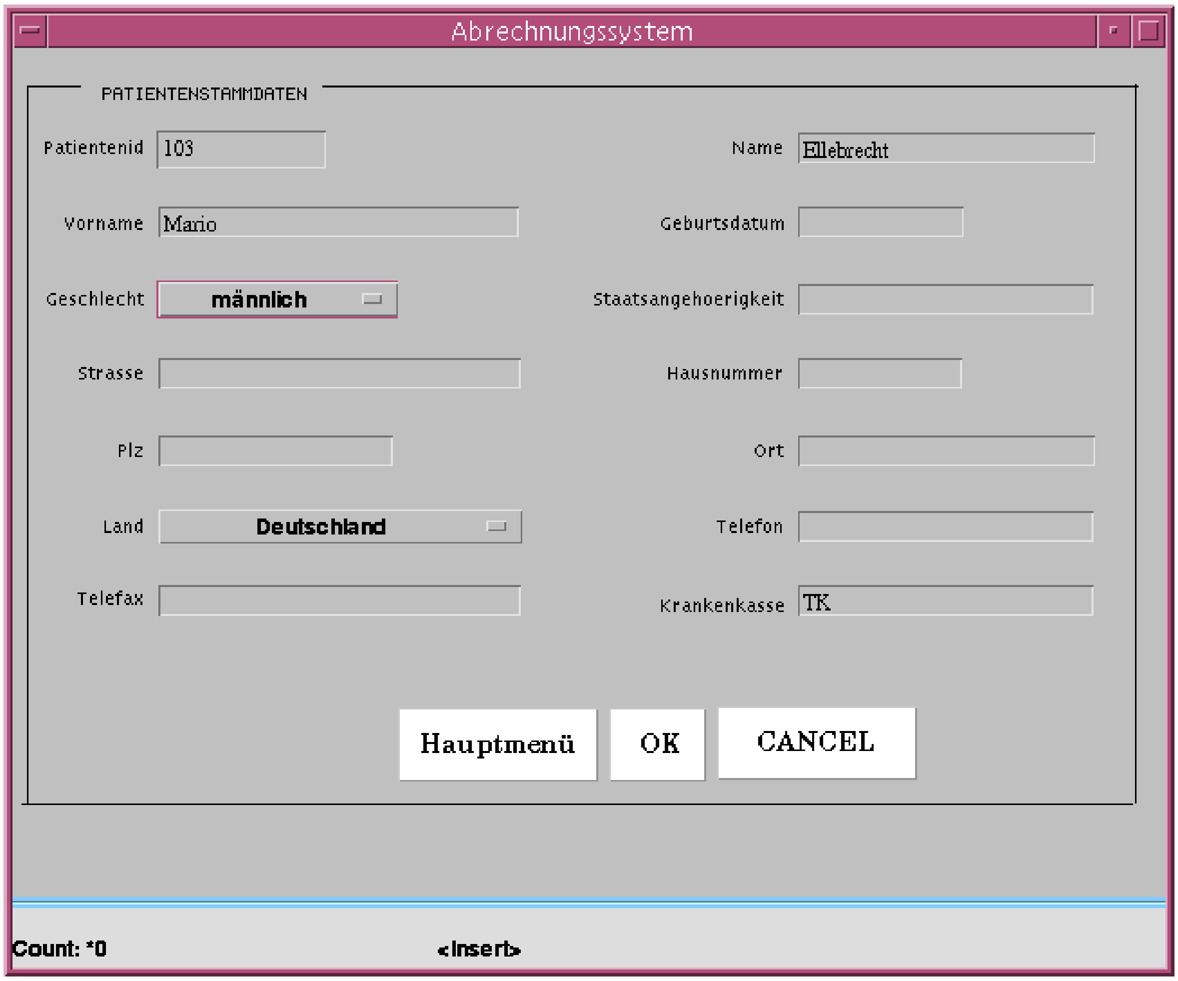


Abbildung 13.4: Die Maske zur Patientenaufnahme

Die vierzehn grafisch dargestellten Items entsprechen jeweils der gleichnamigen Spalte in der Datenbanktabelle „Patientenstammdaten“.

Diese sind wie folgt zu bearbeiten:

- Die *Patientenid* wird vom System automatisch vergeben und verwaltet. Der Benutzer kann diese nicht ändern.
- Die Felder *Geschlecht* und *Land* bieten dem Benutzer eine Auswahlmöglichkeit an. Bei letzterem ist auch eine Ergänzung weiterer Einträge möglich.
- Das *Geburtsdatum* kann in der üblichen zehnstelligen-Form eingegeben werden (TT.MM.JJJJ).
- Die restlichen Felder sind Freitextfelder und nehmen somit beliebigen Text auf.

Am unteren Rand der Form befinden sich die drei Buttons, gemäß der Layoutkonventionen. Die Semantik ist in allen Eingabemasken die gleiche. Sie ist wie folgt:

- Durch Drücken des Buttons *Hauptmenü* wird die aktuelle Form sofort verlassen und in das Hauptmenü gewechselt. Es wird insbesondere an dieser Stelle nichts gespeichert!
- Mit dem *Cancel*-Button werden alle sichtbaren Daten wieder verworfen und die Eingabefelder gelöscht.
- Bei Betätigung des *OK*-Buttons wird zunächst überprüft, ob alle Bedingungen seitens der Datenbank erfüllt sind (Integritätsbedingungen, NOT-NULL-Bedingung, IN-Bedingungen, ...). Ist dies der Fall, so werden die Daten gesichert. Anderenfalls wird eine Fehlermeldung angezeigt.

Entlassen eines Patienten

Die Patientenentlassung geschieht mit folgender Eingabemaske.

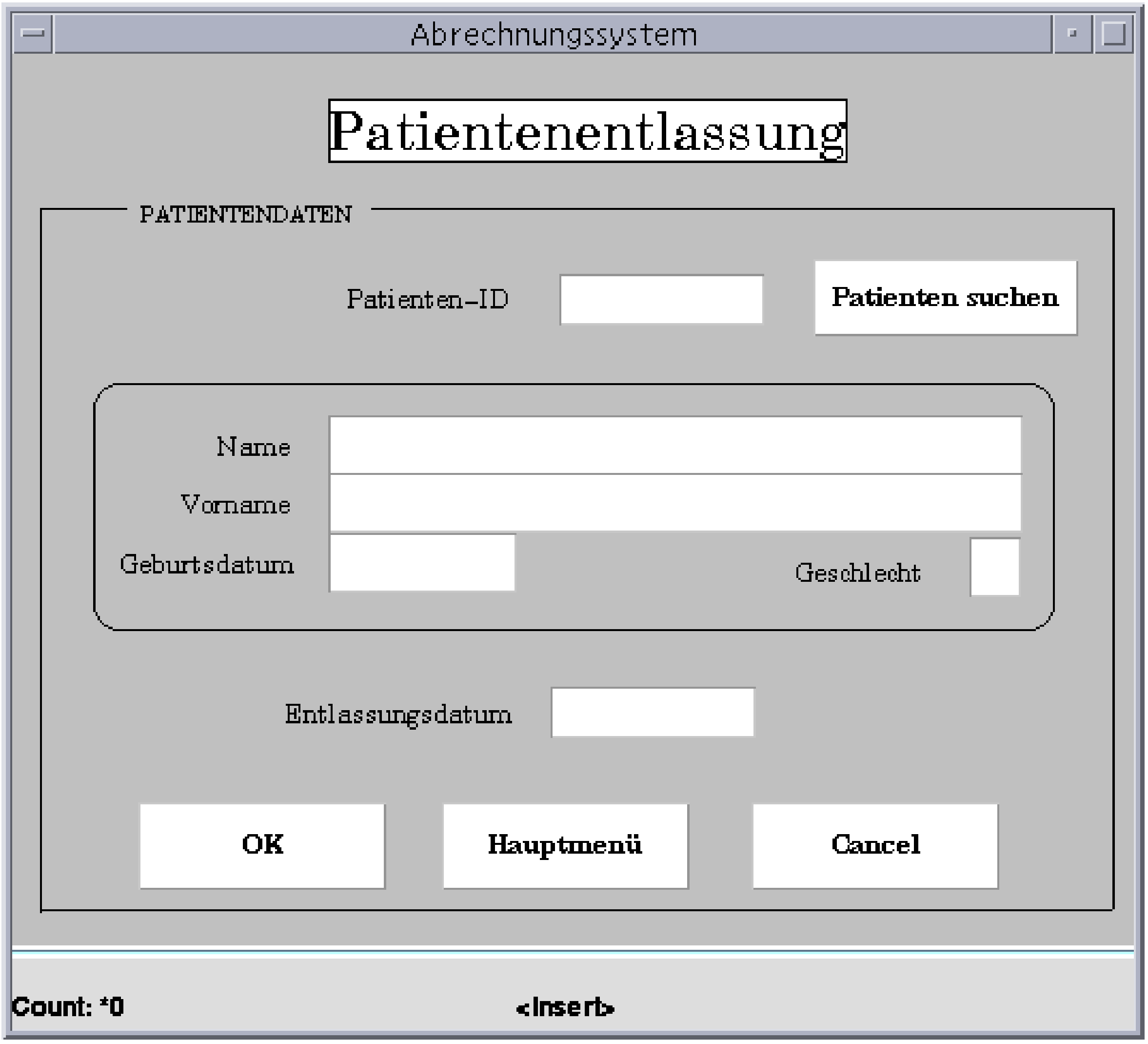


Abbildung 13.5: Die Maske zur Patientenentlassung

Zunächst muß der Patient eindeutig durch seine Patienten-ID bestimmt werden. Dies kann entweder durch direkte Eingabe selbiger im gleichnamigen Item geschehen, oder aber durch eine Suche. Diese wurde im Abschnitt **Patientensuche** beschrieben.

Nach dem Bestätigen mit Return erscheinen in den umrandeten Felder die zur Patienten-ID gehörenden Personalien des Patienten. Dies soll dem Benutzer lediglich als Kontrolle dienen.

Im folgenden Feld kann jetzt das Entlassungsdatum eingegeben werden. Die Form muß folgendem Muster entsprechen: TT.MM.JJ

Am unteren Rand befinden sich wieder die drei Standardbuttons mit der bereits erläuterten Semantik.

Erfassen der Kostenübernahmeerklärung

Die Erfassung der Kostenübernahmeerklärung geschieht mit der folgenden Maske.

Abbildung 13.6: Die Maske für die Kostenübernahmeerklärung

Zunächst muß der Patient eindeutig durch seine Patienten-ID bestimmt werden. Dies kann entweder durch direkte Eingabe selbiger im gleichnamigen Item geschehen, oder aber durch eine Suche. Diese wurde bereits im Abschnitt **Patientensuche** beschrieben.

Nach dem Bestätigen mit Return erscheinen in den umrandeten Felder die zur Patienten-ID gehörenden Personalien des Patienten. Dies soll dem Benutzer lediglich als Kontrolle dienen.

Falls die Krankenkasse die Kosten für die aktuellen Behandlungen des Patienten übernimmt, soll das *OK*-Button gedrückt werden. Die Rechnung geht dann somit an die Krankenkasse des Patienten. Technisch realisiert wird dies durch ein PL/SQL-Statement in dem jeweiligen `WHEN-BUTTON-PRESSED-TRIGGER`. Ein solcher befindet „hinter" jedem Button einer Maske.

Erfassen der medizinischen Daten

Mit Hilfe folgender Eingabemaske kann man die Krankheiten (ICD), Therapie (ICPM), die Materialien, deren Anzahl und Preis erfassen.

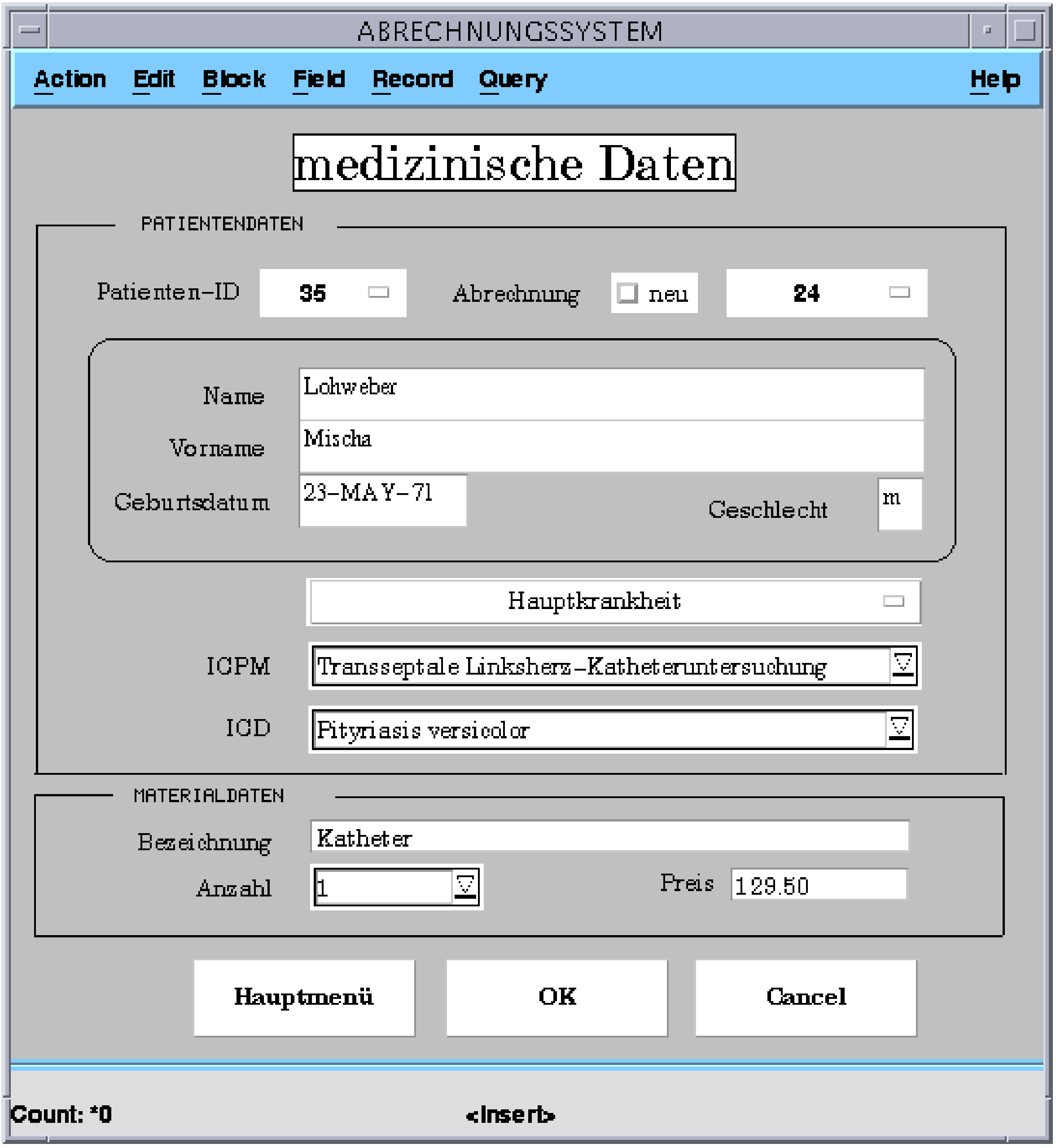


Abbildung 13.7: Die Maske für das Erfassen der medizinischen Daten

Diese Maske ist wie folgt zu bearbeiten.

- Durch Anklicken des *Patienten-ID*-Buttons erscheint eine Liste von allen Patienten-ID's. Dies wurde technisch mit einer LOV (**L**ist **O**f **V**alues) realisiert.

  Die Funktion **L**ist **O**f **V**alues bietet die Möglichkeit, eine Werteauswahl für ein bestimmtes Feld (Item) in Abhängigkeit von einer Spalte einer beliebigen, definierten Tabelle, auf die der Benutzer Zugriffsrechte hat, zu treffen. Der Benutzer kann die Werte sequentiell abrufen und ggf. übernehmen.

- Bei der Aufnahme wird dem Patienten neben der Patienten-ID auch automatisch eine Abrechnungs-ID vergeben.

Bei der ersten Eingabe der medizinischen Daten werden die Daten unter dieser Abrechnungs-ID gespeichert. Dazu muß man in dem entsprechenden Feld die jeweilige Abrechnungs-ID anklicken. Im Falle einer bzw. mehrerer Nebenkrankheiten muß man das Feld *neu* anklicken. Dann wird dem Patienten eine neue Abrechnungs-ID vergeben. Jedoch darf dann nicht schon eine Abrechnungs-ID angeklickt sein. Dies muß man immer dann machen, wenn man eine neue Nebenkrankheit eintragen will.

- Es kann zwischen einer Haupt- und Nebenkrankheit gewählt werden. Ein Patient hat eine Hauptkrankheit und evtl. eine oder mehrere Nebenkrankheiten. Zu jeder Krankheit existiert eine Abrechnungs-ID.
- Das Feld *ICPM* ist die Therapiebezeichnung zu der in *ICD* gewählten Krankheit. Aus einer LOV kann man die jeweilige ICD bzw. ICPM auswählen.
- In die Felder von dem Block `Materialdaten` kann man das Material, deren Anzahl und Preis eintragen.

Externe Patientenabrechnung

Diese Funktion dient dazu, die Behandlungen eines Patienten mit seiner Krankenkasse oder im Falle der Ablehnung der Kostenübernahme mit ihm selbst abzurechnen. Es wird eine druckbare Rechnung erstellt.

Die Auswahl des abzurechnenden Patienten geschieht mit folgender Maske:

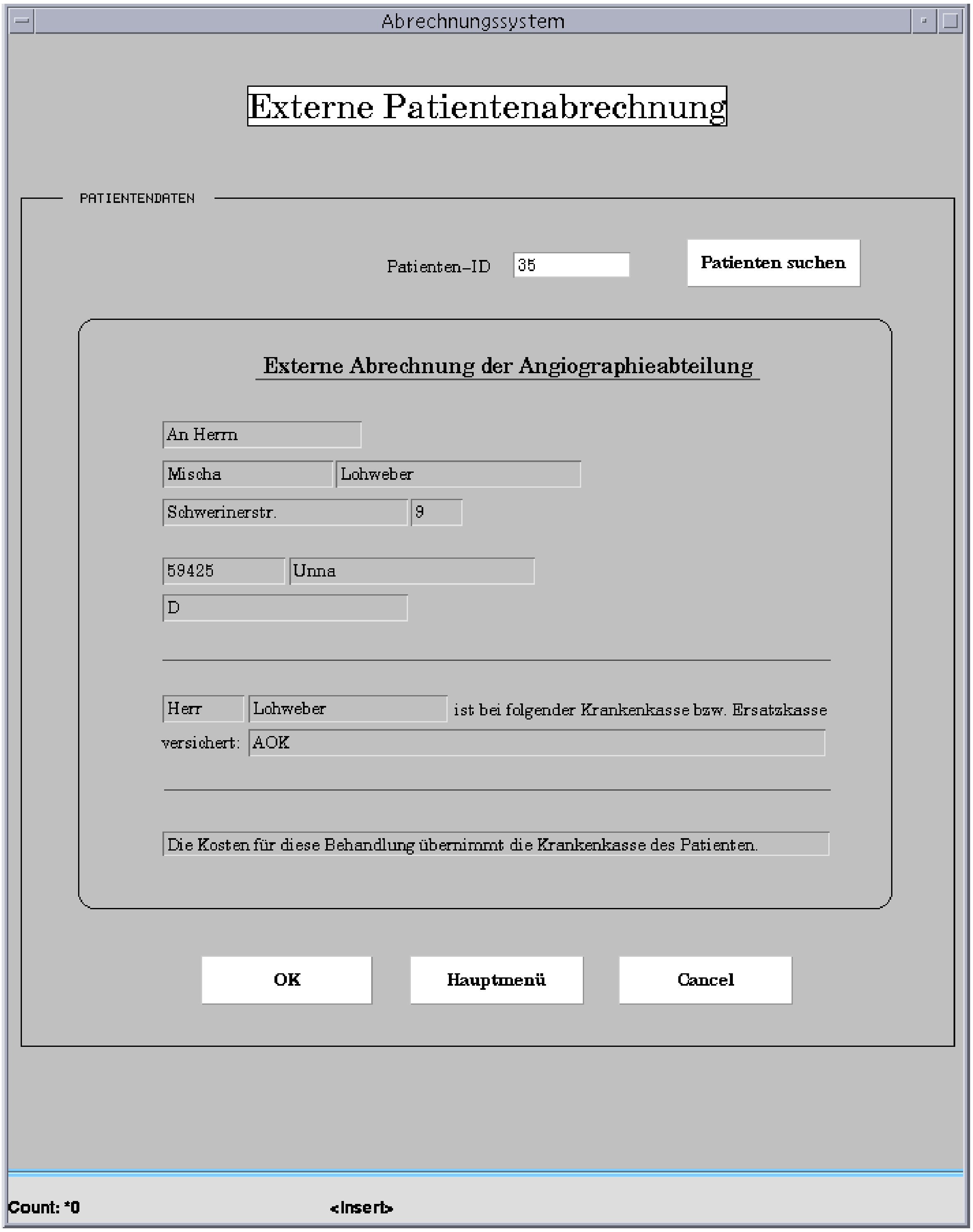


Abbildung 13.8: Die Maske zur externen Patientenabrechnung

Zunächst muß der Patient eindeutig durch seine Patienten-ID bestimmt werden. Dies kann entweder durch direkte Eingabe selbiger im gleichnamigen Item geschehen oder aber durch eine Suche. Um einen Patienten zu suchen, steht die schon erwähnte Patientensuchfunktion zur Verfügung, die durch den Button *Patienten suchen* aufgerufen wird. Diese wurde im Abschnitt **Patientensuche** beschrieben.

Ist die Patienten-ID eingegeben, werden die Stammdaten in der Maske angezeigt, wie oben abgebildet. Ein Betätigen des OK-Buttons ruft die Berechnungsfunktion für die Behandlungen dieses Patienten auf.

Das Resultat der Berechnung ist eine Patientenabrechnung. Im folgenden ist eine solche Rechnung beispielhaft abgedruckt.

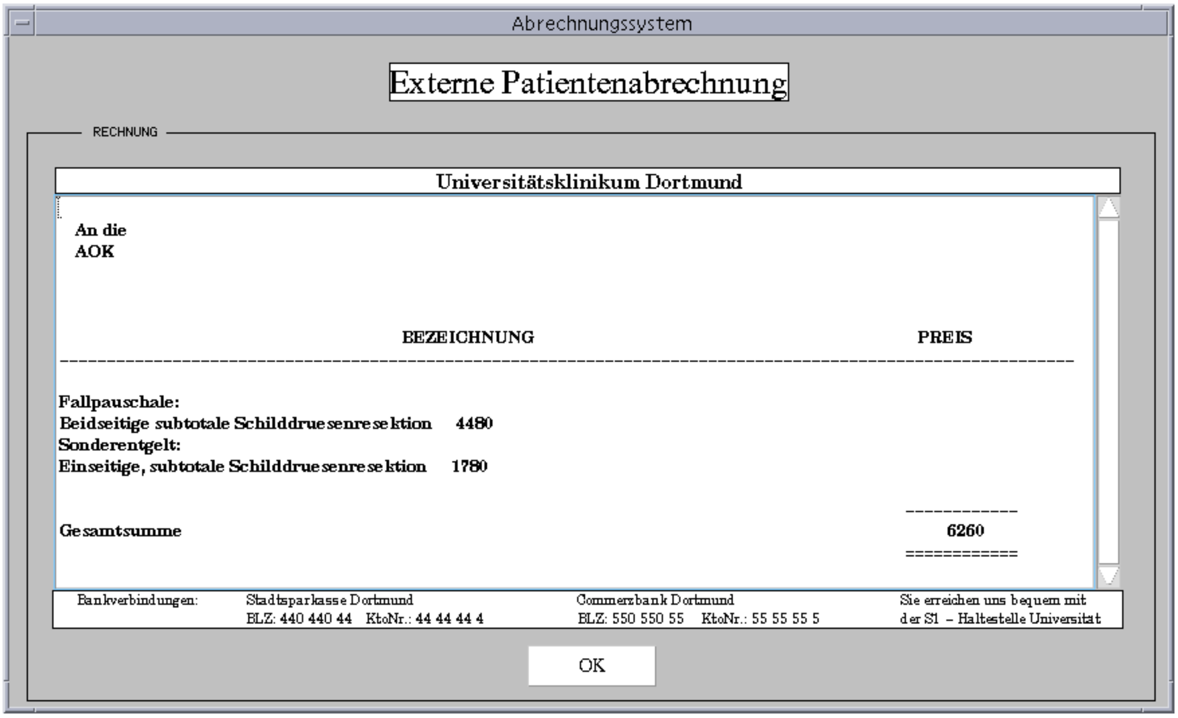


Abbildung 13.9: Eine externe Patientenabrechnung

Zu sehen ist eine Rechnung für die Krankenkasse eines Patienten, der aufgrund einer Haupt- und einer Nebenkrankheit behandelt wurde. Die Hauptkrankheit wurde in Anlehnung an die gesetzlichen Bestimmungen mit einer Fallpauschale, die Nebenkrankheit mit einem Sonderentgelt abgerechnet.

Im folgenden ist der PL/SQL Code abgedruckt, der durch den WHEN-BUTTON-PRESSED-Trigger des OK-Buttons in der o.a. Maske zur externen Patientenabrechnung aktiviert wird und der Erstellung der Rechnung dient. Er gliedert sich grob in folgende Schritte:

- die Abrechnungsdaten des angegebenen Patienten werden selektiert
- der Rechnungskopf wird erstellt
- für jeden Abrechnungsdatensatz wird eine Position in der Rechnung erstellt, und zwar abhängig davon, ob es sich um eine Haupt- oder Nebenkrankheit handelt und ob es zur Abrechnung Fallpauschalen oder Sonderentgelte gibt oder pauschal abgerechnet werden muß
- der Rechnungsfuß wird erstellt
- der Abrechnungsstatus der Abrechnungsdaten des Patienten wird aktualisiert

Bei der Entwicklung des Codes wurden die von Oracle-Forms zur Verfügung gestellten Selektions- und Änderungsfunktionen für die Datenbank nicht benutzt. Stattdessen werden Datensätze mit eigenen SELECT und UPDATE-Statements selektiert und verändert. Dadurch wird das abschließende COMMIT notwendig, das sich am Ende des Programmstücks befindet.

```
DECLARE
        -- deklariere die Abrechnungskonstanten
        basispflegesatz         FLOAT(10,2);
        abteilungspflegesatz    FLOAT(10,2);
        pflegesatz              FLOAT(10,2);
        erm_pflegesatz          FLOAT(10,2);

        -- selektiere alle Abrechnungsdatensaetze passend zur eingegebenen
        -- PatientenId im Cursor "abr_daten"
        CURSOR abr_daten_cursor IS
                SELECT *
                FROM abrechnungsdaten
                WHERE patientenid = :patientendaten.patientenid AND
                      entlassungsdatum IS NOT NULL AND
                      abrechnungsstatus = 'f';

        -- deklariere Datentyp fuer eine Zeile der Tabelle HK_NK_Zuordnung
        -- und der Patientenstammdaten
        lokale_hk_nk_zuordnung  hk_nk_zuordnung\%ROWTYPE;
        lokale_stammdaten       patientenstammdaten\%ROWTYPE;
        lokale_summe            FLOAT;
        lokale_rechnung         VARCHAR2;
        lokale_bezeichnung      fallpauschale.bezeichnung\%TYPE;
        lokaler_preis           fallpauschale.preis\%TYPE;
        lokale_dauer            FLOAT;

        ---------------------------------------------------------------------

        PROCEDURE schreibe_leerzeichen (n INTEGER)
        IS
                zaehler         INTEGER;
        BEGIN
                zaehler := 0;
                LOOP
                        zaehler := zaehler + 1;
                        :text.textausgabe := :text.textausgabe || CHR(1) ;
                        IF zaehler >= n THEN
                                EXIT;
                        END IF;
                END LOOP;
        END;

        ---------------------------------------------------------------------

        PROCEDURE schreibe_kopf
        IS
                lok_kuebern     abrechnungsdaten.kostenuebernahme\%TYPE;
                lok_aid         NUMBER;
        BEGIN
                :text.textausgabe := '';
                SELECT  *
                INTO    lokale_stammdaten
                FROM    patientenstammdaten
                WHERE   patientenid = :patientendaten.patientenid;
```

```
        SELECT  MIN(abrechnungsid)
        INTO    lok_aid
        FROM    abrechnungsdaten
        WHERE   patientenid = :patientendaten.patientenid;

        SELECT  kostenuebernahme
        INTO    lok_kuebern
        FROM    abrechnungsdaten
        WHERE   abrechnungsid = lok_aid;

        :text.textausgabe := :text.textausgabe || CHR(10) ;

        IF lok_kuebern = 't' THEN
                -- im Kopf muss die Krankenkasse eingetragen werden
                :text.textausgabe := :text.textausgabe ||
                '    An die' || CHR(10) || '    ' ||
                lokale_stammdaten.krankenkasse || CHR(10) ||
                CHR(10) || CHR(10) || CHR(10);
        ELSE
                -- im Kopf muss der Patient eingetragen werden
                :text.textausgabe := :text.textausgabe ||
                '    An' || CHR(10) || '    ' ||
                lokale_stammdaten.vorname || ' ' ||
                lokale_stammdaten.name || CHR(10) || '    ' ||
                lokale_stammdaten.strasse || ' ' ||
                lokale_stammdaten.hausnummer || CHR(10) || '    ' ||
                TO_CHAR (lokale_stammdaten.plz) || ' ' ||
                lokale_stammdaten.ort || CHR(10) || '    ' ||
                lokale_stammdaten.land || CHR(10) ||
                CHR(10) || CHR(10) || CHR(10);
        END IF;

        schreibe_leerzeichen (85) ;
        :text.textausgabe := :text.textausgabe || 'BEZEICHNUNG' ;
        schreibe_leerzeichen (95) ;
        :text.textausgabe := :text.textausgabe ||
        'PREIS' || CHR(10) ||
        '------------------------------------------------------' ||
        '------------------------------------------------------' ||
        CHR(10) || CHR(10) ;

END schreibe_kopf;

----------------------------------------------------------------------

PROCEDURE schreibe_fuss
IS
BEGIN
        :text.textausgabe := :text.textausgabe || CHR(10) ;
        schreibe_leerzeichen (210) ;
        :text.textausgabe := :text.textausgabe ||
        '------------' || CHR(10) ||
        'Gesamtsumme' ;
```

```
            schreibe_leerzeichen (190) ;
            :text.textausgabe := :text.textausgabe ||
            TO_CHAR (lokale_summe) || CHR(10) ;
            schreibe_leerzeichen (210) ;
            :text.textausgabe := :text.textausgabe ||
            '===========' ;
    END;

    ---------------------------------------------------------------------


-- hier beginnt das Hauptprogramm
BEGIN
    -- setze die Abrechnungskonstanten
    basispflegesatz := 55.55;
    abteilungspflegesatz := 33.33;
    pflegesatz := basispflegesatz + abteilungspflegesatz;
    erm_pflegesatz := 0.8 * abteilungspflegesatz;
    lokale_summe := 0;

    -- schreibe den Adresskopf auf die Rechnung
    schreibe_kopf;

    -- Schleife die alle in die Rechnung einfliessenden
    -- Abrechnungsdatensaetze durchlaeuft
    FOR abrechnungsdaten_rec IN abr_daten_cursor LOOP

            -- selektiere aktuellen hk_nk-Datensatz in lokale Variable
            SELECT *
            INTO lokale_hk_nk_zuordnung
            FROM hk_nk_zuordnung
            WHERE abrechnungsid = abrechnungsdaten_rec.abrechnungsid;

            -- selektiere aktuelle FallpauschalenDaten in lok. Var.
            SELECT bezeichnung, preis
            INTO lokale_bezeichnung, lokaler_preis
            FROM fallpauschale
            WHERE fallpauschalennummer = lokale_hk_nk_zuordnung.fallpauschalennummer;

            -- berechne die verweildauer bezueglich des akt. Datensatzes
            lokale_dauer := ROUND (MONTHS_BETWEEN(abrechnungsdaten_rec.entlassungsdatum,
                            abrechnungsdaten_rec.aufnahmedatum) * 31);

            -- pruefe, ob es sich um die Hauptkrankheit handelt
            IF lokale_hk_nk_zuordnung.hk_or_nk = 'h' THEN
                    -- pruefe, ob eine Fallpauschale existiert
                    IF lokale_hk_nk_zuordnung.fallpauschalennummer <> '0' THEN
                            :text.textausgabe := :text.textausgabe ||
                            'Fallpauschale: ' || CHR(10) ||
                            lokale_bezeichnung  ||
                            '       '  ||
                            TO_CHAR(lokaler_preis) ||
                            CHR(10);
                            lokale_summe := lokale_summe + lokaler_preis;
```

```
                ELSE    -- es existiert keine Fallpauschale
                        :text.textausgabe := :text.textausgabe ||
                        'Pflegesatzabrechnung Hauptkrankheit:' ||
                        '                                     ' ||
                        TO_CHAR(lokale_dauer * pflegesatz) ||
                        CHR(10);
                        lokale_summe := lokale_summe + lokale_dauer * pflegesatz;
                END IF;

        ELSE    -- es handelt sich um eine Nebenkrankheit
                -- pruefe, ob ein Sonderentgelt existiert
                IF lokale_hk_nk_zuordnung.fallpauschalennummer <> '0' THEN
                        :text.textausgabe := :text.textausgabe ||
                        'Sonderentgelt: ' || CHR(10) ||
                        lokale_bezeichnung ||
                        '       '  ||
                        TO_CHAR(lokaler_preis) ||
                        CHR(10);
                        lokale_summe := lokale_summe + lokaler_preis;
                ELSE    -- es existiert kein Sonderentgelt
                        :text.textausgabe := :text.textausgabe ||
                        'Pflegesatzabrechnung Nebenkrankheit:' ||
                        '                                     ' ||
                        TO_CHAR(lokale_dauer * erm_pflegesatz) ||
                        CHR(10);
                        lokale_summe := lokale_summe + lokale_dauer * erm_pflegesatz;
                END IF;

        END IF;

        -- setze Abrechnungsstatus des aktuellen Satzes auf True
        UPDATE  abrechnungsdaten
        SET     abrechnungsstatus = 't'
        WHERE   abrechnungsid = abrechnungsdaten_rec.abrechnungsid;

    END LOOP;

    schreibe_fuss;

    FORMS_DDL('commit');

    GO_ITEM('TEXT_OK_BUTTON');
END;
```

Interne Patientenabrechnung

Durch den Aufruf der Funktion **Interne Patientenabrechnung** wird eine Übersicht über die für den ausgewählten Patienten verbrauchten Materialkosten erstellt.

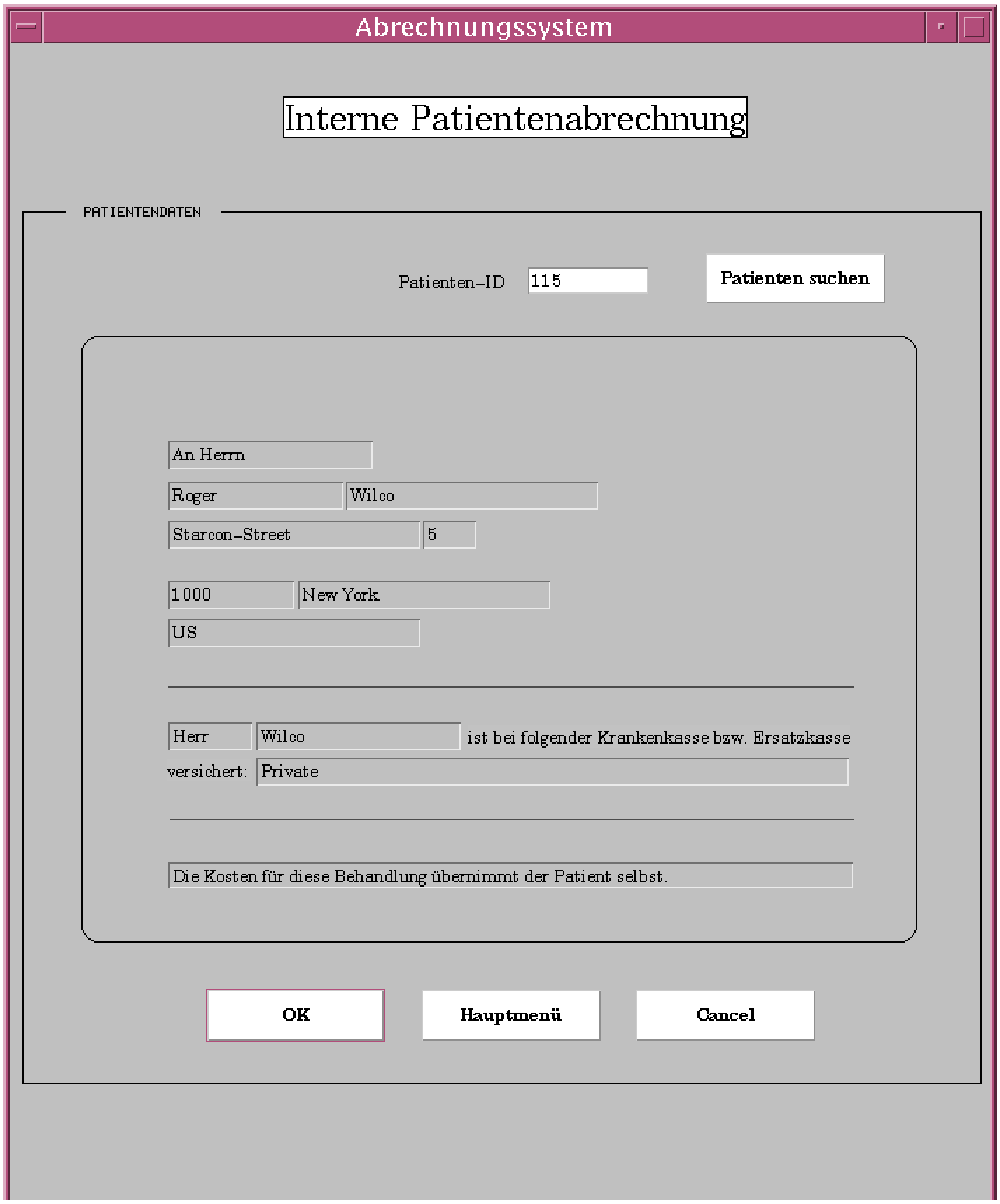


Abbildung 13.10: Die Maske zur internen Patientenabrechnung

Die Patientenid des gewünschten Patienten wird in das Feld Patientenid eingetragen. Nach Bestätigung mit der RETURN-Taste werden die Stammdaten über den Patienten automatisch angezeigt. **Bemerkung:** Ist die Patientenid nicht bekannt, so kann man auch nach einem bestimmten Patienten suchen. Die Suchfunktion eines Patienten wurde im Abschnitt **Patientensuche** beschrieben.

Wenn man den OK-Button drückt, so erscheint folgender Bildschirm. An dieser Stelle werden Informationen darüber angezeigt, welche Materialkosten in welcher Höhe für den Patienten bereits aufgebraucht sind.

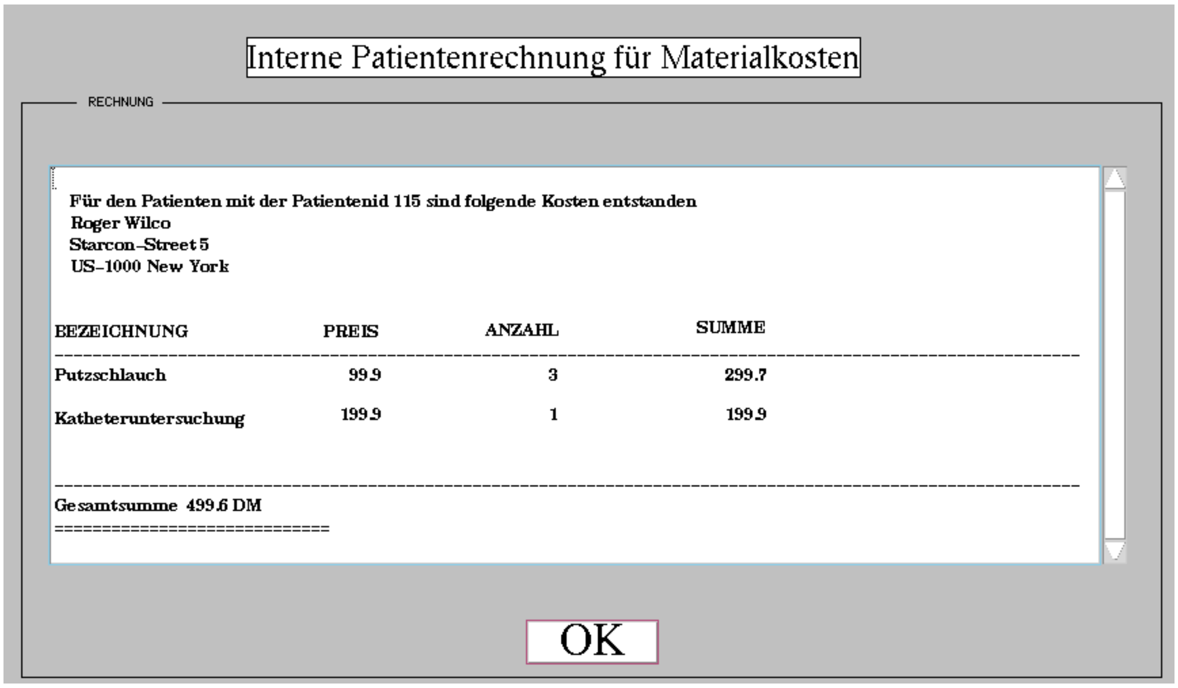


Abbildung 13.11: Die Maske zur Anzeige der Materialkosten eines Patienten

13.3.5 Ein beispielhaftes Szenario

An dieser Stelle soll ein typischer Krankenhausaufenthalt eines Patienten aus Sicht des Systems beschrieben werden. Anhand eines solchen Szeanrios läßt sich die Applikation auch gut im Ganzen testen.

Die einzelnen Schritte sind nun wie folgt:

1. Der Patient wird aufgenommen mit der entsprechenden Maske. Seine Stammdaten werden dann in der Datenbank gespeichert.

2. Für den Patienten werden medizinische Daten erfaßt. Insbesondere erhält er eine Hauptkrankheit und eventuell auch eine oder mehrere Nebenkrankheiten. Verbrauchte Materialien werden an dieser Stelle auch gleich mit erfaßt.

3. Sollte die Krankenkasse dem Krankenhaus die Kostenübernahme für diesen Patienten bestätigen, so wird dies in der gleichnamigen Eingabemaske erfaßt.

4. Der Patient wird mit der Entlassungsmaske entlassen.

5. Ein Controller möchte die für einen speziellen Patienten entstandenen Materialkosten ermitteln. Dazu ruft er den Unterpunkt *Interne Patientenabrechnung* auf. Diese Funktion kann zu jedem beliebigen Zeitpunkt aufgerufen werden.

6. Nun soll die Abrechnung für einen speziellen Patienten erstellt werden. Dazu ist der Unterpunkt *Externe Patientenabrechnung* aufzurufen. Ist dieses einmal geschehen, so werden die abgerechneten Krankheitsfälle als abgerechnet markiert.

Kapitel 14

Der Test

Verfasser: Dilber Yavuz, Djamel Kheldoun

Im Hauptmenü gibt es folgende Menüpunkte:

- `Aufnehmen eines Patienten`
- `Entlassen eines Patienten`
- `Erfassen der Kostenübernahmeerklärung`
- `Erfassen der medizinischen Daten`
- `Erstellung einer Abrechnung`
- `Programm beenden`

1. Patienten aufnehmen
   Der Patient muß aufgenommen werden, bevor die anderen Menüpunkte durchgeführt werden können. Nach Anklicken des Menüpunktes `Aufnehmen eines Patienten` erscheint ein Fenster, in das folgende Daten des Patienten eingegeben werden müssen, sonst erscheint eine Fehlermeldung.

   (a) Name
   (b) Vorname
   (c) Geburtsdatum
   (d) Krankenkasse

   Die anderen Felder können frei gelassen werden. Dann sollen folgende Schritte durchgeführt werden:

   (a) `OK` und dann
   (b) `Hauptmenü` anklicken

   Es erscheint ein Dialog, ob die Änderungen gespeichert werden sollen. Bei diesem Dialog muß immer `No` angeklickt werden, damit überhaupt zum Hauptmenü zurückgekehrt werden kann. Das ist ein Fehler, der nicht behoben wurde.

2. Eingabe von medizinischen Daten
   Man wählt das Patienten-ID aus der Liste der IDs. Das Problem hierbei ist, daß wenn die Liste zu groß wird, die IDs nicht gefunden werden können. Dieses Problem wurde nicht behoben.
   Nachdem ein ID ausgewählt werden konnte, muß ausgewählt werden, ob eine neue Abrechnung

erstellt werden soll oder eine alte Abrechnung benutzt wird.
Man muß für ICPM die Art der Untersuchung auswählen, sonst werden die anderen Angaben nicht akzeptiert.
Bei Angabe der Materialdaten müssen Daten für die Felder Bezeichnung, Anzahl und Preis angegeben werden, wobei man bei der Preisangabe kein Komma, sondern ein Punkt benutzen darf.

3. Patienten entlassen

   (a) im Hauptmenü den Menüpunkt `Entlassen eines Patienten` auswählen
   (b) Patienten-ID eingeben oder `Patienten suchen` anklicken und den gesuchten Namen eingeben
   (c) Entlassungsdatum eingeben und `OK` anklicken, dann zurück zum Hauptmenü

   Das Problem hierbei ist, daß wenn in das Feld für das Patienten-ID angeklickt wurde, auch eine ID angegeben werden muß.

4. Erstellen einer Abrechnung
   Man kann eine externe und eine interne Patientenabrechnung erstellen. Zur Erstellung der externen Patientenabrechnung müssen folgende Schritte durchgeführt werden:

   (a) Patienten-ID eingeben oder `Patienten suchen` anklicken und den gesuchten Namen eingeben
   (b) `Patienten-ID` anklicken und Return-Taste drücken.
   (c) `OK` anklicken.

   Dann wird eine Abrechnung erstellt. Wenn es keine Daten gibt, die abgerechnet werden müssen, dann wird eine leere Abrechnung erstellt.

   Zur Erstellung der internen Patientenabrechnung müssen folgende Schritte durchgeführt werden:

   (a) Patienten-ID eingeben oder `Patienten suchen` anklicken und den gesuchten Namen eingeben
   (b) `Patienten-ID` anklicken und Return-Taste drücken
   (c) `OK` anklicken.

   Bei beiden Abrechnungsformen gibt es das gleiche Problem wie beim Patienten entlassen: wenn in das Feld für das Patienten-ID angeklickt wurde, muß auch eine ID angegeben werden.

Teil VI

Das zweite Semester

Kapitel 15

Die Planung für das 2. Semester

Verfasser: Patrick Koehne

Zum Start in die zweite Etappe der Projektarbeit wurde nur ein Pert-Chart entworfen. Dieses wurde jedoch insofern erweitert, als daß die jeweils an dem Prozeß beteiligten Personen mit in das Chart aufgenommen wurden. Somit erhält man eine Kombination aus Pert- und Gantt-Chart ohne, meiner Ansicht nach, die Lesbarkeit großartig zu verschlechtern.

15.1 Der Projektverlauf

Aus der Abbildung 15.1 ist zunächst zu entnehmen, daß sich das Projekt stark in vier Teilprozesse aufteilt. Diese sind weitgehend unabhängig und somit verläuft die Implementierung auch wirklich parallel. Noch mehr als im ersten Semester werden also die allgemeinen Probleme der parallelen Entwicklung auf uns zukommen. Dazu zählen vermutlich vorwiegend mangelnde Absprachen unter den einzelnen Arbeitsgruppen.

Die Fertigstellung des Abrechnungssystems wird zunächst auf Ende April festgelegt. Danach soll der Datenbankadapter, welcher unabhängig implementiert wird, mit dem System zusammengeführt werden. Dieses Vorhaben ist, weil es die erste Integration in dem System darstellt, auch mit knapp zwei Arbeitswochen angesetzt. Das O2-Adapter-System aus der Einarbeitungsaufgabe soll im dritten genannten Prozeß als Grundlage dienen, einen dem Gesamtsystem entsprechenden Adapter für das Angiographiesystem zu erstellen. Der Schwerpunkt der Arbeit, die aller Vorraussicht nach auch am zeitintensivsten sein wird, gilt der Erstellung des eigentlichen Förderierungssystems. Es unterteilt sich in den eigentlichen Graphen, der später in einer O2-Datenbank gehalten werden soll, der ComI, welche zum grösten Teil aus der Einarbeitungsaufgabe entnommen werden soll, und den Algorithmen, die auf dem Graphen arbeiten und diesen pflegen.

Der Start der Integration all dieses Teilprozesse soll spätestens zum 1. Juni beginnen, da für den 14. Juni eine Präsentation der Projektarbeit im Rahmen des Tages der offenen Tür der Universität angesetzt worden ist. Die sehr optimistisch angelegte Planung ist an dieser Stelle auch nicht eingehalten worden. Die Integration konnte erst zum 9. Juni beginnen.

Da der eigentliche Schwerpunkt der Projektarbeit bis zu dem Zeitpunkt der Präsentation am 14. Juni eigentlich erledigt sein sollte, kommen nun einige organisatorische und dokumentarische Arbeiten auf uns zu. Für die Präsentation müssen schließlich Schaubilder für Stellwände entworfen und Szenarien, die man demonstrieren möchte, ausgedacht werden. Nach dieser Präsentation wird sich ein Teil der Projektgruppe noch um die Weiterentwicklung des Programms kümmern. Darunter fällt im wesentlichen die Portierung des Förderierungsgraphen von der C++ -Struktur in die O_2-Datenbank. Die restlichen Teilnehmer werden sich um die Vervollständigung des Endberichtes kümmern. Dies wird noch einige Zeit in Anspruch nehmen, da während der Implementierungsphase die dokumentarische

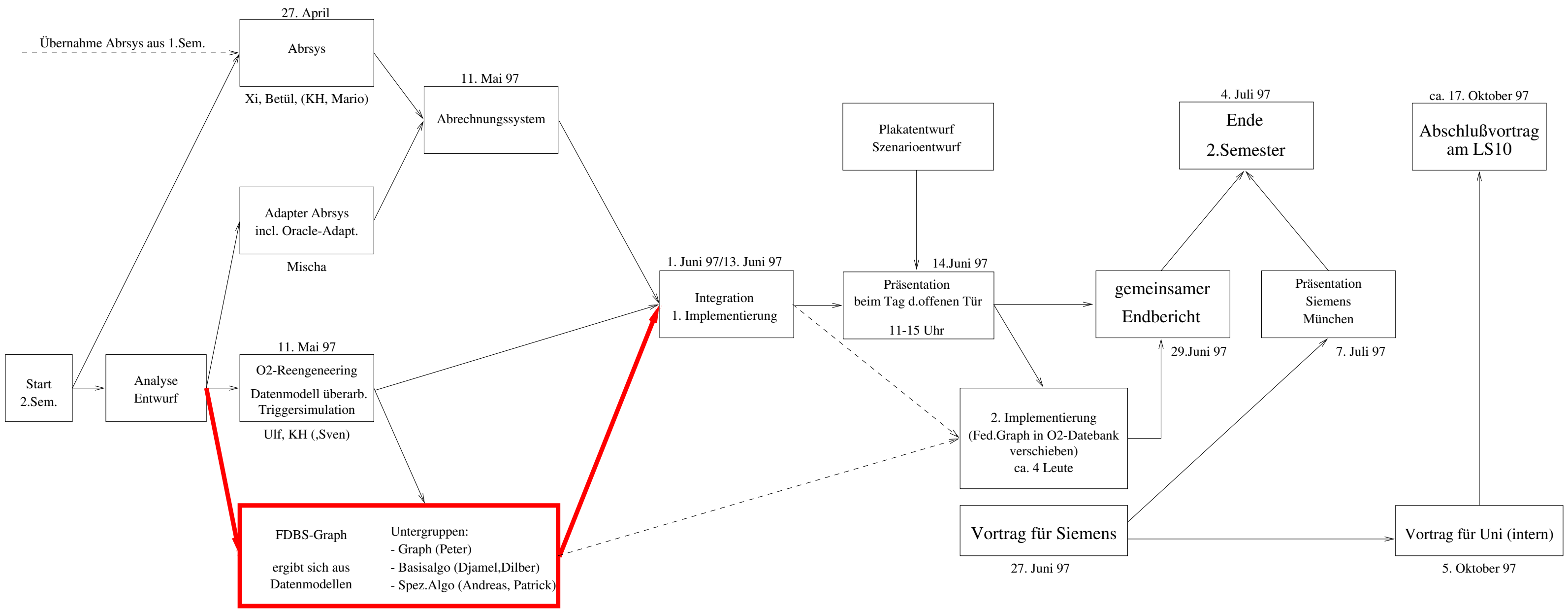


Abbildung 15.1: Das Pert–Chart für das 2.Semester des Projektes

Arbeit ein wenig in den Hintergund geraten ist. Da für den 7. Juli eine weitere Präsentation der Ergebnisse, diesmal bei Siemens in München, vorgesehen ist, müssen auch hierfür Vortragsunterlagen entworfen werden. Aus diesen sollten sich auch die Unterlagen für die eigentlich wichtige, uniinterne Abschlußpräsentation der Projektergebnisse ergeben. Zum momentanen Zeitpunkt gehen wir davon aus, daß in der vorlesungsfreien Zeit im Sommer nur einige wenige Gruppensitzungen notwendig sind, um den Abschlußbericht zu korrigieren.

Kapitel 16

Reengineering des Angiographiesystems

Verfasser: Sven Gerding, Ulf Radmacher

16.1 Anforderungen an das Angiographiesystem

Das Ziel der Projektgruppe, das Abrechnungssystem und das Angiographiesystem zu föderieren, erfordert es, sich Gedanken über die Aufgabenverteilung innerhalb dieses föderierten Systems zu machen. Um die Daten im föderierten System konsistent zu halten, muß ein Einfügen, Ändern oder Löschen von Datensätzen in einer der Datenbanken, dem FDBS und, als Folgereaktion, der anderen Datenbank bekannt gemacht werden. Es soll also möglich sein, einen Datensatz unabhängig von der eigentlichen Applikation in das Angiographiesystem einzutragen, zu verändern oder zu löschen. Diese Aufgabe ist dem O_2-Adapter zugedacht. Andereseits müssen aber auch Änderungen des föderierten Datenbestandes, die beim Einsatz des Angiographiesystem eintreten, dem FDBS gemeldet werden. Hierbei ist es sowohl notwendig, die Art der Änderung bekanntzugeben, als auch die Objekte, die sich geändert haben. Dies macht es notwendig, daß föderierte Objekte eine eindeutige Objektidentität besitzen, die auch außerhalb des Angiographiesystem bekannt sein muß.

16.2 Analyse des Angiographiesystems

Die Analyse des Angiographiesystems hat ergeben, daß es notwendig ist, die Autonomie der Applikation einzuschränken. So ist O_2 das Konzept der Trigger, wie man es von Oracle her kennt, nicht bekannt. Daraus folgt, daß ein Eingriff in den Quellcode der Applikation nicht umgangen werden kann. Änderungen des Datenbestandes müssen dem FDBS aktiv bekanntgemacht werden. Als weiteres Problem hat sich die Handhabung der Objektidentität erwiesen. O_2 vergibt diese zwar eindeutig, verbirgt sie jedoch vor dem Benutzer. Da keine der föderierten Klassen über Schlüsselattribute verfügt, ist es notwendig, ein solches hinzuzufügen. Des weiteren existiert in O_2 keine Möglichkeit sich Schlüsselwerte automatisch generieren zu lassen, so daß man für die Verwaltung solcher Schlüsselwerte selbst verantwortlich ist.

16.3 Reengeniering des Angiographiesystems

Um das Problem der Trigger im Angiographiesystem zu lösen, haben wir bei den zu föderierenden Klassen (`CDraht, CMedikament, CKatheter`) die Methodesammlung um die Methode `export_object` erweitert. Diese wird mit dem Parameter `BOID` vom Typ Integer von derjenigen Therapie aufgerufen, in welche das Material (Katheter, Draht oder Medikament) eingefügt worden ist. Der Parameter `BOID` repräsentiert in dieser ersten Version die Objektidentität der Behandlung, in der eingefügt worden ist. Es soll später die Objektidentität der Therapie übermittelt werden. Da die Applikation nicht vom Konstrukt `inverse` gebraucht macht, mit dessen Hilfe man Beziehungen in beide Richtungen verfolgen kann, war es notwendig, die Klasse `CTherapie` um das Attribut `BOID` zu erweitern, das bei der Initialisierung der Klasse gesetzt wird, und die Objektidentität der aufrufenden Behandlung repräsentiert.

Um die Objektidentitäten der Klassen zu verwalten, die für die Föderation bedeutend sind, war es notwendig, jede dieser Klassen um das Attribut `OID` vom Typ Integer zu erweitern. Im einzelnen sind dies diejenigen Klassen, welche von `CMaterial` und `CPerson` erben, sowie die Klassen `CBehandlung` und `CTherapie`. Damit diese Attribute in der Darstellung durch O_2*Look* nicht im Gesamtbild der Applikation erscheinen, wurden sie als `private` deklariert. Hieraus haben sich Probleme ergeben, wenn dieses Attribut vererbt werden sollte (z.B. von der Klasse `CPerson` an die Klasse `CPatient`), da sich private Attribute nicht vererben lassen. Um nicht eine neue Oberfläche unter Verwendung von Masken für die Applikation zu erstellen, haben wir in den Fällen, in denen mit Verebung gearbeitet werden sollte, das Attribut `OID` als `read` deklariert, wodurch es zwar in der Applikation angezeigt, aber nicht verändert werden kann, und sich vererben läßt. Um das private Attribut `OID` vom Adapter lesen zu können, sind in den entsprechenden Klassen die Methode `export_OID` eingeführt worden, welche ohne Parameter aufgerufen wird und als Ergebnis die Objektidentität als Integer liefert.

Das Attribut `OID` der oben aufgezählten Klassen wird bei der Initialisierung durch `init` gesetzt und bleibt danach unverändert. Die Eindeutigkeit der Objektidentität wird durch den sogenannten Objecthandler gewährleistet. Dieser wird aus der `init`- Methode der zu initialisierenden Klasse über einen für jede Klasse spezifischen Methodenaufruf ohne Parameter aktiviert und liefert als Resultat die eindeutige Objektidentität in Form eines `Integer` (z.B. aus der Klasse `CDraht` durch den Aufruf `OID=OIDHandler->LiefereCDrahtOID`). Der Objecthandler `OIDHandler` ist als `named_Object` im Schema `angio` implementiert und muß über den Programmaufruf `Initialisieren` aus dem Hauptmenu angelegt werden, sobald eine neue `Base` gesetzt worden ist. Die Klasse `COIDHandler` besteht aus den Methoden `LiefereKlassennameOID` und den Attributen `AnzahlKlassenname` vom Typ `Integer`, sowie einen allgemeinen Zähler über die bereits vergebenen Objektidentitäten `AnzahlOID`. Bis jetzt wird dieser Zähler bei jedem Methodenaufruf als Ergebnis geliefert und anschließend um eins erhöht. Man könnte jedoch in späteren Prototypen das Ergebnis der Methoden zu einem `STRING` abändern, der die Objektidentität aus der Erzeugungshierarchie abbildet. So wäre die Objektidentität für einen Patienten beispielsweise `P01`, die zu dem Patienten gehörigen Behandlungen würden die `OIDs P01B01, P01B02, ....` erhalten. Hierbei müssten dann die entsprechenden Attribute hochgezählt werden.

Damit die Applikation mit dem realisierten `C++`-Adapter lauffähig ist, mußten weitere Änderungen am Quellcode vorgenommen werden. So ist beispielsweise das Einbinden der Grafik aus der `init`-Methode der Klasse `CArterien` entfernt worden, da dies sonst eine grafische Oberfläche für den Adapter erfordert hätte. Für die Applikation hat dies die Folge, daß schwarze Kästchen für die nicht initialisierten Objekte vom Typ `CArterie` erzeugt werden.

Im Rahmen der Testszenarios hat es sich ergeben, daß das Programm `Patient suchen` fehlerhaft implementiert worden ist, was durch eine einfache Änderung behoben werden konnte (Es wurde nie ein Patient gefunden, da auf der falschen Liste gesucht worden ist).

Gemäß den Ablaufanforderungen des `FDBS`, daß in der ersten `Insert`-Operation ein Patient in das Angiographiesystem aufgenommen werden soll und anschließend in einer zweiten `Insert`-Operation die zugehörige (leere) Behandlung, war es nötig, die `init`-Methode der Klasse `CPatient` zu ändern. Hier wird in der ursprünglichen Version eine leere Behandlung automatisch beim Erzeugen eines Patienten generiert, was nun nicht mehr geschehen darf.

```
class COIDHandler inherit Object public type
    tuple(AnzahlKatheter: integer,
          AnzahlDraehte: integer,
```

```
          AnzahlBehandlungen: integer,
          AnzahlTherapien: integer,
          AnzahlMedikamente: integer,
          AnzahlPatienten: integer,
          AnzahlObjekte: integer)
method
    public LiefereCMedikamentOID: integer,
    public LiefereCPatientOID: integer,
    public LiefereCTherapieOID: integer,
    public LiefereCBehandlungOID: integer,
    public LiefereCKatheterOID: integer,
    public LiefereCDrahtOID: integer,
    public init
end;
export schema class COIDHandler;


class CPatient inherit CPerson private type
    tuple(public Telefon: string,
          public Geschlecht: boolean,
          public Behandlungen: list(CBehandlung),
          public Kontrollen: list(CAmbulanteKontrolle))
method
    public menu: list(string),
    public title: string,
    public init,
    public Bearbeiten,
    public neue_Kontrolle,
    public neue_Behandlung,
    public Neu,
    public Aufnahme_drucken,
    private export_OID: integer
end;


class CDraht inherit CMaterial private type
    tuple(public K: string,
          public Fr: string,
          public Durchmesser: real,
          public Laenge: string,
          OID: integer)
method
    public Bearbeiten,
    public menu: list(string),
    public title: string,
    public init,
    public export_object(BOID: integer)
end;


class CKatheter inherit CMaterial private type
    tuple(public K: string,
          public Fr: string,
          public Durchmesser: real,
          public Laenge: string,
          OID: integer)
method
    public Bearbeiten,
    public menu: list(string),
```

```
    public title: string,
    public init,
    public export_object(BOID: integer)
end;


class CMedikament inherit CMaterial private type
    tuple(public Name: string,
          OID: integer)
method
    public Bearbeiten,
    public menu: list(string),
    public title: string,
    public init,
    public export_object(BOID: integer)
end;


class CTherapie inherit CMedAktion private type
    tuple(public Mitwirkende: tuple(erster_Untersucher: CArzt,
                                    zweiter_Untersucher: CArzt,
                                    erster_steriler_Ass: CArzt,
                                    zweiter_steriler_Ass: CArzt,
                                    unsteriler_Ass: CArzt),
          public Zeiten: tuple(Untersuchungsbeginn: string,
                               Untersuchungsdauer: string,
                               Durchleutung: tuple(Dauer: string,
                                                   Dosis: string)),
          public Kontrastmittel: tuple(Name: string,
                                       Menge: string),
          public Punktionsstelle: CPunktion,
          public Medikamente: set(CMedikament),
          public Massnahmen: set(CMassnahme),
          public Material: tuple(Katheter: set(CKatheter),
                                 Draehte: set(CDraht),
                                 Verbrauch: set(CVerbrauch)),
          public Komplikationen: set(CKomplikation),
          public Befunde: set(CBefund),
          OID: integer,
          BOID: integer)
method
    public erster_Untersucher,
    public zweiter_Untersucher,
    public erster_steriler_Ass,
    public zweiter_steriler_Ass,
    public unsteriler_Ass,
    public neues_Medikament,
    public neue_Massnahme,
    public neue_Komplikation,
    public neuer_Befund,
    public neuer_Verbrauch,
    public neuer_Draht,
    public neuer_Katheter,
    public title: string,
    public init(hBOID: integer),
    public Bearbeiten,
    public menu: list(string)
end;
```

```
class o2_list_CPatient inherit Object public type
    list(CPatient)
end;


class o2_set_CPatient inherit Object public type
    unique set(CPatient)
end;


name OIDHandler :COIDHandler;
export schema name OIDHandler;


method body init in class CBehandlung
{
        o2 CTherapie t;
        self->OID = OIDHandler->LiefereCBehandlungOID;
        t=new CTherapie(self->OID);

        self->Therapien += list(t);
        self->Aufnahme = new CAufnahme;
        self->Voruntersuchung.Farbduplex.Datum = new Date(0,0,0);
        self->Voruntersuchung.Farbduplex.Arterien  = new CArterien;
        self->Voruntersuchung.Doppler = new CDoppler;
        self->Voruntersuchung.Gehstrecke = new CGehstrecke;
        self->Nachuntersuchung.Farbduplex.Datum = new Date(0,0,0);
        self->Nachuntersuchung.Farbduplex.Arterien  = new CArterien;
        self->Nachuntersuchung.Doppler = new CDoppler;
        self->Nachuntersuchung.Gehstrecke = new CGehstrecke;
};


method body neue_Therapie in class CBehandlung
{
        o2 CTherapie t = new CTherapie(self->OID);
        transaction;
        self->Therapien += list(t);
        self->refresh_all;
};


method body LiefereCMedikamentOID in class COIDHandler
{
    o2 integer flag = false;
    o2 integer anzahl;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;
        }

        anzahl = self->AnzahlObjekte;
    self->AnzahlObjekte = anzahl + 1;
        return anzahl;


        if(o2_in_transaction() && flag)
                        validate;
```

```
};


method body LiefereCPatientOID in class COIDHandler
{
        o2 integer flag = false;
    o2 integer anzahl;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;    }
    anzahl = self->AnzahlObjekte;
    self->AnzahlObjekte = anzahl + 1;
        return anzahl;
        if(o2_in_transaction() && flag)
                        validate;
    self->refresh_all;
};


method body LiefereCTherapieOID in class COIDHandler
{
    o2 integer flag = false;
    o2 integer anzahl;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;    }
    anzahl = self->AnzahlObjekte;
    self->AnzahlObjekte = anzahl + 1;
        return anzahl;
        if(o2_in_transaction() && flag)
                        validate;
};


method body LiefereCBehandlungOID in class COIDHandler
{   o2 integer flag = false;
    o2 integer anzahl;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;    }
    anzahl = self->AnzahlObjekte;
    self->AnzahlObjekte = anzahl + 1;
        return anzahl;
        if(o2_in_transaction() && flag)
                        validate;
};


method body LiefereCKatheterOID in class COIDHandler
{
    o2 integer flag = false;
    o2 integer anzahl;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;}
    anzahl = self->AnzahlObjekte;
    self->AnzahlObjekte = anzahl + 1;
        return anzahl;
        if(o2_in_transaction() && flag)
```

```
                        validate;
};


method body LiefereCDrahtOID in class COIDHandler
{
        o2 integer flag = false;
    o2 integer anzahl;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;    }
    anzahl = self->AnzahlObjekte;
    self->AnzahlObjekte = anzahl + 1;
        return anzahl;
        if(o2_in_transaction() && flag)
                        validate;
};


method body init in class COIDHandler
{
        self->AnzahlKatheter = 0;
        self->AnzahlDraehte = 0;
        self->AnzahlBehandlungen = 0;
        self->AnzahlTherapien = 0;
        self->AnzahlMedikamente = 0;
        self->AnzahlPatienten = 0;
        self->AnzahlObjekte = 0;
};


method body export_OID in class CPatient
{ return self->OID; };


method body Bearbeiten in class CDraht
{       int flag = false;
        Lk_mask mk;
        Lk_presentation pid;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;}
        mk = lk_protected(0,0,0);
        pid = lk_present(self,mk);
        /* self->edit*/
        mk_map(pid, LK_MOUSE, 0, 0, 0 ,0);
        lk_grab (pid);
        if(o2_in_transaction() && flag)
                validate;
};


method body init in class CDraht
{ self->OID = OIDHandler->LiefereCDrahtOID; };


method body export_object in class CDraht
{
#include "stdio.h"
```

```
#include "stdlib.h"
  FILE *fp;
  if ( !(fp = fopen("/home/pg/pg290/gerding/.angiofdbs", "w+")) )
  {
    fprintf ( stderr, "Couldn't open pipe!\n" );
  }
  else
  {
    fprintf( fp, "CMaterial\n" );
    fprintf( fp, "%d\n", self->OID );
    fprintf( fp, "%d\n", BOID );
    fprintf( fp, "Bezeichnung\n%s\n", self->Produkt );
    fprintf( fp, "Preis\n%f\n", self->Preis );
    fprintf( fp, "#EOO#\n" );
    fclose( fp );
    sleep(1);
}};


method body Bearbeiten in class CKatheter
{       int flag = false;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;}
        self->edit;
        if(o2_in_transaction() && flag)
                validate;
};


method body init in class CKatheter
{ self->OID = OIDHandler->LiefereCKatheterOID; };


method body export_object in class CKatheter
{#include "stdio.h"
#include "stdlib.h"
  FILE *fp;
  if ( !(fp = fopen("/home/pg/pg290/gerding/.angiofdbs", "w+")) )
  {
    fprintf ( stderr, "Couldn't open pipe!\n" );
  }
  else
  {
    fprintf( fp, "CMaterial\n" );
    fprintf( fp, "%d\n", self->OID );
    fprintf( fp, "%d\n", BOID );
    fprintf( fp, "Bezeichnung\n%s\n", self->Produkt );
    fprintf( fp, "Preis\n%f\n", self->Preis );
    fprintf( fp, "#EOO#\n" );
    fclose( fp );
    sleep(1);
}};



method body Bearbeiten in class CMedikament
{       int flag = false;
        if(!o2_in_transaction())
```

```
        {       transaction;
                flag = true; }
        self->edit;
        if(o2_in_transaction() && flag)
                validate;
};


method body init in class CMedikament
{ self->OID = OIDHandler->LiefereCMedikamentOID; };

method body export_object in class CMedikament
{
#include "stdio.h"
#include "stdlib.h"
  FILE *fp;
  if ( !(fp = fopen("/home/pg/pg290/gerding/.angiofdbs", "w+")) )
  {
    fprintf ( stderr, "Couldn't open pipe!\n" );
  }
  else
  {
    fprintf( fp, "CMaterial\n" );
    fprintf( fp, "%d\n", self->OID );
    fprintf( fp, "%d\n", BOID );
    fprintf( fp, "Bezeichnung\n%s\n", self->Produkt );
    fprintf( fp, "Preis\n%f\n", self->Preis );
    fprintf( fp, "#EOO#\n" );
    fclose( fp );
    sleep(1);
  }
};


method body neues_Medikament in class CTherapie
{       o2 Box mMedikamente     = new Box;
        o2 list(string)         MedikamenteListe;
        o2 list(string)         gesamteMedikamente;
        o2 CMedikament          einMedikament;
        o2 string               einMedikamentstr;
        o2 set(CMedikament)     queryresMed;

        for (einMedikament in alleMedikamente)
        {
                gesamteMedikamente += list(einMedikament->Name);
        }
        MedikamenteListe = mMedikamente -> mult_selection ("Liste der Medikamente",
        "", gesamteMedikamente);

        for (einMedikamentstr in MedikamenteListe)
        {       o2query(queryresMed,"select x \
                from x in alleMedikamente \
                where x.Name like $1", einMedikamentstr);

                for(einMedikament in queryresMed)
        {   self->Medikamente += set(einMedikament);
            einMedikament->export_object(self->BOID);
        }
        }
```

```
        self->refresh_all;
};


method body init in class CTherapie
{       o2 CArzt a1 = new CArzt;
        o2 CArzt a2 = new CArzt;
        o2 CArzt a3 = new CArzt;
        o2 CArzt a4 = new CArzt;
        o2 CArzt a5 = new CArzt;
        self->BOID = hBOID;
        self->OID = OIDHandler->LiefereCTherapieOID;
        self->Datum = new Date(0, 0, 0);
        self->Mitwirkende.erster_Untersucher = a1;
        self->Mitwirkende.zweiter_Untersucher = a2;
        self->Mitwirkende.erster_steriler_Ass = a3;
        self->Mitwirkende.zweiter_steriler_Ass= a4;
        self->Mitwirkende.unsteriler_Ass = a5;
        self->Punktionsstelle = new CPunktion;
};


program body Suchen in application angio
{       o2 Box mFrage = new Box;
        o2 string y;
        o2 set(CPatient) mres;

        y = mFrage -> dialog ("Bitte geben Sie den
        gesuchten Nachnamen des Patienten ein","");

        o2query(mres, "select x \
                from x in allePatienten \
                where x.Name like $1",y);
        display (mres);
};


transaction body init in application angio
{   o2 COIDHandler hilfe;
        o2 integer flag = false;
        if(!o2_in_transaction())
        {       transaction;
                flag = true;
        }

        hilfe = new COIDHandler;
        hilfe->init;
    OIDHandler = hilfe;

        if(o2_in_transaction() && flag)
                        validate;

        display(OIDHandler);
}
;
```

16.4 Test des Angiographiesystems

Beim Starten der Applikation erscheint ein Fenster, und durch Anklicken mit der rechten Maustaste auf `angio` wird das Hauptmenü angezeigt. Zum Testen werden folgende Schritte ausgeführt.

1. Der Menüpunkt `Neuer_Patient` wird ausgewählt, dann werden Patientenstammdaten eingegeben. Bei der Eingabe des Geburtsdatums muß man mit der rechten Maustaste auf `Date` klicken und den Menüpunkt `edit` auswählen
2. Es wird eine leere Behandlung angelegt. Mit der rechten Maustaste anklicken und Bearbeiten auswählen.
3. Neue Therapie anlegen und auf die neu angelegte Therapie mit der rechten Maustaste anklicken und `Bearbeiten` auswählen.
4. `erster_Untersucher` eingeben
5. Mit der rechten Maustaste auf `methods` anklicken und `neuer_Katheter` bzw. `neues_Medikament` auswählen, dann erscheint eine Liste von Katheter bzw. Medikamenten.

Bemerkungen

1. Die geöfneten Fenster müssen in bestimmter Reihenfolge geschlossen werden, nämlich das zuletzt geöffnete Fenster wird zuerst geschlossen.
2. Bei der Eingabe eines neuen Medikaments bzw. eines Katheters wird nicht überprüft, ob das Medikament bzw. Katheter schon in der Liste vorhanden ist.
3. Bei der Aufnahme eines neuen Patienten wird nicht überprüft, ob die Patientenstammdaten schon eingegeben wurden, sondern es wird automatisch eine neue `OID` vergeben und die Patientenstammdaten nochmal gespeichert.

Kapitel 17

Schemaintegration

Verfasser: Mario Ellebrecht, Peter Ziesche

17.1 Die UML-Modelle

Als Basis für die Föderierung der Daten aus verschiedenen Schemata dient ein föderiertes Schema. Dieses wurde manuell erstellt. Als Grundlage dienten die UML-Modellierungen der Schmata der beiden zu föderierenden Datenbanken aus dem Angiographiesystem und dem Abrechnungssystem, die in den Abbildungen 17.1 und 17.2 abgebildet sind.

17.2 Das föderierte Schema

Die Überschneidungen in den gespeicherten Daten beider Datenbanken sind zu föderieren. Dies sind hauptsächlich die Patientenstammdaten, die im Abrechnungssystem eingegeben werden, und die Materialdaten, die aus dem Angiographiesystem kommen.

17.3 Das Datenmodel für den Föderierungsgraphen

Im folgenden wird das konkrete Datenmodell beschrieben, daß im Rahmen der Projektgruppe zur Föderierung der Anwendungen benutzt wurde. [1] Eine Übersicht über die Schemata gibt Abbildung 17.3.

[1] Bei der Notation handelt es sich um die *Object-Definition-Language* (*ODL*), die um das Schlüsselwort `schema` erweitert wurde.

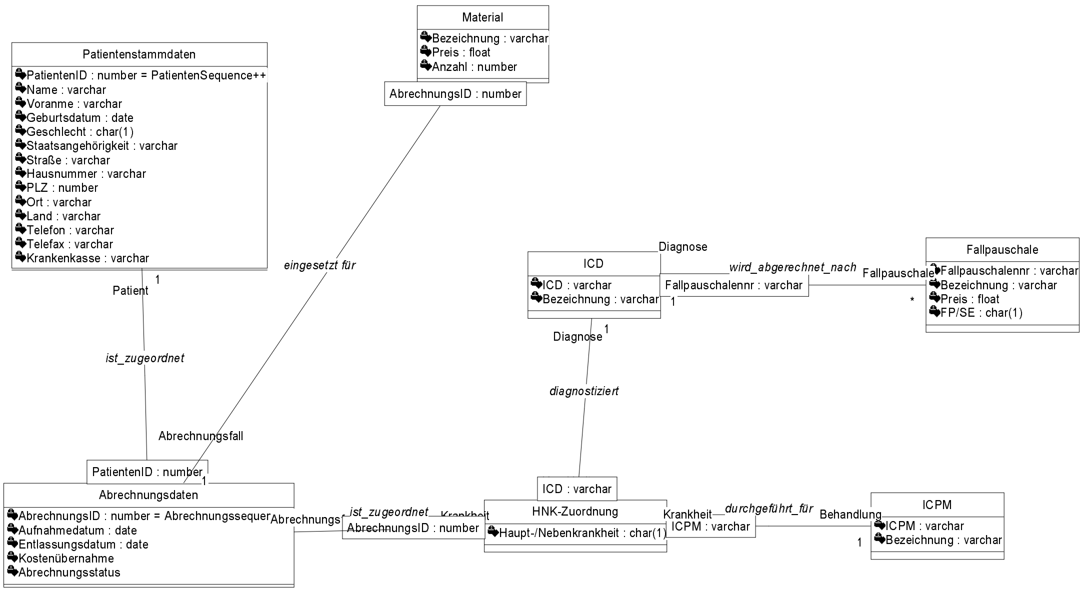


Abbildung 17.1: Das UML-Diagramm des Abrechnungssystems

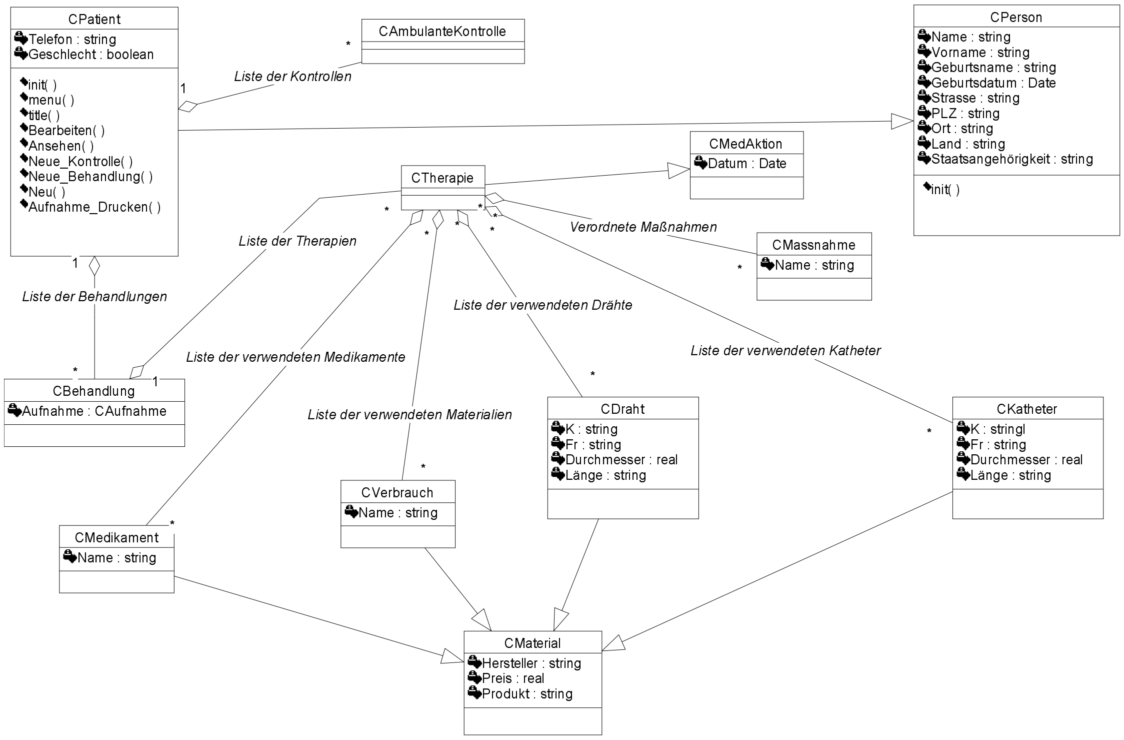


Abbildung 17.2: Das UML-Diagramm des Angiographiesystems

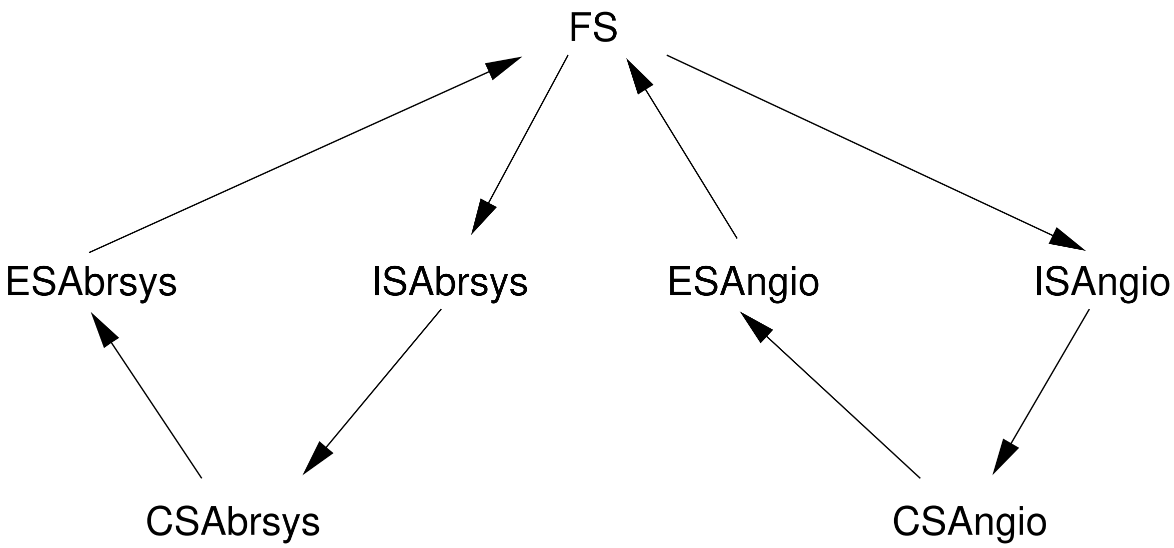


Abbildung 17.3: Die Schemata des Föderierungssystem

```
//
// Das Komponentenschema des Abrechnungs-Systems
//
schema CSAbrsys
{
  class Patient
  {
    string Name;
    string Vorname;
    date   Geburtsdatum;
    int    PLZ;
    string Ort;
    string Land;
    string Staatsangehoerigkeit;
    char   Geschlecht;
    string Telefon;
    string Telefax;
    string Krankenkasse;
  };

  class Abrechnungsdaten
  {
    Patient* PatientenID;
  };

  class Material
  {
    string            Bezeichnung;
    float             Preis;
    Abrechnungsdaten* AbrechnungsID;
  };
};

//
// Das Importschema des Abrechnungs-Systems
//
schema ISAbrsys
{
  class Patient
  {
    list< Behandlung* > Behandlungen
      invers Behandlung::PatientenID;
  };

  class Behandlung
  {
    Patient* PatientenID
      invers Patient::Behandlungen;
    list< Material* > Materialien
      invers Material::AbrechnungsID;
  };

  class Material
  {
    string Bezeichnung;
    float  Preis;
    Behandlung* AbrechnungsID
      invers Behandlung::Materialien;
  };
```

```
};

//
// Das Exportschema des Abrechnungs-Systems
//
schema ESAbrsys
{
  class Patient
  {
    string Name;
    string Vorname;
    date   Geburtsdatum;
    string PLZ;
    string Ort;
    string Land;
    string Staatsangehoerigkeit;
    bool   Geschlecht;
    string Telefon;
  };
};

//
// Das Importschema des Angiographie-Systems
//
schema ISAngio
{
  class Patient
  {
    string Name;
    string Vorname;
    date   Geburtsdatum;
    string PLZ;
    string Ort;
    string Land;
    string Staatsangehoerigkeit;
    bool   Geschlecht;
    string Telefon;
    string Telefax;
    string Krankenkasse;
  };
};

//
// Das Exportschema des Angiographie-Systems
//
schema ESAngio
{
  class Patient
  {
    list< Behandlung* > Behandlungen
      invers Behandlung::Patient;
  };

  class Behandlung
  {
    Patient* Patient
      invers Patient::Behandlungen;
    list< Material* > Materialien
      invers Material::Behandlung;
```

```
  };

  class Material
  {
    string Bezeichnung;
    float  Preis;
    Behandlung* Behandlung
      invers Behandlung::Materialien;
  };
};

//
// Das Komponentenschema des Angiographie-Systems
//
schema CSAngio
{
  class CPatient
  {
    string Name;
    string Vorname;
    date   Geburtsdatum;
    string PLZ;
    string Ort;
    string Land;
    string Staatsangehoerigkeit;
    bool   Geschlecht;
    string Telefon;
    list< CBehandlung* > Behandlungen
      invers CBehandlung::Patient;
  };

  class CBehandlung
  {
    CPatient* Patient
      invers CPatient::Behandlungen;
    list< CTherapie* > Therapien
      invers CTherapie::Behandlung;
  };

  class CTherapie
  {
    Behandlung* Behandlung
      invers Behandlung::Therapien;
    list< CMaterial* > Materialien
      invers CMaterial::Therapie;
  };

  class CMaterial
  {
    string Bezeichnung;
    float  Preis;
    CTherapie* Therapie
      invers CTherapie::Materialien;
  };
};

//
// Das föderierte Schema
//
```

```
schema FS
{
  class Patient
  {
    string Name;
    string Vorname;
    date   Geburtsdatum;
    string PLZ;
    string Ort;
    string Land;
    string Staatsangehoerigkeit;
    bool   Geschlecht;
    string Telefon;
    list< Behandlung* > Behandlungen
      invers Behandlung::Patient;
  };

  class Behandlung
  {
    Patient* Patient
      invers Patient::Behandlungen;
    list< Matarial* > Materialien
      invers Material::Behandlung;
  };

  class Material
  {
    string Bezeichnung;
    float  Preis;
    Behandlung* Behandlung
      invers Behandlung::Materialien;
  };
};
```

Kapitel 18

Die Kommunikationsschnittstelle

Verfasser: Peter Ziesche

18.1 Einleitung

Eine der Vorgaben bei der Entwicklung des Föderierungssystems war die Verwendung von CORBA zur Interprozeß-Kommunikation. Zu Beginn stand jedoch nicht fest, welche Daten von wo nach wo fließen würden und welche Struktur diese Daten haben würden. Ferner war nur sehr wenig über die Benutzung des CHORUS COOL-ORB bekannt, so daß wir nicht wußten, wie kompliziert die Verwendung von CORBA ist.

Es wurde daher eine Kommunikations-Schnittstelle entwickelt, die im wesentlichen zwei Zielsetzungen verfolgt:

- Erstens sollte sie sehr flexibel im Bezug auf die Struktur der zu übertragenen Daten sein um mit häufigen Änderungen problemlos fertig zu werden.
- Zweitens sollte die CORBA-Schnittstelle so weit wie möglich eingekapselt werden. Dadurch sollte die Zahl der PG-Teilnehmer, die sich mit der Benutzung des COOL-ORB befassen müssen, auf ein Minimum beschränkt werden.

18.2 Architektur

Die Flexibilität in Bezug auf die Struktur der zu auszutauschenden Daten basiert auf Operationsobjekten und Operations-Handlern.

18.2.1 Operationsobjekte

Operationsobjekte sind Instanzen von C++-Klassen. Eine solche Klasse bildet in ihrem Aufbau die Struktur der zu übermittelnden Daten nach. Da die Kommunikations-Schnittstelle lediglich die Serialisierbarkeit der Objekte (siehe Abschnitt 18.4.1) verlangt, können die Klassen einen beliegigen Aufbau haben und optimal an die Daten angepaßt werden.

Wenn ein Prozeß Daten versenden will, instanziiert er ein entsprechendes Operations-Objekt, füllt es mit Daten und verschickt es. Der Empfänger liest diese Daten aus, verarbeitet sie und schreibt eventuell Ergebnisse in das Operationsobjekt zurück. Diese werden dann automatisch wieder an den Sender geleitet. Das Füllen eines Operationsobjektes mit Daten kann im einfachsten Fall durch die

Belegung von Attributen mit Werten erfolgen. Es sind jedoch auch komplexere Methoden denkbar, die das Füllen weitgehend automatisch erledigen. Die Kommunikations-Schnittstelle setzt hier keine Beschränkungen.

18.2.2 Operations-Handler

Der Empfänger einer Operation muß die gelieferten Daten verarbeiten. Die technischen Details des Empfangs sollen für ihn jedoch unsichtbar bleiben. Um dies zu erreichen, werden Operations-Handler definiert. Dabei handelt es sich um C++-Klassen, die eine `Handle`-Methode bereitstellen. In dieser erfolgt dann die Verarbeitung der Daten. Da auch hier keine weiteren Anforderungen an die Klasse gestellt werden, kann die Verarbeitung beliebig komplex werden.

Der Empfänger teilt der Kommunikations-Schnittstelle mit, welche Operationen er verarbeiten will und welche Operations-Handler für welche Operation zuständig sein sollen. Empfängt die Kommunikations-Schnittstelle eine Operation, wird ein entsprechendes Handler-Objekt erzeugt, die Operation an die `Handle`-Methode übergeben und die Verarbeitung somit gestartet.

18.2.3 Sender und Empfänger

Operationen werden von *einer* Komponente an *eine* andere Komponente geschickt. Dazu muß der Sender den Namen des Empfängers kennen. Ein Name ist ein String, über den der Empfänger identifiziert und angesprochen werden kann.

Sender und Empfänger sind wieder in Objekte gekapselt. Das Versenden einer Operation geschieht durch Instanziierung eines Senders, der Angabe des Empfängers und Übergabe der zu versendenden Operation. Der Versand erfolgt dann automatisch, der sendende Prozeß erhält die Kontrolle erst dann wieder zurck, wenn das Ergebnis vom Empfänger vorliegt.

Jedes Sender-Objekt verschickt Operationen nur an einen Empfänger. Dadurch braucht der Empfängername nur einmal bei der Instanziierung angegeben zu werden.

Der empfangende Prozeß teilt der Kommunikations-Schnittstelle mit, welche Operationen von welchen Operations-Handlern verarbeitet werden sollen. Anschließend kann er ein Empfänger-Objekt instanziieren und diesem die Kontrolle übergeben. Das Empfänger-Objekt ruft beim Empfang einer Operation automatisch den zugehörigen Handler auf.

Da der Prozeß die Kontrolle an die Kommunikations-Schnittstelle abgibt (genauer an das Empfänger-Objekt), kann nur jeweils ein Empfänger-Objekt aktiv sein. Die Schnittstelle erlaubt daher nur ein einziges Empfänger-Objekt zu einer bestimmten Zeit.

18.3 Entwurfsmuster in der Kommunikations-Schnittstelle

Um bei der Entwicklung der Kommunikations-Schnittstelle auf bereits bestehende Konzepte zurückgreifen zu können, wurden in der Entwurfsphase der Schnittstelle Entwurfsmuster (siehe [G$^+$95]) eingesetzt.

18.3.1 Operations-Objekte und -Handler sind Prototypen

Da die Schnittstelle beliebige Operationen versenden und empfangen können soll, muß sie unabhängig von bestimmten Operationen und Handlern sein. Dies bedeutet, daß ein Handler instanziiert werden muß, ohne den Namen der Klasse fest zu codieren. Die in C++ übliche Schreibweise

```
CxyHandler* handler = new CxyHandler;
```

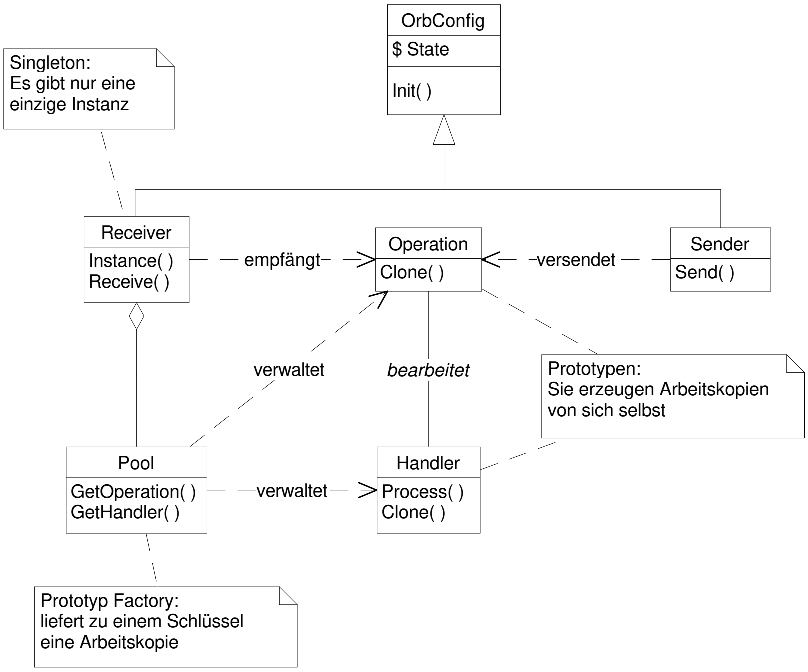


Abbildung 18.1: Architektur der Kommunikations-Schnittstelle

kann somit nicht benutzt werden.

Stattdessen wird ein Handler einmal instanziiert, bevor Operationen empfangen werden. Diese Instanz wird als Prototyp bezeichnet. Der Handler verfügt über die Methode `Clone`, die eine Kopie von sich selbst zurückliefert. `Clone` ist als rein virtuelle Funktion in einer Basisklasse (`CHandler`) deklariert, von der alle Handlerklassen erben.

Der so erzeugte Prototyp wird bei der Kommunikations-Schnittstelle registriert, die ihn nur polymorph über die Schnitttelle der Basisklasse anspricht. Wird nun eine Operation empfangen, so ruft das Empfänger-Objekt die `Clone`-Methode des Prototypen auf und beauftragt die Kopie mit der Verarbeitung der Operation. Nach Abschluß der Verarbeitung wird diese Kopie wieder zerstört.

Auch die Operations-Objekte müssen über eine `Clone`-Methode verfügen und einmal als Prototypen instanziiert werden. Dies hängt damit zusammen, daß von CORBA nicht das Objekt selbst, sondern nur ein serielisiertes Objekt physisch übertragen wird. Der Empfänger muß diese Daten beim Empfang wieder in ein Objekt transformieren.

Die Instanziierung der Prototypen muß durch den jeweiligen Prozeß erfolgen, der die Kommunikations-Schnittstelle nutzen will. Um den Implementierungsaufwand an dieser Stelle so gering wie möglich zu halten, stellt die Kommunikations-Schnittstelle leicht zu benutzende Makros zur Verfügung (siehe Abschnitt 18.4.3).

18.3.2 Der Empfänger ist ein Singleton

Wie bereits erwähnt, kann immer nur ein Empfänger-Objekt gleichzeitig aktiv sein. Die Kommunikationschnittstelle erlaubt daher nur eine einzige Instanziierung der Klasse `CReceiver`. Um dies zu erreichen, wird das Entwurfsmuster *Singleton* benutzt.

Der Konstruktor der Klasse ist `private`. Dadurch kann die Klasse von außen nicht instanziiert werden. Ferner existiert eine `static`-Methode[1] namens `Instance`, die einen Zeiger auf ein `CReceiver`-Objekt zurückliefert. `Instance` ist nun so implementiert, daß es einmal ein Objekt erzeugt, sich dessen Adresse in einer `static`-Variablen merkt und bei jedem Aufruf diesen Zeiger zurückliefert.

Auf diese Weise kann an jeder Stelle im Programm, an der die Klasse `CReceiver` bekannt ist, auf diese einzige Instanz (den *Singleton*) zugegriffen werden.

18.4 Die Implementierung

18.4.1 Die Klassen des Kommunikations-Interfaces

Die Kommunikations-Schnittstelle setzt sich aus einer Reihe von Klassen zusammen. Die Schnittstellen werden in der Datei `comi.H` definiert. Durch das Einbinden dieser Datei steht die Schnittstelle zur Benutzung bereit. Im folgenden wird eine Übersicht über die Klassen gegeben. In Kapitel 18.5 werden Beispiele für die Benutzung der Kommunikations-Schnittstelle gegeben.

[1] `static`-Methoden werden nicht auf ein Objekt, sondern auf eine Klasse angewendet. Der Aufruf erfolgt in der Form `CReceiver::Instance()`.

Die Klasse `CReceiver`

Eine Anwendung kann nur eine einzige Instanz dieser Klasse erzeugen (Entwurfsmuster Singleton, Abschnitt 18.3.2). Vor der Benutzung muß diese Instanz mit `Init()` initialisiert werden.

Anschließend wird mit `WaitForOperation()` eine Warteschleife betreten. Von dort aus werden empfangene Operationen automatisch verarbeitet, indem je nach Typ der Operation ein entsprechender Handler erzeugt wird, der dann die Weiterverarbeitung übernimmt.

Alle Methoden, deren Rückgabewerte vom Typ `bool` sind, liefern im Falle der erfolgreichen Ausführung `true` als Ergebnis, sonst `false`. Nähere Informationen über den Fehler können anschließend über die Methode `GetLastError()` ermittelt werden. Ein von diesem Standard abweichendes Verhalten geht aus der Beschreibung einer Methode hervor.

`static CReceiver* Instance()`
: Die (*Klassen*-)Methode liefert einen Zeiger auf die einzige Instanz der Klasse `CReceiver`. Da sie statisch deklariert ist, muß der Aufruf die Form `CReceiver* poRec = CReceiver::Instance()` haben.

`static Destroy()`
: Da ein Objekt dieser Klasse vom Benutzer auf direktem Weg weder erzeugt noch zerstört werden kann, ist an Stelle eines `delete`-Aufrufes der Aufruf `CReceiver::Destroy()` erforderlich.

`bool Init( const string i_sReceiverName)`
: Die Methode initialisiert den Empfänger. Der Parameter `i_sReceiverName` legt den Namen fest, unter dem der Empfänger von einem Sender angesprochen werden kann.

`bool Terminate()`
: Sorgt für ein ordnungsgemäßes Beenden des Empfängers.

`void WaitForOperation()`
: Mit dieser Methode wird in eine Endlosschleife gesprungen, in der Operationen empfangen und automatisch verarbeitet werden (siehe auch Beschreibung von `CReceiver`).

`void Shutdown()`
: Wenn die mit `WaitForOperation` betretene Warteschleife wieder verlassen werden soll, so muß diese Methode aufgerufen werden. Dies geschieht in der Regel innerhalb eines Handlers. Der Empfang von Operationen wird daraufhin beendet. Der aufrufende Handler wird jedoch noch bis zum Ende ausgeführt. Erst wenn er seine Arbeit beendet hat wird die Warteschleife verlassen.

`bool RegisterOperation( COperation* OpPrototype, CHandler* HandlerPrototype )`
: Die Methode registriert ein Operations-Handler-Paar. Alle zukünftig empfangenen Operationen des übergebenen Typs werden durch den übergebenen Handler verarbeitet. Bei den beiden Objekten handelt es sich um Prototypen (Abschnitt 18.3.1), von denen automatisch Kopien erstellt werden. Da der Aufruf dieser Methode durch Makros automatisiert wird, wird ein Benutzer sie normalerweise nicht direkt aufrufen.

`unsigned int GetLastError()`
: Wenn eine boolsche Methode `false` als Ergebnis liefert, ihre Ausführung also fehlgeschlagen ist, liefert diese Methode eine Fehlernummer, die den aufgetretenen Fehler genauer beschreibt.

`string Name()`
: Die Methode liefert den Namen, unter dem der Empfänger beim System bekannt ist. Er entspricht dem bei `Init` übergebenen String.

`bool Receive( TSerialOperation ReceivedOperation )`
: Diese Methode wird immer dann aufgerufen, wenn eine Operation empfangen wurde. Sie sorgt dafür, daß die serialisierte Operation in ein Objket konvertiert wird, die Operations- und Handler-Prototypen kopiert und die Operation verarbeitet wird. Der Aufruf dieser Methode erfolgt durch das CORBA-System. Der Benutzer wird diese Methode nicht selbst aufrufen.

Die Klasse `CSender`

Die Klasse dient als Schnittstelle für den Versand von Operationen. Jede Instanz dieser Klasse versendet Operationen an genau einen Empfänger. Bevor zum ersten Mal eine Operation erzeugt wird, muß ein Objekt mit `Init()` initialisiert werden.

Der Versand von Operationen erfolgt durch den Aufruf von `Send()`. Der Benutzer ist für die anschließende Zerstörung des Operations-Objektes zuständig.

Alle Methoden, deren Rückgabewerte vom Typ `bool` sind, liefern im Falle der erfolgreichen Ausführung `true` als Ergebnis, sonst `false`. Nähere Informationen über den Fehler können anschließend über die Methode `GetLastError()` ermittelt werden. Ein von diesem Standard abweichendes Verhalten geht aus der Beschreibung einer Methode hervor.

`bool Init( const string i_sReceiverName )`
: Initialisiert ein Sender-Objekt und legt den Empfänger für den Versand von Operationen fest.

`bool Terminate()`
: Sorgt für ein ordnungsgemäßes Beenden des Senders.

`int GetLastError()`
: Die Routine liefert eine Fehlernummer, die dem Aufrufer Aufschluß über die Art eines Fehlers gibt, wenn eine Methode `false` geliefert hat.

`bool Send( COperation* i_poOperation )`
: Der Aufruf von Send versendet eine Operation an den bei der Initialisierung bestimmten Empfänger. Der Versand erfolgt synchron, d.h. die Methode kehrt erst dann wieder zurück, wenn die Operation beim Empfänger verarbeitet worden ist.

Die Klasse `COperation`

Dies ist die abstrakte Basisklasse für Operationen (siehe Abschnitt 18.2.1). Alle Methoden dieser Klasse sind rein virtuell, d.h. nicht implementiert. Die Implementierung erfolgt erst in einer abgeleiteten Klasse.

`bool IsComplete()`
: Mit dieser Funktion kann überprüft werden, ob alle Attribute eines Operations-Objektes mit Daten belegt wurden, die Operation also versandt werden kann.

`COperation* Clone()`
: Diese Methode dient dazu, Kopien eines Prototypen zu machen. Sie liefert ein neues Operationobjekt zurück.

`int Id()`
: Die Methode liefert die ID der Operation. Über die ID kann der Typ einer Operation ermittelt werden. Dies ist z.B. wichtig, um den richtigen Handler bestimmen zu können.

`bool ToSerial( TSerialOperation* Sequence )`
: Diese Methode wird aufgerufen, wenn sich die Operation für den Versand serialisieren soll. Alle Daten müssen in die übergebene Sequenz hinein geschrieben werden (siehe Abschnitt 18.4.1).

`bool FromSerial( TSerialOperation* Sequence )`
: Dies ist die inverse Methode zu `ToSerial`. Aus der seriellen Operation werden die Daten ausgelesen und in die internen Attribute des Objektes geschrieben.

Die Klasse `CHandler`

Diese Klasse definiert die Schnittstelle für einen Operations-Handler. Von ihr abgeleitete Klassen dienen dazu, Operationen zu verarbeiten. Dies geschieht dadurch, daß das die Operation empfangende `CReceiver`-Objekt die `Handle`-Methode aufruft und ihr die Operation übergibt. Alle Methoden dieser Klasse sind rein virtuell, d.h. nicht implementiert. Die Implementierung erfolgt erst in einer abgeleiteten Klasse.

`CHandler* Clone()`
: Diese Methode dient dazu, Kopien eines Prototypen zu erzeugen. Sie liefert ein neues Handlerobjekt zurück.

`bool Handle( COperation* Op )`
: Der Methode wird eine empfangene Operation übergeben. Diese wird von der Methode verarbeitet.

Die Klasse `COrbUser`

Ein Prozeß, der die Kommunikations-Schnittstelle benutzt, soll in der Lage sein, im Zuge der Verarbeitung einer empfangenen Nachricht wiederum Nachrichten an andere Prozesse zu versenden. Dies bedeutet, daß von den Klassen `CSender` und `CReceiver` gleichzeitig Instanzen existieren können. Für den Zugriff auf die Funktionen des COOL-ORB müssen diese Objekte bestimmte Daten gemeinsam nutzen. Darüber hinaus müssen diese Daten bei der Zerstörung eines `CSender`- oder `CReceiver`-Objektes erhalten bleiben. Die Klasse `COrbUser` verkapselt diese gemeinsam genutzten Daten. Für die Benutzung der Kommunikations-Schnittstelle wird diese Klasse nicht benötigt.

`bool InitOrb()`
: Mit `InitOrb()` wird die einmalige Initialisierung des COOL-ORB vorgenommen. Anschließend kann der ORB zur Interprozeß-Kommunikation benutzt werden.

`bool IsOrbInitialized()`
: Diese Methode liefert `true`, wenn der ORB korrekt initialisiert ist. Sie dient dazu, um vor jedem Aufruf von `InitOrb()` zu prüfen, ob eine Initialisierung notwendig ist oder nicht.

`CORBA_ORB_ptr ObjectRequestBroker()`
: Diese Methode liefert einen Zeiger auf den ORB. Die Klasse `CReceiver` (siehe Abschnitt 18.4.1) benötigt diesen Zeiger intern, um Informationen über das System abzufragen.

`COOL_NamingService_var NamingService()`
: Mit dieser Methode kann der Naming-Service des COOL-ORB angesprochen werden. Sowohl `CSender` als auch `CReceiver` nutzen diese Methode intern.

Die Klasse `COrbImpl`

Diese Klasse stellt die Implementierung der IDL-Schnittstelle der Kommunikations-Schnittstelle bereit. Sie verfügt nur über die Methode

```
CORBA_Long HandleOp( TSerialOp& aOp, CORBA_Environment& env );
```

Diese Methode wird vom CORBA-Laufzeitsystem aufgerufen, wenn eine Nachricht für dieses Objekt empfangen wurde. Für die Benutzung der Kommunikations-Schnittstelle wird diese Klasse nicht benötigt.

Die Klasse CDecoder

Die Versendung von Operationen erfolgt in mehreren Schritten. Um die Serialisierung eines Operations-Objektes möglichst einfach zu halten, erfolgt sie in ein Array-Objekt der Standard-Template-Library. Die Klasse `CDecoder` übernimmt beim Senden die Konvertierung von einem STL-Array in eine CORBA-Sequenz und beim Empfang die Zurück-Konvertierung von einer CORBA-Sequenz in ein STL-Array. Die beiden sehr eng miteinander verbundenen Funktionen sind somit in einer getrennten Klasse verkapselt.

Für die Benutzung der Kommunikations-Schnittstelle wird diese Klasse nicht benötigt.

18.4.2 Die IDL-Schnittstelle

Mit der IDL (Interface Definition Language) wird die Schnittstelle eines CORBA-Objektes spezifiziert. Jeder Prozeß, der Operationen empfängt ist ein CORBA-Objekt. Die Schnittstelle ist jedoch für alle Empfänger gleich, unabhängig davon, welche Operationen er empfangen will.

Die Schnittstelle sieht wie folgt aus[2]:

```
interface Orb
{
        typedef sequence< string> TSerialOp;

        long HandleOp( inout TSerialOp aOp );
};
```

18.4.3 Globale Präprozessor-Makros

Registrierung von Operationen
Um die Erzeugung von Prototypen zu vereinfachen, existiert das Makro `REGISTER_OPERATION`. Es erhält die Klassen für ein Operations-Handler-Paar als Makroparameter, erzeugt daraus Prototypen und kümmert sich um den korrekten Aufruf von `CReceiver::RegisterOperation`.

```
#define REGISTER_OPERATION( _COperation, _CHandler ) \
        CReceiver::Instance()->RegisterOperation(  \
        new _COperation,  new _CHandler )
```

Kopieren von Prototypen
Das Kopieren eines Prototypen sieht für alle Handler- bzw. Operationsklassen gleich aus. Daher kann die `Clone`-Methode auch mit Hilfe eines Makros implementiert werden. Die eigentliche Arbeit beim Kopieren leistet der Copy-Konstruktor der Klasse.

```
#define IMPLEMENT_CLONE_OPERAION( _CLASS ) \
        COperation* _CLASS::Clone() \
        { return (COperation*) new _CLASS( this );}

#define IMPLEMENT_CLONE_HANDLER( _CLASS ) \
        CHandler* _CLASS::Clone() \
        { return (CHandler*) new _CLASS( this );}
```

Deklaration von Standard-Methoden Eine von `COperation` erbende Klasse muß eine Reihe von Methoden überschreiben. Um dabei Schreibarbeit zu sparen, können die Deklarationen aller dieser Standard-Methoden durch einen einzigen Makro-Aufruf eingefügt werden.

[2] Der Typ `TSerialOp` wird von der Kommunikations-Schnittstelle intern benutzt. Er ist *nicht* mit `TSerialOperation` identisch, den die von `COperation` erbenden Klassen zur Serialisierung verwenden müssen.

```
#define DECLARE_STANDARD_METHODS( _CLASS ) \
        _CLASS(); \
        _CLASS( const _CLASS* other ); \
        virtual COperation* Clone(); \
        virtual int Id(); \
        virtual IsComplete(); \
        virtual bool ToSerial( TSerialOperation* paSequenz ); \
        virtual bool FromSerial( const TSerialOperation* paSequence )
```

18.5 Die Benutzung der Kommunikations-Schnittstelle

18.5.1 Ein Beispiel für einen Empfänger

```
#include "comi.H"

int main()
{
   // Empfaenger erzeugen und initialisieren
   CReceiver* poReceiver = CReceiver::Instance();
   poReceiver->Init( "ORACLE_DB" );

   // Eigener Initialisierungscode
   ...

   // Endlosschleife fuer Empfang
   poReceiver->WaitForOperation();

   // Programm beenden
   poReceiver->Terminate();
   CReceiver::Destroy();

   return 0;
} //main
```

18.5.2 Ein Beispiel für einen Sender

```
#include "comi.H"

int main()
{
   // Sender erzeugen und initialisieren
   CSender oSender;
   oSender.Init( "ORACLE_DB" );

   // Eigener Initialisierungscode
   ...

   // Operation erzeugen und mit Daten fuellen
   CMyOperation Op;
   ...

   // Operation versenden
   oSender.Send( &Op );

   // Programm beenden
   oSender.Terminate();
```

```
   return 0;
} //main
```

18.5.3 Ein Beispiel für einen Operations-Handler

```
#include "comi.H"

class CMyHandler : public CHandler
{
   CMyHandler( CMyHandler* other )
   {
      // Code fuer den Copy-Konstruktor
   }


   virtual CHandler* Clone()
   {
      return new CMyCHandler( this );
   }


   virtual bool Handle( COperation* Op )
   {
      // Operation downcasten
      CMyOperation* MyOp = (CMyOperation*) Op;

      // verarbeite die Operation
      ...

      return true;
   }
};
```

Da hier keine Unterteilung in Schnittstelle und Implementierung stattfindet, wurde die `Clone`-Methode nicht mit dem oben vorgestellten Makro (siehe Abschnitt 18.4.3) implementiert.

18.5.4 Ein Beispiel für eine Operation

```
#include "comi.H"

//
// Interface
//
class CMyOperation : public COperation
{
   DECLARE_STANDARD_METHODS( CMyOperation );

   int  EinDatum;
   float EinAnderesDatum;
}

//
// Implementierung
//
CMyOperation::CMyOperation()
  : EinDatum( 0 ),
```

```
    EinAnderesDatum( 0.0 )
{
   // nothing to do here
}


CMyOperation::CMyOperation( const CMyOperation* other )
  : EinDatum( other->EinDatum ),
    EinAnderesDatum( other->EinAnderesDatum )
{
  // nothing to do here
}


int CMyOperation::Id()
{
  // CMyOperation hat z.B. die ID 5000
  return 5000;
}


bool CMyOperation::ToSerial( TSerialOperation* Sequence )
{
   // IntToString und RealToString sind nicht Bestandteil
   // der Kommunikationsschnittstelle

   // Die ID muss zuerst geschrieben werden
   Sequence->push_back( IntToString( Id() ) );

   Sequence->push_back( IntToString( EinDatum ) );
   Sequence->push_back( FloatToString( EinAnderesDatum ) );

   return true;
}


bool CMyOperation::FromSerial( TSerialOperation* Sequence )
{
   TSerialOperation::iterator i;

   i = Sequence->begin;

   if( *i == IntToString( Id() ) ) {
      i++;
      EinDatum = StringToInt( *i );
      i++;
      EinAndresDatum = StringToFloat( *i );
      return true;
   }
   else {
      // falsche Operation
      return false;
   }
}

IMPLEMENT_CLONE_OPERATION( CMyOperation )
```

Die Serialisierung ist zur Zeit noch etwas aufwendig, da eine serielle Operation nur Strings aufnehmen kann. Dies wird in einer künftigen Version jedoch einfacher.

18.6 Die Kommunikations-Schnittstelle im Rahmen des Föderierungssystem

Nachdem in Kapitel 18.5 allgemeine Hinweise zur Nutzung der Kommunikations-Schnittstelle gegeben worden sind, soll in diesem Kapitel speziell auf die Verwendung innerhalb des Föderierungssystems eingegangen werden. Insbesondere werden die Operationsklassen beschrieben, mit denen das Föderierungssystem Daten zwischen einzelnen Komponenten austauscht.

18.6.1 Klassen für den Datenaustausch

In den Abschnitten 18.2.1 und 18.2.2 wurde bereits beschrieben, wie Daten in Objekte verkapselt werden und über Handler bearbeitet können. Während die Operationsklassen in allen Komponenten des Föderierungssystem die selben sind, hat jede Komponente eine eigene Handlerklasse für jede Operationsklasse. In folgenden werden die Operationsklassen und die Handlerklassen des Föderierungskerns beschrieben. Eine Beschreibung der Handlerklassen der Datenbankadapter findet sich bei der Dokumentation der Adapter (Kapitel 21 und Kapitel 22).

Im einzelnen werden die folgenden Klassen dokumentiert:

`CInsert`	Abschnitt 18.6.1, Seite 239
`CUpdate`	Abschnitt 18.6.1, Seite 240
`CDelete`	Abschnitt 18.6.1, Seite 240
`CInsertHandler`	Abschnitt 18.6.1, Seite 240

Die Klasse `CInsert`

Die Operations-Klasse `CInsert` dient dazu, das Einfügen eines Objektes an eine andere Komponente zu melden. Wird in einer lokalen Datenbank ein neues Objekt eingefügt, so meldet der Adapter dies an das Föderierungssystem, indem er eine `CInsert`-Operation verschickt. Das Föderierungssystem reagiert darauf, indem es, falls nötig, `CInsert`-Operationen an andere Datenbanken verschickt, in die dann ebenfalls ein Objekt eingefügt wird.

Die `Id`-Funktion von `CInsert` liefert die Konstante `ID_INSERT`.

`string sSchema`
: Der Name des Komponentenschemas, in das ein Objekt eingefügt werden soll, bzw. eingefügt wurde.

`string sClass`
: Der Name des Types, von dem ein Objekt instanziiert werden soll (die Operation wird von einem Adapter an das Föderierungssystem geschickt), bzw. instanziiert worden ist (die Operation wird vom Föderierungssystem an einen Adapter geschickt).

`string sLoid`
: Die Objekt-ID des einzufügenden bzw. eingefügten Objektes in der Datenbank. Wenn die Operation von einem Adapter an das Föderierungssystem geschickt wird, enthält das Attribut die lokale Objekt-ID des eingefügten Objektes. Andernfalls enthält es eine *default-object-ID*. Der empfangende Adapter muß die richtige Objekt-ID eintragen, nachdem er das Objekt in die Datenbank eingefügt hat.

`bool AddAttribute( const string i_sName, const string i_sValue)`
: Die Methode dient dazu, die Attribute eines Typs (siehe Attribut `sClass`) in die Operation aufzunehmen. Der erste Parameter gibt den Namen des Attributs an, der im Komponentenschema definiert wurde, der zweite bestimmt den Wert, den dieses Attribut erhalten hat.

`bool GetValue( const string i_sName, string* o_sValue )`
: Diese Methode liefert zu einem Attribut (spezifiziert durch den Namen des Attributes) den Wert.

Die Klasse `CUpdate`

Die Operations-Klasse `CUpdate` ist identisch mit `CInsert`. Ihre `Id`-Funktion liefert die Konstante `ID_UPDATE`.

Die Klasse `CDelete`

`CDelete` ist ebenfalls mit `CInsert` identisch. Allerdings fehlen die Methoden `AddAttribute` und `GetValue`, da sich Löschoperationen immer auf ganze Objekte und nicht auf einzelne Attribute beziehen.

Die `Id`-Funktion liefert die Konstante `ID_DELETE`.

Die Klasse `CInsertHandler`

Die Klasse `CInsertHandler` empfängt mit ihrer `Handle`-Methode *Insert-Operationen*, also `CInsert`-Objekte. Ihre Aufgabe besteht darin, die Angaben zu Schemata, Typen und Attributen, die innerhalb eines Operations-Objektes als Text vorliegen, in Zeiger innerhalb *Föderierungsgraphen* (siehe Kapitel 19) zu konvertieren. Anschließend wird der Algorithmus zum Einfügen eines Objektes (siehe Abschnitt 20.2) gestartet. Nach dessen Beendigung liegen die aus der empfangenen *Insert-Operation* resultierenden Operationen vor. Diese werden an die zuständigen Datenbank-Adapter verschickt.

18.6.2 Die Klassen für die System-Administration

Neben den im vorherigen Abschnitt beschriebenen Operationsklassen für den Datenaustausch git es noch eine Reihe weiterer Klassen. Sie dienen administrativen Aufgaben, z.B. einen Datenbank-Adapter beim Föderierungssystem an- und abzumelden oder das gesamte System geordnet herunterzufahren.

Es handelt sich dabei um die folgenden Klassen:

`CAdapterPool`	Abschnitt 18.6.2, Seite 240
`CStandardOp`	Abschnitt 18.6.2, Seite 241
`CShutdownHandler`	Abschnitt 18.6.2, Seite 241
`CShutdownHandler`	Abschnitt 18.6.2, Seite 241
`CAttachHandler`	Abschnitt 18.6.2, Seite 241
`CDetachhandler`	Abschnitt 18.6.2, Seite 241

Die Klasse `CAdapterPool`

Die Klasse `CAdapterPool` speichert alle Datenbankadapter, die sich zur Zeit beim Föderierungssystem angemeldet haben. Sie wird dazu benötigt, um

- beim Versenden einer Datenbank-Operation den für ein Komponentenschema zuständigen Datenbankadapter zu ermitteln und
- beim Herunterfahren des Föderierungssystems alle Datenbankadapter beenden zu können.

`void Insert( const string i_sName )`
: Die Methode registriert einen Adapter mit dem übergebenen Namen. Vor Aufruf der Methode muß überprüft werden, ob bereits ein Adapter unter diesem Namen registriert ist. Dieser muß zuerst gelöscht werden.

`bool Has( const string i_sName )`
: Die Methode dient dazu, festzustellen, ob unter dem übergebenen Namen ein Adapter registriert ist. Falls ja wird `true` geliefert, sonst `false`.

`void Delete( const string i_sName )`
: Mit `Delete` wird ein registrierter Adapter aus dem Pool gelöscht. Wird diese Methode aufgerufen, obwohl unter dem übergebenen Namen kein Adapter registriert ist, so wird der Pool nicht verändert.

`list< string > All()`
: Beim Herunterfahren des Föderierungssystems müssen alle Adapter beendet werden. Mit Hilfe dieser Methode erhält man eine Liste aller Adapter, über die iteriert werden kann (siehe auch Abschnitt 18.6.2).

Die Klasse `CStandardOp`

Die Operationen, die zu administrativen Zwecken zwischen dem Föderierungssystem und den Adaptern ausgetauscht werden, basieren alle auf der Klasse `CStandardOp`. Neben den Standardmethoden für Operationsklassen (siehe Abschnitt 18.4.1), auf die hier nicht näher eingegangen werden soll, verfügt diese Klasse nur über einen Konstruktor und das eine Attribut `string sSender`.

Durch den Konstruktor wird festgelegt, ob das Objekt als *Attach-Operation*, *Detach-Operation* oder *Shutdown-Operation* dienen soll. Zu diesem Zweck wird eine der Konstanten `ID_ATTACH`, `ID_DETACH` oder `ID_SHUTDOWN` übergeben, die alle in der Datei `pg290.H` definiert sind.

Das Attribut `sSender` wird vom Sender der Operation mit seinem Namen gefüllt (siehe dazu auch Abschnitt 23.3.2). Der Empfänger kann dadurch entscheiden, von wo die Operation kommt und entsprechend reagieren.

Die Klasse `CShutdwonHandler` im Föderierungssystem

Das Föderierungssystem erhält die *Shutdown-Operation* in der Regel von dem *Shutdown-Programm* (siehe Abschnitt 23.3.2). Als Reaktion darauf wird das System heruntergefahren. Dazu wird an alle registrierten Adapter (siehe Abschnitt 18.6.2) eine *Shutdown-Operation* verschickt und anschließend das Föderierungssystem selbst heruntergefahren.

Die Klasse `CShutdwonHandler` in den Adaptern

Ein Adapter erhält die *Shutdown-Operation* entweder vom Föderierungssystem oder vom *Shutdown-Programm* (siehe Abschnitt 23.3.2). Im ersten Fall soll das ganze System heruntergefahren werden. Der Adapter beendet sich selbst ohne weitere Operationen zu verschicken. Im zweiten Fall soll nur der Adapter beendet werden, das System ansonsten aber weiterlaufen. Der Adapter schickt eine *Detach-Operation* an das Föderierungssystem und beendet sich selbst.

Die Klasse `CAttachHandler`

Wenn ein Datenbank-Adapter startet, so sendet er eine *Attach-Operation* an das Föderierungssystem. Dieses empfängt die Operation mit `CAttachHandler` und registriert den Adapter (siehe Abschnitt 18.6.2). Anschließend können von diesem Adapter Datenbankoperationen empfangen und an ihn verschickt werden.

Die Klasse `CDetachHandler`

Wenn ein Datenbank-Adapter heruntergefahren wird, sendet er eine *Detach-Operation* an das Föderierungssystem (siehe Abschnitt 18.6.2). Das Föderierungssystem entfernt den Adapter aus der Liste der registrierten Adapter (siehe Abschnitt 18.6.2).

Kapitel 19

Der Föderierungsgraph

Verfasser: Peter Ziesche

19.1 Einleitung

Das Föderierungssystem ist so ausgelegt, daß es eine Föderation unterschiedlicher Datenbanken realisieren kann. Um dies zu erreichen, müssen Informationen über die lokalen Datenbank-Schemata vorliegen und in einer geeigneten Weise vom Föderierungssystem gespeichert werden.

Der Föderierungsgraph ist ein *Meta-Datenmodell*, in dem solche Schema-Beschreibungen abgelegt werden können. Auf diesem Datenmodell arbeiten die Algorithmen des Föderierungssystems (siehe Abschnitt 20.2) und stellen die für eine Föderation benötigte Funktionalität zur Verfügung.

19.2 Die Schema-Architektur

Die Föderierung lokaler Datenbanken erfolgt gemäß der Sheth-Larson-Referenzarchitektur [SL90]. Für jede an der Föderation beteiligte lokale Datenbank wird ein Komponentenschema (CS) angelegt, in dem eine Beschreibung des lokalen Datenbankschemas (LS) gespeichert wird. Dieser Beschreibung liegt das gewählte *kanonische Datenmodell*, in unserem Fall das objektorientierte Datenmodell, zugrunde.

Für ein Komponenten-Schema können nun Import- und Export-Schemata (IS bzw. ES) angelegt werden. Sie bestimmen, welche Teile der lokalen Datenbank föderiert werden sollen.

Das föderierte Schema (FS) schließlich speichert die Abhängigkeiten zwischen den Import- und Exportschemata der lokalen Datenbanken, d.h. Informationen darüber, wohin Änderungen in einer lokalen Datenbank weitergegeben werden müssen.

Die Sheth-Larson-Referenzarchitektur sieht darüber hinaus noch ein externes Schema vor, über das Teile des föderierten Schemas von außen zugänglich sind. Diese Schicht wird von unserer Anwendung nicht unterstützt. Das Meta-Datenmodell des Graphen würde die Definition eines externen Schemas jedoch zulassen, so daß eine spätere Erweiterung leicht möglich ist.

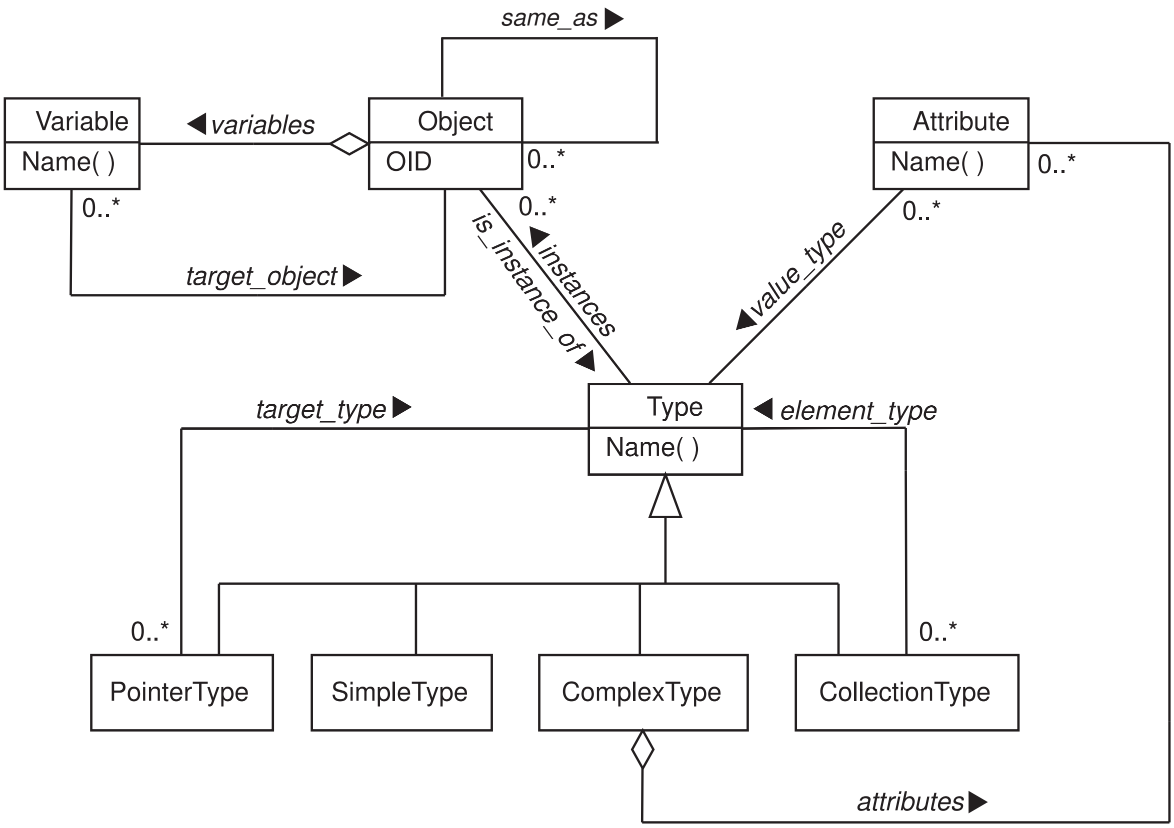


Abbildung 19.1: Die Typhierarchie zur Beschreibung der Datenbank-Schemata

19.3 Das Meta-Datenmodell zur Beschreibung von Datenbank-Schemata

19.3.1 Die Typhierarchie

Innerhalb des Föderierungssystems müssen Datenbank-Schemata beschrieben werden. Es wird also ein (Meta-)Datenmodell benötigt, in dem eine solche Beschreibung ausgedrückt werden kann. Das von uns gewählte Datenmodell lehnt sich eng an den ODMG-93-Standard [Cat96] an. Abbildung 19.1 zeigt die Hauptbestandteile unseres Datenmodells.

An dieser Stelle ist es wichtig, sich noch einmal genau klar zu machen, daß die Typhierarchie ein *Meta-Datenmodell* ist. Die abgebildeten Klassen[1] beschreiben bestimmter Arten von Klassen (z.B. Containerklassen). Jede Instanz einer solchen Klasse beschreibt eine konkrete Klasse in einem Datenbank-Schema. Es handelt sich also gewissermaßen um *Metaklassen*.

Die einzelnen Klassen haben folgende Bedeutung:

`Type`
: Dies ist die Basisklasse für Typen. Sie dient zur Verwaltung von Objekten (siehe Abschnitt 19.3.2), stellt also die dazu benötigten Routinen zur Verfügung. Die Klasse ist abstrakt, d.h. es gibt in einem Datenbank-Schema keine Klassen vom Type "`Type`".

SimpleType

[1] gemäß dem ODMG-93 Standard bezeichnen wir Klassen als Typen

Instanzen von `SimpleType` sind einfache Typen wie sie von Programmiersprachen als eingebaute Typen zur Verfügung gestellt werden. Der ODMG-93-Standard definiert die folgenden Typen:

- `integer`
- `long`
- `boolean`
- `real`
- `double`
- `character`
- `string`
- `date`
- `time`

ComplexType
Mit der Klasse ComplexType werden Klassen modelliert, die aus mehreren Komponenten (siehe Abschnitt 19.4.1) zusammengesetzt sein können. Dadurch lassen sich beliebig tiefe, ineinander geschachtelte Hierarchien aufbauen. Dieses Konstrukt entspricht einer Klasse, wie sie die meisten objektorientierten Programmiersprachen zur Verfügung stellen.

CollectionType
Hierbei handelt es sich um einen Typ zur Modellierung von Containerklassen. Eine Unterscheidung mehrerer Arten von Containerklassen, z.B. Listen, Mengen, Multimengen usw. ist innerhalb des Föderierungssystem nicht erforderlich. Eine mit CollectionType modellierte Containerklasse ist getypt, d.h. die in einem Container gespeicherten Elemente müssen einen bestimmten Typ haben, der Bestandteil desselben Schemas wie der Container selbst sein muß.

PointerType
Mit PointerType können Referenzen (Zeiger) auf andere Typen modelliert werden.

Attribute
Attribute sind Bestandteile (Komponenten) von Klassen. Ein Attribut gehört also immer zu einem bestimmten `ComplexType`. Darüber hinaus hat ein Attribut einen Typ, der jedoch im selben Schema wie das Attribut definiert sein muß.

19.3.2 Objekte

Ein Problem beim Betrieb des Föderierungs-Systems ist, daß auf Objekte in lokalen Datenbanken zugegriffen werden muß. Da konzeptionell nur objektorientierte Datenbanken unterstützt werden[2] können alle Objekte durch ihre *Objekt-Id* identifiziert werden. Für jedes föderierte Objekt muß diese lokale *Objekt-Id* gespeichert werden.

Objekte in verschiedenen lokalen Datenbanken können in einer *same_as*-Beziehung zueinander stehen. Dies bedeutet, daß sie (eventuell unterschiedliche Aspekte) das *selbe* Objektes in der realen Welt modellieren. Diese *same_as*-Beziehung muß im Föderierungsgraphen gespeichert werden. Änderungen an Attributen eines Objektes können dann an alle Objekte weitergegeben werden, mit denen es in einer *same_as*-Beziehung steht (vorausgesetzt diese Objekte haben ebenfalls das veränderte Attribute).

Im Föderierungsgraphen gibt es daher *Proxy-Objekte*, mit denen genau diese beiden Aspekte (lokale Objekt-Id und *same_as*-Beziehung) modelliert werden können. Ein *Proxy-Objekt* in einem Komponentenschema speichert die Objekt-Id des Objektes in der lokalen Datenbank. Es ist verknüpft mit einem *Proxy-Objekt* im föderierten Schema. Von diesem können die *Proxy-Objekte* in anderen Komponenten-Schemata erreicht werden, wodurch die *same_as*-Beziehung realisiert ist.

[2] Für andere Datenbanken muß durch einen Adapter eine Konvertierung zwischen lokalem und kanonischem Datenmodell vorgenommen werden.

19.4 Implementierung

Die einzelnen Klassen des Föderierungsgraphen haben zum Teil eine sehr ähnliche Funktionalität. Diese lassen sich in drei Gruppen zusammenfassen, die im folgenden ausführlich beschrieben werden. Dadurch kann bei den einzelnen Klassen auf eine Beschreibung jeder einzelnen Methode verzichtet werden, was die Klassenbeschreibungen übersichtlicher macht.

19.4.1 Verwaltung von Komponenten

Der Föderierungsgraph ist hierarchisch aufgebaut. Der Graph enthält Schemata, Schemata enthalten Typen, Typen enthalten Objekte, usw. Um diese Komponenten effizient verwalten zu können, wird die Template-Klasse `CComponents` benutzt. `CComponents` verwaltet Komponenten eines bestimmten Typs und stellt Methoden für das Einfügen und Auslesen von Komponenten zur Verfügung. Jede Klasse, die Komponenten verwalten muß, stellt Methoden in ihrer Schnittstelle zur Verfügung, die Aufrufe dann lediglich an die entsprechenden Methoden von `CComponents` delegieren müssen.

Jede Klasse benötigt die folgenden Methoden (Das `X` steht jeweils für den Typ der Komponente):

`void AddX( PX poNewX )`
: Die übergebene Komponente wird neu hinzugefügt. Der Name der Komponente muß eindeutig sein, d.h. es darf noch keine Komponente mit dem gleichen Namen geben.

`bool HasX( const string i_sName )`
: Es wird getestet, ob unter den Komponenten eine mit dem übergebenen Namen ist.

`PX GetX( const string i_sName )`
: Es wird ein Zeiger auf die Komponente mit dem übergebenen Namen geliefert. Es muß eine Komponente mit dem Namen existieren.

`list< PX > GetAllX()`
: Die Methode liefert eine Liste mit Zeigern auf alle Komponenten. Die Komponenten selbst werden nicht kopiert, d.h. über die Zeiger in der Liste gemachte Änderungen wirken sich auf die Komponenten aus. Das Einfügen oder Löschen von Listenelementen hat dagegen keinen Einfluß auf die Komponenten.

`int NumberOfX()`
: Die Methode liefert die Anzahl der Komponenten.[3]

19.4.2 Erzeugung (Konstruktoren)

Bei der Erzeugung einer Instanz soll dem Benutzer so viel Arbeit wie möglich abgenommen werden. Der Konstruktor einer Komponente erhält einen Pointer auf das Aggregat[4] und ruft sofort die `AddX`-Methode des Aggregates auf, registriert sich also selbständig. Dieses Verfahren stellt sicher, daß Komponenten nicht verloren gehen. Sollte es einmal erforderlich sein, eine Komponente ohne automatische Registrierung zu erzeugen, kann der Parameter `fRegister` mit `false` belegt werden.

19.4.3 Verwaltung von Verkettungen

Einige Objekte im Graphen (Schemata, Typen, Attribute und Objekte) sind mit Objekten der gleichen Klasse auf anderen Schema-Ebenen verknüpft. Diese Verknüpfungen werden ebenfalls durch eine Templateklasse verwaltet. Die Klasse `CLayer` stellt Methoden zur Verfügung, an die andere Klassen einfach delegieren können.

Im einzelnen sind dies die Methoden (das `X` steht wieder für den jeweiligen Typ):

[3] In der Klasse `CComponents` heißt diese Methode `Size()`.

[4] Das Aggregat ist das Objekt, das die Komponente beinhalten soll

`list< PX > Pre() const`
: Es wird eine Liste mit allen Objekten geliefert, die Vorgänger dieses Objektes sind. Wird die Methode für ein Objekt auf Exportschema-Ebene aufgerufen, so zeigen die Listenelemente auf Objekte in der Komponentenschema-Ebene.

`list< PX > Post() const`
: Es wird eine Liste mit allen Objekten geliefert, die Nachfolger dieses Objektes sind. Wird die Methode für ein Objekt auf Exportschema-Ebene aufgerufen, so zeigen die Listenelemente auf Objekte in der Föderierungsschema-Ebene.

  Auch bei diesen beiden Methoden handelt es sich bei den zurückgelieferten Listen um Kopien der Vorgänger- bzw. Nachfolger-Listen. Die Elemente zeigen jedoch auf die echten Objekte.

`void AddPreX( PX poPreX )`
: Die Methode fügt das übergebene Objekt in die Liste der Vorgänger ein.

`void AddPostX( PX poPostX )`
: Die Methode fügt das übergebene Objekt in die Liste der Nachfolger ein.

19.4.4 Die Klassen des Föderierungsgraphen

`CGraph`	Abschnitt 19.4.4, Seite 246
`CSchema`	Abschnitt 19.4.4, Seite 246
`CType`	Abschnitt 19.4.4, Seite 247
`CSimpleType`	Abschnitt 19.4.4, Seite 247
`CComplexType`	Abschnitt 19.4.4, Seite 248
`CCollectionType`	Abschnitt 19.4.4, Seite 248
`CCPointerType`	Abschnitt 19.4.4, Seite 248
`CAttribute`	Abschnitt 19.4.4, Seite 249
`CObject`	Abschnitt 19.4.4, Seite 249
`CVariable`	Abschnitt 19.4.4, Seite 250
`CLayer`	Abschnitt 19.4.4, Seite 250
`CComponents`	Abschnitt 19.4.4, Seite 250

Eine Übersicht über die Klassen des Föderierungsgraphen gibt Abbildung 19.1.

Die Klasse `CGraph`

Diese Klasse bildet die Wurzel des Föderierungsgraphen. Von ihr braucht normalerweise nur eine einzige Instanz zu existieren.

```
// Komponente
   void AddSchema( PCSchema poNewSchema );
   bool HasSchema( const string i_sName ) const;
   PCSchema GetSchema( const string i_sName ) const;
   list< PCSchema > GetAllSchemata() const;
   int NumberOfSchemata() const;
```

Die Klasse `CSchema`

Alle Schemata des Föderierungsgraphen sind Instanzen dieser Klasse. Es gibt also keine strukturelle Unterscheidung zwischen Komponenten-, Import/Export- und Föderierten-Schemata. Der Name des Schemas muß eindeutig innerhalb des gesamten Föderierungsgraphen sein.

```
// Konstruktore
   CSchema( const string i_sName,
            PCGraph poGraph,
            bool fRegister = true );

// Identifizierung
   string Name() const;

// Komponenten
   void AddType( PCType poNewType );
   bool HasType( const string i_sName ) const;
   PCType GetType( const string i_sName ) const;
   list< PCType > GetAllTypes() const;
   int NumberOfTypes() const;

// Verkettung
   list< PCSchema > Pre() const;
   list< PCSchema > Post() const;
   void AddPreSchema( PCSchema poPreSchema );
   void AddPostSchema( PCSchema poPostSchema );
```

Die Klasse `CType`

Dies ist die abstrakte Basisklasse für Typen. Sie dient der Verwaltung von CObject-Objekten, die Instanzen des `CType`s darstellen.

Die Methode `TypeId()` liefert die Art des Typs, also `CComplexType`, `CPointerType`, usw. Sie kann dann benutzt werden, wenn nur eine Referenz vom Type `CType` zur Verfügung steht, die genaue Typ-Art jedoch ermittelt werden muß (*downcasting*).

```
// Identifizierung
   virtual string Name() const = 0;
   PCSchema Schema() const;
   virtual const int TypeId() const = 0;

// Komponenten
   void AddObject( PCObject poNewObject );
   bool HasObject( const string i_sOid ) const;
   PCObject GetObject( const string i_sOid ) const;
   list< PCObject > GetAllObjects() const;
   int NumberOfObjects() const;

// Verkettung
   list< PCType > Pre() const;
   list< PCType > Post() const;
   void AddPreType( PCType poPreType );
   void AddPostType( PCType poPostType );
```

Die Klasse `CSimpleType`

Diese Klase implementiert die in Abschnitt 19.3.1 beschriebenen einfachen Typen. Der Name ist einer der dort genannten Typ-Arten. Die Methode `TypeId` liefert die Konstante `ID_SIMPLETYPE`.

```
// Konstruktore
```

```
   CSimpleType( const string i_sName,
                PCSchema poSchema,
                bool fRegister = true );

// Identifizierung
   virtual string Name() const;
   virtual const int TypeId() const;
```

Die Klasse `CComplexType`

Diese Klase implementiert die in Abschnitt 19.3.1 beschriebene Typ-Art *ComplexType*. Die Methode `TypeId` liefert die Konstante `ID_COMPLEXYPE`.

```
// Konstruktor
   CComplexType( const string i_sName,
                 PCSchema poSchema,
                 bool fRegister = true );

// Identifizierung
   virtual string Name() const;
   virtual const int TypeId() const;

// Komponenten
   void AddAttribute( PCAttribute poNewAttribute );
   bool HasAttribute( const string i_sName ) const;
   PCAttribute GetAttribute( const string i_sName ) const;
   list< PCAttribute > GetAllAttributes() const;
   int NumberOfAttributes() const;
```

Die Klasse `CCollectionType`

Diese Klase implementiert die in Abschnitt 19.3.1 beschriebene Typ-Art *CollectionType*. Die Methode `TypeId` liefert die Konstante `ID_COLLECTIONTYPE`. Die Methode `ElementType` liefert einen Zeiger auf den Typ der Elemente. Dieser muß im gleichen Schema existieren. Der Name setzt sich zusammen aus dem Namen des Element-Typs, gefolgt von dem Wort "`List`".

```
// Konstruktor
   CCollectionType( PCSchema poSchema,
                    PCType poElementType,
                    bool fRegister = true );

// Identifizierung
   virtual string Name() const;
   virtual const int TypeId() const;
   PCType ElementType() const;
```

Die Klasse `CPointerType`

Diese Klase implementiert die in Abschnitt 19.3.1 beschriebene Typ-Art *PointerType*. Die Methode `TypeId` liefert die Konstante `ID_POINTERTYPE`. Die Methode `TargetType` liefert einen Zeiger auf den referenzierten Typ. Dieser muß im gleichen Schema existieren. Der Name setzt sich zusammen aus dem Namen des referenzierten Typs, gefolgt von "`*`".

```
// Konstruktor
   CPointerType( PCSchema poSchema,
                 PCType poTargetType,
                 bool fRegister = true );

// Identifizierung
   virtual string Name() const;
   virtual const int TypeId() const;
   PCType TargetType() const;
```

Die Klasse `CAttribute`

Diese Klasse implementiert Attribute, also Komponenten eines `CComplexType`-Objektes. Die Methode `IsComponentOf` liefert einen Zeiger auf dieses Objekt. Mit `Type` kann der Typ des Attributes abgefragt werden.

Die im ODMG-93-Standard beschriebene `invers`-Relation kann durch die Methoden `GetInversAttribute` bzw. `SetInversAttribute` ausgelesen, bzw. gesetzt werden. Der Aufruf von `SetInversAttribute` bewirkt automatisch den Aufruf dieser Routine in dem übergebenen Objekt, so daß die `invers`-Beziehung stehts konsistent ist.

```
// Konstruktor
   CAttribute (const string i_sName,
               PCType poType,
               PCComplexType poClass,
               bool fRegister = true );

// Identifizierung
   string Name() const;
   PCType Type() const;
   PCComplexType IsComponentOf() const;
   PCSchema Schema() const;
   PCAttribute GetInversAttr();
   void SetInversAttr( PCAttribute poInvers, bool fRegister = true );

// Verkettung
   list< PCAttribute > Pre() const;
   list< PCAttribute > Post() const;
   void AddPreAttribute( PCAttribute poPreAttribute );
   void AddPostAttribute( PCAttribute poPostAttribute );
```

Die Klasse `CObject`

Diese Klasse dient zur Verwaltung lokaler Objekte in lokalen Datenbanken (siehe Abschnitt 19.3.2). Die Methode `Name` dient nur einer einheitlichen Schnittstelle, `Name` und `Oid` liefern stets das gleiche Ergebnis. Die Methode `IsInstanceOf` liefert einen Zeiger auf den Typ des `CObject`-Objetes. Mit `GenerateOid` kann eine eindeutige Objekt-Id erzeugt werden.

```
// Konstruktor
   CObject ( PCType poClass, bool fRegister = true );

// Identifizierung
   static string GenerateOid();
   string Oid() const;
```

```
   string Name() const;
   void SetOid( const string i_sNewOid );
   PCType IsInstanceOf() const;

// Verkettung
   list< PCObject > Pre() const;
   list< PCObject > Post() const;
   void AddPreObject( PCObject poPreObject );
   void AddPostObject( PCObject poPostObject );

// Komponenten
   void AddVariable( PCVariable poNewVar );
   bool HasVariable( const string i_sName ) const;
   PCVariable GetVariable( const string i_sName ) const;
   list< PCVariable > GetAllVariables() const;
   int NumberOfVariables() const;
```

Die Klasse `CVariable`

Während `CAttribute` Attribute auf Datenbankschema-Ebene, also statisch beschreibt, dient `CVariable` dazu, Attribute von `CObject`-Instanzen zu beschreiben. Dies sind die eigentlichen Daten in einer Datenbank. Da im Föderierungsgraphen keine Daten gespeichert werden, wird `CVariable` nur benötigt, um Zeiger auf andere Objekte zu speichern. Dies kann notwendig sein, um den Föderierungsgraphen bei Änderungsoperationen konsistent halten zu können.

```
// Konstruktor
   CVariable( const string i_sName,
              PCObject poOwnerObject,
              bool fRegister = true );

// Identifizierung
   string Name() const;
   PCObject Value() const;
   void SetValue( PCObject poNewValue );
```

Die Klasse `CLayer`

Diese Template-Klasse wurde in Abschnitt 19.4.3 schon genau beschrieben. Sie wird von `CSchema`, `CType`, `CAttribute` und `CObject` benutzt um die Verknüpfungen zwischen Schema-Ebenen zu verwalten.

```
// Verkettung
   list< T* > Pre()  const;
   list< T* > Post() const;
   void AddPre( T* poElement );
   void AddPost( T* poElement );
```

Die Klasse `CComponents`

Diese Template-Klasse wurde in Abschnitt 19.4.1 schon genau beschrieben. Sie wird von `CSchema`, `CType`, `CComplexType` und `CObject` benutzt.

```
// Komponenten
   void Add( T* poNew );
   bool Has( const string i_sName ) const;
   T* Get( const string i_sName ) const;
   list< T* > All() const;
   int Size() const;
```

Kapitel 20

Der Kern des FDBS

Neben dem informationsverarbeitenden Kern wird noch in zwei andere große Teile unterschieden: den Basisalgorithmen und den Spezialalgorithmen, die jeweils in einer eigenen Klasse zusammengefaßt sind.

20.1 Die Basisalgorithmen

Verfasser: Dilber Yavuz, Djamel Kheldoun

20.1.1 Entwurf

Für die Klasse `CBaseAlgorithms` dient der Graph als Grundlage. Als Eingabe erhält `CBaseAlgorithms` das Komponentenschema und einen Typ. Als Ausgabe liefert er ein Objekt `oCWorkGraph` der Klasse `CWorkGraph`. Folgende Klassen werden benutzt:
Die Klasse `CTupel` besteht aus zwei Attributen Schema und Type, die auf `CSchema` und `CType` zeigen.

```
class CTupel
{
// Initialisierung
public:
   CTupel()
     :Schema( NULL ),
     Type( NULL )
   {
      //empty
   };

//methods
public:
   CSchema* Schema;
   CType*   Type;

};
```

Die Klasse `CBranch` besteht aus zwei Attributen ExportLayer und ComponentLayer.

```
class CBranch
{
//methods
public:
   CTupel ExportLayer;
   CTupel ComponentLayer;

};
```

Die Klasse `CWorkGraph` besteht aus den Attributen ExportLayer, FederatedLayer und aus einer Liste von "Branches".

```
class CWorkGraph
{
//methods
public:
   CTupel ExportLayer;
   CTupel FederatedLayer;

   list< CBranch > aOutBranches;
};
```

Die Klasse `CBaseAlgorithms` hat eine Methode `GetWorkGraph`.

```
class CBaseAlgorithms
{
//methods
public:
   CWorkGraph GetWorkGraph( CSchema InSchema, CType InType );

};
```

Der Graph soll vom Komponentenschema aus mit `Post()` bis zur Föderierungs-Ebene und von dort aus mit `Pre()` bis zu den entsprechenden Komponentenschemata durchlaufen werden.

20.1.2 Implementierung

Ziel des Graph-Durchlaufs ist, zu einem gegebenen Schema und Typ die entsprechenden Schemata und Typen zu finden. Dazu werden für jedes Schema und jeden Typ Listen initialisiert, die alle Nachfolger bzw. Vorgänger der entsprechenden Schemata enthalten, wobei in unserem Fall die Listen nur ein Element enthalten.

Die Liste `aPostCompSchema`, die alle Nachfolger von einem Component-Schema enthält, und die Liste `aPostCompType`, die alle Nachfolger von einem Component-Type enthält, wird mit dem Ergebnis des Aufrufes von `Post()` initialisiert. Das gefundene Schema und Typ der Export-Ebene wird als Komponente in den `CWorkGraph` aufgenommen. Analog werden die entsprechenden Listen für die Export-Schemata und den Export-Type initialisiert. Das gefundene Schema und Typ der Föderierungs-Ebene wird wieder in den `CWorkGraph` aufgenommen.

Von der Föderierungs-Ebene aus sollen alle Wege zu den entsprechenden Komponenten-Schemata durchlaufen werden. Die Liste `aPreFedSchema`, die alle Vorgänger von einem Föderierungs-Schema (Export-Schemata) enthält, und die Liste `aPreFedType`, die alle Vorgänger von einem Federated-Type (Export-Type) enthält, wird mit dem Ergebnis des Aufrufes von `Pre()` initialisiert. Alle Export-Schemata außer dem Export-Schema, von dem die Daten exportiert wurden, werden betrachtet. Dann werden die "Branches", die die Schemata und Typen der Export-Ebene und der Komponenten-Ebene enthalten in den `CWorkGraph` eingefügt.

20.1.3 Testen

Das Programm `CBaseAlgorithms` wird mit dem Testprogramm `TestBaseAlgorihthms` wie folgt getestet:

Aufruf: Der Aufruf von `Buildgraph()` erstellt den Graphen mit den Komponenten-Schemata, Export-Schemata und dem Föderierungs-Schema.

Testdaten: Die entsprechenden Komponenten-Schemata und Komponenten-Typen.

`GetWorkGraph` aufrufen: Die Funktion `GetWorkGraph` soll dann mit den Testdaten aufgerufen werden.

20.2 Die Spezialalgorithmen

Verfasser: Patrick Koehne, Andreas Dinsch

20.2.1 Die Analyse

Die Klasse der Spezialalgorithmen ist dafür zuständig, für eine vom Datenbankadapter eingehende Nachricht eine Menge von entsprechenden Informationen aus dem Föderierungsgraphen zu sammeln, die für die Generierung der Aufrufe der zu föderierenden Datenbanken dienen. Des weiteren muß von den Methoden der Föderierungsgraph gepflegt werden. Was dies im einzelnen bedeutet, wird bei den entsprechenden Methoden genauer erklärt.

20.2.2 Der Entwurf

Die Methoden

Die Klasse `CSpecialAlgorithms` enthält zunächst die folgenden öffentlichen Methoden:

```
TUpdateList* Update( PCSchema, PCComplexType, PCObject, TAttributeList* )
TInsertList* Insert( PCSchema, PCComplexType, PCObject, TAttributeList*, PCObject )
TDeleteList* Delete( PCObject* )
```

Die Methoden `Update` und `Insert` erhalten als Übergabeparameter jeweils die Informationen, die in überarbeiteter Version vom Datenbankadapter geliefert werden, d.h. als Pointer auf das Schema, den gewünschten Typ und das zu ändernde oder einzufügende Objekt. Dazu kommt noch eine Liste von Attributen mit den dazugehörenden Attributinhalten. Die Methode `Delete` erhält lediglich das zu löschende Objekt als Übergabe. Mit Hilfe dieser Informationen suchen die Methoden diejenigen Objekte zusammen, die in den angeschlossenen Datenbanken geändert bzw. eingefügt werden müssen. Sie liefern eine Liste von Objekten zurück, die alle Informationen enthalten, die notwendig sind, um einen entsprechenden Verarbeitungsbefehl an den Datenbankadapter zu generieren. Hierzu werden später die bereits vorhandenen Klassen `CUpdate`, `CInsert` und `CDelete` verwendet, von denen Objekte instanziiert und mit Informationen gefüllt werden.

Die Methode `Insert` bekommt, im Gegensatz zu `Update`, zusätzlich noch einen Pointer auf ein weiteres Objekt vom Typ `PCObject`. Dies scheint zunächst verwunderlich, kann aber für die Föderierung notwendig sein, wenn relationale Datenbanken angebunden sind. Dabei kann es nämlich vorkommen, daß eine 1:1-Abbildung zwischen Objekten nicht ausreicht, weil in relationalen Datenbanken Beziehungen zwischen Schlüsseln verschiedener Tabellen erzeugt werden müssen. Der Einfachheit halber wird diese Feststellung von einem der Adapter übernommen. Wenn es also erforderlich sein sollte, daß eine angeschlossene Datenbank zusätzliche Informationen braucht, so liefert der Adapter diese gleich in Form dieses Pointers mit. Der Föderierungskern kümmert sich also auch nur im Falle eines Pointers ungleich dem Nullpointer um deren Verarbeitung. In einer späteren Version des Graphen, sollen solche Informationen auch direkt im Föderierungsgraphen gehalten werden.

Die Methode `Insert` sorgt auch daür, daß die für den Graph notwendigen Objekte vom Typ `CObject` angelegt und in den Graphen eingebettet werden. Einbetten bedeutet in diesem Falle, daß ein zugehöriges föderiertes `CObject` angelegt wird und dieses bidirektional mit dem übergebenen Objekt verbunden wird. Ebenso wird für jedes Insertobjekt vom Typ `CInsert`, welches von `Insert` erzeugt wird, ein entsprechendes `CObject` erzeugt und ebenfalls mit dem föderierten Objekt bidirektional verbunden.

Bei der Methode `Update` ist ein Pflegen der `CObject`-Objekte nicht notwendig, da diese bereits existieren und sich auch nicht wieder ändern sollten. Die Methode `Delete` ist z.Z. noch nicht in der Lage, die entsprechenden Objekte aus dem Graphen wieder zu löschen, da die Klasse `CType`, die vom Graphen geliefert wird, noch keine Methode vorsieht, um die Objekte aus seiner Liste zu entfernen. Es werden also nur Delete-Befehle an die Datenbanken abgeschickt und der Graph nicht reduziert.

Um die gewünschten Informationen sammeln zu können, stützen sich die öffentlichen Methoden hauptsächlich auf zwei private Methoden ab:

TEqAttributeList* GetEqAttributes(PCAttribute): Die Methode `getEqAttributes` liefert zu einem gewünschten Attribut eine Liste von äquivalenten Attributen zurück, die in den anderen Datenbanken von einer Änderung betroffen sind.

TEqObjectList* getEqObjects(PCObject): Die Methode `getEqObjects` liefert zu einem gewünschten Objekt genau die Liste von Objekten zurück, die von der Änderung betroffen sind.

Aus genau diesen beiden Listen können dann die notwendigen Informationen zusammengesammelt werden, die für die zurückzuliefernde Objektliste gebraucht werden.

Die Datenstrukturen

Bei den Übergabeparametern dürften die Pointer auf `CSchema`, `CType` und `CObject` eindeutig sein. Die zu übergebene Attributliste besteht aus Tupeln von Attributpointern und Attributwerten als String:

```
struct TAttribute {
   PCAttribute Attribute;
   string      Value;
   }
typedef list< TAttribute > TAttributeList;
```

Die zurückgelieferten Listen von `Update`, `Insert` und `Delete` bestehen aus Listen von entsprechenden Objekten:

```
typedef list< CUpdate > TUpdateList;
typedef list< TInsertObjectPair > TInsertList;
typedef list< CDelete > TDeleteList;
```

Eine Ausnahme bildet hierbei die Rückgabeliste der Methode `Insert`. Hier wird neben dem eigentlichen Objekt vom Typ `CInsert` noch ein Pointer auf ein neues Objekt vom Typ `CObject` zurückgeliefert.

```
struct TInsertObjectPair {
    CInsert     first;
    PCObject    second;
  };
```

Dieses Objekt dient im Graphen dazu, die OIDs der lokalen Datenbanken zu speichern und zu verwalten. Da bei einer Insertoperation in einer der angeschlossenen Datenbanken eine neue OID natürlich erst nach einem realen Einfügen in die Datenbank bekannt sein kann, aber das Objekt bereits an dieser Stelle in den Föderierungsgraphen eingefügt und entsprechend verknüpft wird, ist es notwendig, in dem Objekt eine Dummy-OID zu speichern und diese nachträglich zu ändern. Um dies erledigen zu können, wird zusätzlich zu dem eigentlichen `CInsert`-Objekt noch der Pointer auf eben jenes "OID-Objekt" zurückgeliefert. Wenn später vom Datenbankadapter die OID geliefert wird, kann der Operationhandler auf Grund des gelieferten Pointers sofort die richtige OID in das Objekt eintragen.

Die von den privaten Methoden verwendete Datenstruktur ist eine Liste von Listen. Die Hauptliste enthält die vom Aufrufenden übergebenen Originalinformationen, d.h. z.B. den Pointer auf das Attribut und den Inhalt als String. Jedes Element dieser Liste hat einen Pointer auf eine zweite Liste, die aus Tupeln besteht, welche Pointer auf äquivalente Attribute in anderen Schemata und deren zugehörigen Typen beinhaltet. Die Pointer auf die Typen werden aus Gründen der einfacheren Suche an dieser Stelle gehalten.

```
struct TEqAttribute {
  PCAttribute Attribute;
  PCType      Type;
  }
typedef list< TEqAttribute > TEqAttributeList;

struct TOrgAttribute {
  PCAttribute       Attribute;
  string            Value;
  TEqAttributeList* Synonyms;
  }
typedef list< TOrgAttribute > TOrgAttributeList;
```

Bei der Liste `TEqObjectList` ist eine einfachere Struktur ausreichend. Da bei jedem Aufruf von `Update`, `Insert` oder `Delete` jeweils nur ein Objektpointer übergeben wird, reicht es aus, eine direkte Liste von entsprechenden äquivalenten Objektpointern und deren zugehörigen Typen zurückzuliefern. Eine Liste von 'Original-Objekten', wie bei den Attributen (`TOrgAttributeList`), wird somit nicht benötigt.

```
struct TEqObject{
  PCObject Object;
  PCType   Type;
  }
typedef list< TEqObject > TEqObjectList;
```

Der erweiterte Entwurf

In einer zweiten Implementierung ist nun das Problem aufgegriffen, daß es in bestimmten Fällen notwendig sein kann, bei der Methode `Insert` den zusätzlichen Pointer vom Typ `PCObject` zu übergeben. Da der Entwurf des Graphen eine unabhängige Bearbeitung dieses Falles vorsieht, soll nun Abstand davon genommen werden, daß der Adapter die notwendige Entscheidung übernehmen muß.

Die Schnittstelle wird aus Gründen der Kompatibilität beibehalten und der übergebene Extra-Pointer ist immer der Nullpointer. Die Struktur des Graphen macht es möglich, den Typ eines Attributes abzufragen. Für gewöhnlich sind dies Typen wie *string*, *int* oder ähnliches. Falls eine solche Abfrage nun auf den Typen *PointerType* stößt, ist der Fall eingetreten, daß zusätzliche Informationen benötigt werden. Von hier ist es möglich, mit der Methode `TargetType()` des Objektes einen Pointer auf die Klasse zu bekommen, die zu dem Attributwert gehört. Mit Hilfe dieses Pointers und dem Attributwert ist man nun in der Lage, auf herkömmliche Art und Weise (`GetEqObjects`, `GetEqAttributes`) den richtigen Attributnamen auf der zu föderierenden Seite des Graphen zu finden. Dieser Attributname wird dann mit dem übergebenen Attributwert als normales Attribut-Wert-Paar in die Insertoperation eingebaut.

Die genauen Erläuterungen zum Aufbau des Graphen und die somit entstandenen Suchmöglichkeiten sind im Kapitel 19.3 ausführlich erläutert.

20.2.3 Die Implementierung

Die zunächst wichtigsten Methoden der ganzen Klassen sind die Methoden `GetEqObjects` und `GetEqAttributes`:

```
TEqObjectList* CSpecialAlgorithms::getEqObjects( PCObject poOrgObject )

  TEqObjectList*  paErgebnisListe = new( TEqObjectList );
  // Erzeugt Liste fuer das Ergebnis
  list <PCObject> aFedObjects; // Liste von Objectpointern im FedSchema
  list <PCObject> aEqObjects;  // Liste aller equivalenten Objektpointern, geliefert von pre
  list <PCObject>::iterator iFedObjects; // Iterator auf eine Liste mit Objektzeigern
  list <PCObject>::iterator iEqObjects; // Iterator auf Objektpointerliste
  TEqObject oObjectTupel; // definiert einen ObjectTupel (PCObject,PCType) fuer die Ruechgabeliste

  aFedObjects = poOrgObject->Post(); // Holt die Liste von Objektpointern
  iFedObjects = aFedObjects.begin(); // setze Iterator auf erstes Listenelement
  aEqObjects = (*iFedObjects)->Pre(); // holt die Liste der equivalenten Objektpointern

  for( iEqObjects = aEqObjects.begin();
       iEqObjects != aEqObjects.end();
       iEqObjects++ )
    if( *iEqObjects != poOrgObject ) {
      oObjectTupel.Object = *iEqObjects;
      oObjectTupel.Type = (*iEqObjects)->IsInstanceOf();
      paErgebnisListe->push_back( oObjectTupel );
    }
  return( paErgebnisListe );
};


TEqAttributeList* CSpecialAlgorithms::getEqAttributes( PCAttribute poOrgCSAttribute )
{
  TEqAttributeList* paFedAttributeList = new( TEqAttributeList );

  // Diese Listen sind Listen von Zeigern auf CAttribute, somit ist ein Iterator
  // auf diesen Listen ein Zeiger auf einen Zeiger auf CAttribute.
  list< PCAttribute > apListeDerOrgESAttribute;
  list< PCAttribute > apListeDerOrgFSAttribute;
  list< PCAttribute > apListeDerFedESAttribute;
  list< PCAttribute > apListeDerFedCSAttribute;

  // Iteratoren
  list< PCAttribute >::iterator iOrgESAttribute;
  list< PCAttribute >::iterator iOrgFSAttribute;
  list< PCAttribute >::iterator iFedESAttribute;
  list< PCAttribute >::iterator iFedCSAttribute;

  PCAttribute poOrgESAttribute;

  TEqAttribute oEqAttribute;
```

```
// Zuerst im Graph aufwaerts bis zur FS-Ebene wandern

// Zum Original-CS-Attribut gibt es eine Liste von Original-ES-Attributen
// Diese Liste enthaelt zur Zeit jedoch nur ein Element,
// da nur 1:1-Abbildung von CS und ES bestehen.

apListeDerOrgESAttribute = poOrgCSAttribute->Post();

// Schleife zum Durchlaufen aller Original-ES-Attribute
for( iOrgESAttribute = apListeDerOrgESAttribute.begin();
     iOrgESAttribute != apListeDerOrgESAttribute.end();
     iOrgESAttribute++ ) {

  // Das Original-ES-Attribut muss ich mir merken, weil es nicht
  // foederiert werden darf.

  poOrgESAttribute = *iOrgESAttribute;

  // Zu jedem Original-ES-Attribut gibt es eine Liste von Original-FS-Attributen
  // Diese Liste enthaelt zur Zeit jedoch nur ein Element,
  // da nur ein Foederiertes Schema implementiert ist.
  apListeDerOrgFSAttribute = (*iOrgESAttribute)->Post();

  // Schleife zum Durchlaufen aller Attribute der FS-Ebene
  for( iOrgFSAttribute = apListeDerOrgFSAttribute.begin();
       iOrgFSAttribute != apListeDerOrgFSAttribute.end();
       iOrgFSAttribute++ ) {

    // Ab hier im Graph wieder abwaerts wandern
    // Die Liste foederierten ES-Attribute ist der Vorgaenger der FS-Attribute
    apListeDerFedESAttribute = (*iOrgFSAttribute)->Pre();

    // Schleife zum Durchlaufen aller Attribute der ES-Ebene
    for( iFedESAttribute = apListeDerFedESAttribute.begin();
         iFedESAttribute != apListeDerFedESAttribute.end();
         iFedESAttribute++ ) {

        apListeDerFedCSAttribute = (*iFedESAttribute)->Pre();

        // Schleife zum Durchlauf aller Attribute der CS-Ebene
        for( iFedCSAttribute = apListeDerFedCSAttribute.begin();
             iFedCSAttribute != apListeDerFedCSAttribute.end();
             iFedCSAttribute++ )
          // Das Original-CS-Attribut muss ausgeschlossen werden
          if( *iFedCSAttribute != poOrgCSAttribute ) {
            // *iFedCSAttribute ist nun ein Zeiger auf ein foederiertes Attribut auf CS-Ebene
            oEqAttribute.Attribute = *iFedCSAttribute;
            oEqAttribute.Type = (*iFedCSAttribute)->IsComponentOf();

            // foederiertes Attribut in Ergebnisliste schreiben
            paFedAttributeList->push_back( oEqAttribute );
          } // END Schleife zum Durchlauf aller Attribute der CS-Ebene

    } // END Schleife zum Durchlaufen aller Attribute der ES-Ebene

  } // END Schleife zum Durchlaufen aller Attribute der FS-Ebene

} // END Schleife zum Durchlaufen aller Original-ES-Attribute
```

```
  return( paFedAttributeList );
};
```

Die eigentlichen Methoden `Insert`, `Update` und `Delete` sind nicht unbedingt kompliziert, wenn auch lang. Sie durchlaufen die Rückgabelisten der Methoden `GetEqObjects` und `GetEqAttributes` und sammeln sich darauf entsprechende Informationen zusammen. Die einzige, vielleicht noch wichtige Stelle ist diejenige, in der Methode `Insert`, wo die zusätzlichen Informationen gebraucht werden.

```
if( (*iOrgAttributeList).Attribute->Type()->TypeId() == ID_POINTERTYPE ) {

  PCPointerType  poPointerType       = (PCPointerType)
       (*iOrgAttributeList).Attribute->Type();
  PCComplexType  poTargetComplexType = (PCComplexType) poPointerType->TargetType();
  PCObject       poTargetObject =
       poTargetComplexType->GetObject( (*iOrgAttributeList).Value );
  TEqObjectList* paEqObjectList; // fuer die Rueckgabe von GetEqualObjects
  TEqObjectList::iterator iEqObjectList;

  paEqObjectList = getEqObjects( poTargetObject );
  iEqObjectList = paEqObjectList->begin();
  // Hier reicht uns das erste Element der Liste, da nur eine weitere
  // Datenbank angeschlossen ist.

  paFedAttributeList = getEqAttributes( (*iOrgAttributeList).Attribute );
  iFedAttributeList = paFedAttributeList->begin(); // siehe iEqObjectList

  oInsert.AddAttribute( (*iFedAttributeList).Attribute->Name(),
       (*iEqObjectList).Object->Name() );

  delete( paEqObjectList, paFedAttributeList );
}
```

Die Basisalgorithmen aus Abschnitt 20.1 wurden entgegen der ursprünglichen Planung nicht genutzt.

20.2.4 Known Bugs

Sämtliche Methoden dieser Klasse sind zunächst nur auf Funktionalität programmiert worden. Sie enthalten keinerlei Fehlerabfangroutinen. Dies bedeutet, daß sie sich auf einen korrekten Graphen und korrekte Übergabeparameter verlassen. Auch wird davon ausgegangen, daß nach dem Rückliefern der Operationsobjekte nachträglich keine Fehler mehr auftauchen, z.B. von den Adaptern, und somit ein tatsächliches Einfügen/Ändern in den Datenbanken gar nicht stattgefunden hat. Entsprechende Modifikationen im Graphen werden in diesem Falle nicht rückgängig gemacht.

Das Löschen von Objekten aus dem Graphen ist noch nicht vorgesehen.

Auf Effizienz wurde in dieser Stufe der Entwicklung noch keine Rücksicht genommen.

Kapitel 21

Die O_2-Adapter für das Angiographiesystem

Verfasser: Sven Gerding

21.1 Die Analyse

Es ist festgelegt worden, daß ausschließlich Patienten-, Behandlungs- und Materialdaten föderiert werden. Patietendaten werden nicht mehr lokal im unter O_2 realisierten Angiographiesystem eingegeben, sondern über das Föderierungssystem eingefügt. Im Rahmen einer Behandlung verbrauchtes Material wird lokal erfaßt und zusätzlich dem Föderierungssystem kenntlich gemacht. Der Agent für den Im- und Export von Daten gliedert sich in zwei Programme: Der Import-Adapter empfängt Objekte vom FDBS und fügt diese in die O_2-Datenbank ein. Der Export-Adapter verschickt im Angiographiesystem erfaßte Abrechnungsdaten, d.h. Medikamente, Drähte und Katheter an das FDBS.

21.2 Der Entwurf für den Import-Adapter

Empfangen werden über das FDBS ausschließlich Objekte vom Typ `Patient` und `Behandlung`. Das Einfügen eines neues Patienten (insert) geschieht dabei folgendermaßen:

- Empfangen eines Objektes vom Typ Patient vom FDBS.
- Erzeugung eines neues Objektes vom Typ CPatient in O_2.
- Eintragen der entsprechenden, zuvor vom FDBS empfangenen Attributwerte.
- Auslesen der OID des neues Objektes und Übergabe dieser OID an das FDBS. Die OID ist vom OID-Manager bei der Instanziierung vergeben worden.
- Kontrolle wieder an FDBS übergeben.

Nach dem Einfügen eines neuen Patienten wird die zugehörige Behandlung angelegt, die ebenfalls vom FDBS übermittelt wird. Das Anlegen der Behandlung geschieht analog zum obigen Einfügen eines Patienten:

- Empfangen eines Objektes vom Typ Behandlung vom FDBS.

- Suchen des Patienten-Objektes über die Patienten-ID, die Teil des empfangenen Behandlungs-Objekts ist. Das Suchen geschieht über ein OQL-Statement.
- Erzeugen einer neuen Behandlung über die Methode `neue_behandlung` der Klasse `CPatient`.
- Auslesen des OID der neuen Behandlung und Übergabe an das FDBS.
- Kontrolle wieder an das FDBS übergeben.

21.3 Die Implementierung des Import-Adapters

Für den Import-Adapter war es notwendig, fast das gesamte Schema des Angiographiesystems zu exportieren. Der Export der O_2-Klassen geschieht über ein mittels `o2makegen` aus einer Konfigurationsdatei erzeugten Makefiles. Objekte werden wie von der COMI vorgegeben in der Klasse `CInsert-Handler` empfangen. Der eigentliche DB-Zugriff geschieht in der Klasse `CO2Handler`. Diese Aufteilung wurde gewählt, um den Code übersichtlicher zu machen. Ist ein Objekt empfangen worden, so wird zunächst ermittelt, um was für ein Objekt es sich handelt. Bei einem Patienten-Objekt wird die Methode `insert_patient` der Klasse `CO2Handler` aufgerufen. Hier erfolgt der Eintrag in die Datenbank. Ein neues Objekt vom Typ `CPatient` wird erzeugt und die Attributwerte eingetragen. Anschließend wird das Objekt in die Menge `allePatienten` eingefügt. Über die Methode `export_OID` der Klasse `CPatient` wird die OID des neuen Objekts ausgelesen und dem FDBS übergeben.
Ist das empfangene Objekt vom Typ Behandlung, so wird über dessen Attribut `Patient`, das die OID des zuvor einfügten Patienten-Objekts ist, eben dieses Objekt aus der Datenbank gelesen und anschließend die Methode `new_treatment` aufgerufen. Hier wird lediglich eine neue leere Behandlung für den Patienten angelegt. Danach wird die OID des Behandlungs-Objekts ermittelt und an das FDBS übermittelt. Alle Einfügeoperationen müssen sich natürlich innerhalb einer O_2-Transaktion befinden.
Neben dem Insert-Handler wurden auch Update- und Delete-Operationen implementiert, so daß im Falle einer Erweiterung der Funktionalität des FDBS-Kerns der Import-Adapter nur neu compiliert werden muß, um auch Update- und Delete-Operationen zu unterstützen. Der Aufruf des Adapters erfolgt mit:
`angio_in <FDBS-Name> <Adapter-Name> <Base-Name>`.
Im getesteten Szenario ist das:
`angio_in FDBS CSAngio angio1`.

Der Quellcode

```
//
//  Datenbankadapter zum Empfangen von Objekten
//

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
extern "C" {
#include <stdio.h>
#include <stdlib.h>
}
#include "o2lib_CC.hxx"
#include "o2template_CC.hxx"
#include "o2util_CC.hxx"
#include "pg290.H"
#include "angio_in_patient.h"

//-----------------------------------------------------------------------
```

```
//  Shutdown Handler
//---------------------------------------------------------------------------

class CShutdownHandler : public CHandler
{
public:
   CShutdownHandler( const string i_sFdbs )
      :m_sFdbs( i_sFdbs )
   {};


   CShutdownHandler( const CShutdownHandler* other )
     :m_sFdbs( other->m_sFdbs )
   {}


   virtual CHandler* Clone()
   {
      return (CHandler*) new CShutdownHandler( this );
   }


   virtual bool Handle( COperation* poOp )
   {
      CStandardOp* pTheOp = (CStandardOp*) poOp;

      cout << "\n";
      cout << "INFO:\n";
      cout << "emfange shutdwon-Nachricht von " << pTheOp->sSender << "\n"
          << flush;

      // wenn der Shutdown-Befehl nicht vom FDBS kommt
      if( pTheOp->sSender != m_sFdbs ){
         SendDetach();
      }

      // Adapter herunterfahren
      CReceiver* poRec = CReceiver::Instance();
      poRec->Shutdown();

      return true;
   }



   void SendDetach()
   {
      // den Namen des Adapters ermitteln
      CReceiver* poRec = CReceiver::Instance();
      string sMyName = poRec->Name();

      cout << "INFO:\n";
      cout << "Abmeldung bei " << m_sFdbs << " erfolgt\n";
      // Detach-Operation fuer FDBS vorbereiten,
```

```
      // Sender ist der eigene Adapter
      CStandardOp oDetachOp( ID_DETACH );
      oDetachOp.sSender = sMyName;

      // Operation an FDBS verschicken
      CSender oSender;
      oSender.Init( m_sFdbs );
      oSender.Send( &oDetachOp );
   }

private:
   string m_sFdbs; // der Name des FDBS

};// CShutdownHandler


//--------------------------------------------------------------------------
// Insert Handler
//--------------------------------------------------------------------------

class CInsertHandler : public CHandler
{
public:
  CInsertHandler( const CInsertHandler* other )
  {}

  CInsertHandler()
  {}

  virtual CHandler* Clone()
  {
    return (CHandler*) new CInsertHandler( this );
  }

  virtual bool Handle( COperation* poOp )
  {
    CO2Handler oO2H;
    RWCString sClass, sPID;

    assert( poOp != NULL );
    // Downcasting ist zwar nicht schoen, in diesem Fall aber sicher
    CInsert* poOperation = (CInsert*) poOp;

    bool result = FALSE;

    sClass = poOperation->sClass;

    if( sClass == "Patient" )
    {
      result =  poOperation->GetValue( "Name", &oO2H.sName );
      result &= poOperation->GetValue( "Vorname", &oO2H.sVorname );
      result &= poOperation->GetValue( "Geburtsdatum", &oO2H.sGeburtsdatum );
      result &= poOperation->GetValue( "PLZ", &oO2H.sPLZ );
      result &= poOperation->GetValue( "Ort", &oO2H.sOrt );
```

```
      result &= poOperation->GetValue( "Land", &oO2H.sLand );
      result &= poOperation->GetValue( "Staatsangehoerigkeit", &oO2H.sNation );
      result &= poOperation->GetValue( "Geschlecht", &oO2H.sGeschlecht );
      result &= poOperation->GetValue( "Telefon", &oO2H.sTelefon );

      if( result )
      {
        poOperation->sLoid = oO2H.insert_patient();
      }
      else
      {
        cout << "angio_in: Fehler bei Receive (Klasse CPatient)" << endl;
        return false;
      }
    } else
    if( sClass == "Behandlung" )
    {
      result = poOperation->GetValue( "Patient", &sPID );
      if( result )
      {
        poOperation->sLoid = oO2H.new_treatment( sPID );
      }
      else
      {
        cout << "angio_in: Fehler bei receive (Klasse CBehandlung)" << endl;
        return false;
      }
    }
    return true;
  }
};

//------------------------------------------------------------------------------
// Update Handler
//------------------------------------------------------------------------------

class CUpdateHandler : public CHandler
{
public:
  CUpdateHandler( const CUpdateHandler* other )
  {}

  CUpdateHandler()
  {}

  CHandler* Clone()
  {
    return (CHandler*) new CUpdateHandler( this );
  }

  virtual bool Handle( COperation* poOp )
  {
    CO2Handler oO2H;
```

```
    assert( poOp != NULL );
    // Downcasting ist zwar nicht schoen, in diesem Fall aber sicher
    CUpdate* poOperation = (CUpdate*) poOp;

    bool result = FALSE;

    result = poOperation->GetValue( "Name", &oO2H.sName );
    result &= poOperation->GetValue( "Vorname", &oO2H.sVorname );
    result &= poOperation->GetValue( "Geburtsdatum", &oO2H.sGeburtsdatum );
    result &= poOperation->GetValue( "PLZ", &oO2H.sPLZ );
    result &= poOperation->GetValue( "Ort", &oO2H.sOrt );
    result &= poOperation->GetValue( "Land", &oO2H.sLand );
    result &= poOperation->GetValue( "Staatsangehoerigkeit", &oO2H.sNation );
    result &= poOperation->GetValue( "Geschlecht", &oO2H.sGeschlecht );
    result &= poOperation->GetValue( "Telefon", &oO2H.sTelefon );

    if( result )
    {
     oO2H.update_patient( poOperation->sLoid );
     }
     else
     {
       cout << "Fehler bei Receive" << endl;
       return false;
     }
    return true;
   }
};

//-------------------------------------------------------------------------
// Delete Handler
//-------------------------------------------------------------------------

class CDeleteHandler : public CHandler
{
public:
  CDeleteHandler( const CDeleteHandler* other )
  {}

  CDeleteHandler()
  {}

  CHandler* Clone()
  {
    return (CHandler*) new CDeleteHandler( this );
  }

  virtual bool Handle( COperation* poOp )
  {
    CO2Handler oO2H;

    assert( poOp != NULL );
    // Downcasting ist zwar nicht schoen, in diesem Fall aber sicher
    CDelete* poOperation = (CDelete*) poOp;
```

```
    oO2H.delete_patient( poOperation->sLoid );
    return true;
   }
};


//------------------------------------------------------------------------
//  SendAttach
//------------------------------------------------------------------------

void SendAttach( const string i_sFdbs, const string i_sAdapter )
{
  cout << "INFO:\n";
  cout << "Anmeldung bei " << i_sFdbs << " erfolgt\n";
   // Attach-Operation fuer FDBS vorbereiten,
   CStandardOp oAttachOp( ID_ATTACH );
   oAttachOp.sSender = i_sAdapter;

   // Operation an FDBS verschicken
   CSender oSender;
   oSender.Init( i_sFdbs );
   oSender.Send( &oAttachOp );
}

//------------------------------------------------------------------------
//  Test insert
//------------------------------------------------------------------------

void test_insert( void )
{
  CO2Handler oO2H;
  RWCString sClass, sPID;

  cout << "Name: "; cin >> oO2H.sName;
  cout << "Vorname: "; cin >> oO2H.sVorname;
  cout << "Geburtsdatum: "; cin >> oO2H.sGeburtsdatum;
  cout << "PLZ: "; cin >> oO2H.sPLZ;
  cout << "Ort: "; cin >> oO2H.sOrt;
  cout << "Land: "; cin >> oO2H.sLand;
  cout << "Nationalitaet: "; cin >> oO2H.sNation;
  cout << "Geschlecht: "; cin >> oO2H.sGeschlecht;
  cout << "Telefon: "; cin >> oO2H.sTelefon;

  oO2H.insert_patient();
}


//------------------------------------------------------------------------
//  Das Hauptprogramm
//  Returnwert: int
//------------------------------------------------------------------------

int main(int argc, char **argv)
```

```
{
   d_Session session;
   d_Database database;
   d_Transaction update;


   if( argc != 4)
   {
     cout << "Argumente: FDBS-Name, Adapter-Name, Base-Name\n";
     return( EXIT_FAILURE );
   }

   string sFdbs = argv[1];
   string sAdapter = argv[2];
   string sBase = argv[3];

   session.set_default_env();
   if ( session.begin ( argc, argv ) )
   {
     cerr << "Something wrong to start o2" << endl;
     exit( EXIT_FAILURE );
   }

   cout << "Opening Database." << endl;

   database.open( sBase );

   if ( sAdapter == "test" )
   {
     cout << "Fuehre test insert aus" << endl;
     test_insert();
   }
   else
   {
   BEGIN_REGISTRATION;
      REGISTER_OPERATION( CInsert, CInsertHandler );
      REGISTER_OPERATION( CUpdate, CUpdateHandler );
      REGISTER_OPERATION( CDelete, CDeleteHandler );
      REGISTER_OPERATION( CStandardOp( ID_SHUTDOWN ), CShutdownHandler( sFdbs ) );
   END_REGISTRATION;

   CReceiver* poReceiver = CReceiver::Instance();
   if( !poReceiver->Init( sAdapter ) )
     return( EXIT_FAILURE );

   // beim Foederierungssystem anmelden
   SendAttach( sFdbs, sAdapter );

   cout << "DB-Adapter working." << endl;
   poReceiver->WaitForOperation();

   poReceiver->Terminate();
   CReceiver::Destroy();
   }
```

```
   cout << "Closing Database." << endl;
   database.close();

   session.end();
   cout << "Programm terminated." << endl;
   return( EXIT_SUCCESS );
}


//---------------------------------------------------------------------------
//  angio_in_patient.cc
//---------------------------------------------------------------------------

#include <iostream.h>
#include <rw/cstring.h>
extern "C" {
#include <stdio.h>
#include <stdlib.h>
}
#include "o2lib_CC.hxx"
#include "o2template_CC.hxx"
#include "o2util_CC.hxx"
#include "OQL_CC.hxx"
#include "angio_in_patient.h"


//---------------------------------------------------------------------------
//  Methode:    convert_sex   -  String nach boolean
//  Parameter:  String        -  "0"=w ,"1"=m
//  Returnwert: char          -  Entsprich O2 Boolean
//---------------------------------------------------------------------------

char CO2Handler::convert_sex( RWCString i_sSex )
{
  char cSex;

  cSex = i_sSex[0];
  return( cSex);
}

//---------------------------------------------------------------------------
//  Methode:    convert_date  -  O2 Date Objekt aus Datumstring erzeugen
//  Parameter:  String        -  Format: dd.mm.jj
//  Returnwert: d_Ref<Date>
//---------------------------------------------------------------------------

d_Ref<Date> CO2Handler::convert_date( RWCString i_sDate )
{
  int zDay, zMonth, zYear;
  RWCString sDay, sMonth, sYear;
  d_Ref<Date> poDate;

  // cast von RWCSubString nach const char* ist nicht moeglich!
```

```
  sDay = i_sDate( 0,2 );
  sMonth = i_sDate( 3,2 );
  sYear = i_sDate( 6,2 );

  zDay = atoi( sDay );
  zMonth = atoi( sMonth );
  zYear = atoi( sYear ) + 1900;

  poDate = new Date( zDay, zMonth, zYear );

  return( poDate );
}

//-------------------------------------------------------------------------
//  Methode:    create_predicate  -  Query-String erstellen
//  Parameter:  String            -  LOID
//  Returnwert: const char
//-------------------------------------------------------------------------

const char *create_predicate(const char *i_sLOID)
{
  char *sPredicate;

  if( !(sPredicate = (char *) malloc(30)) )
  {
    cerr << "Oops! Couldn't allocate memory!" << endl;
    exit( EXIT_FAILURE );
  }
  sprintf( sPredicate, "this.OID=%s", i_sLOID );

  return( sPredicate );
}

//-------------------------------------------------------------------------
//  Methode:    insert_patient
//  Parameter:  -
//  Returnwert: -
//-------------------------------------------------------------------------

char* CO2Handler::insert_patient( void )
{
  d_Transaction update;
  d_Set<d_Ref<CPatient> > poPatienten( "allePatienten" );
  d_Ref<CPatient> poPatient;
  d_Ref<Date> poBirthDate;

  char sLOID[6];

  poBirthDate = convert_date( sGeburtsdatum );

  // Um vom RWCString zum O2String zu kommen, nehmen
  // wir den Umweg ueber const char !

  update.begin();
```

```
  poPatient = new( CPatient );
  poPatient->Name = (const char *)sName;
  poPatient->Vorname = (const char *)sVorname;
  poPatient->Geburtsdatum = poBirthDate;
  poPatient->Geschlecht = convert_sex( sGeschlecht );
  poPatient->Strasse = (const char *) sStrasse;
  poPatient->Plz = (const char *) sPLZ;
  poPatient->Ort = (const char *) sOrt;
  poPatient->Land = (const char *) sLand;
  poPatient->Staatsangeh = (const char *) sNation;
  poPatient->Telefon = (const char *) sTelefon;

  poPatienten.insert_element( poPatient );
  sprintf( sLOID, "%d", poPatient->export_OID() );
  update.commit();

  return( sLOID );
}

//----------------------------------------------------------------------------
//  Methode:    update_patient
//  Parameter:  int             - OID des Patienten
//  Returnwert: -
//----------------------------------------------------------------------------

void CO2Handler::update_patient( const char *i_sLOID )
{
  d_Transaction update;
  d_Set<d_Ref<CPatient> > poPatienten( "allePatienten" );
  d_Ref<CPatient> poPatient;
  d_Ref<Date> poBirthDate;

  poBirthDate = convert_date( sGeburtsdatum );

  char *sPredicate;

  sPredicate = create_predicate( i_sLOID );

  poPatient = NULL;
  update.begin();
  poPatient = poPatienten.select_element( sPredicate );
  if( poPatient )
  {
    poPatient->Name = (const char *) sName;
    poPatient->Vorname = (const char *) sVorname;
    poPatient->Geburtsdatum = poBirthDate;
    poPatient->Geschlecht = convert_sex( sGeschlecht );
    poPatient->Strasse = (const char *) sStrasse;
    poPatient->Plz = (const char *) sPLZ;
    poPatient->Ort = (const char *) sOrt;
    poPatient->Land = (const char *) sLand;
    poPatient->Staatsangeh = (const char *) sNation;
    poPatient->Telefon = (const char *) sTelefon;
  }
```

```
  update.commit();
  free( sPredicate );
}

//-----------------------------------------------------------------------------
//  Methode:    delete_patient
//  Parameter:  int             - OID des Patienten
//  Returnwert: -
//-----------------------------------------------------------------------------

void CO2Handler::delete_patient( const char *i_sLOID )
{
  d_Transaction update;
  d_Set<d_Ref<CPatient> > poPatienten( "allePatienten" );
  d_Ref<CPatient> poPatient;

  char *sPredicate;

  sPredicate = create_predicate( i_sLOID );

  poPatient = NULL;
  update.begin();
  poPatient = poPatienten.select_element( sPredicate );
  if( poPatient )
  {
    poPatienten.remove_element( poPatient );
  }
  update.commit();
  free( sPredicate );
}

//-----------------------------------------------------------------------------
//  Methode:    new_treatment
//  Parameter:  string             - OID des Patienten
//  Returnwert: -
//-----------------------------------------------------------------------------

char* CO2Handler::new_treatment( const char *i_sLOID )
{
  d_Transaction update;
  d_Set<d_Ref<CPatient> > poPatienten( "allePatienten" );
  d_Ref<CPatient> poPatient;

  char sLOID[6];
  char *sPredicate;
  int  zBOID;

  sPredicate = create_predicate( i_sLOID );

  poPatient = NULL;
  update.begin();
  cout << "\n  Retrieving Patient ..." << flush;
  poPatient = poPatienten.select_element( sPredicate );
  cout << " done." << endl;
```

```
  cout << "  new_treatment: Modifying Patient " << poPatient->Name << endl;
  zBOID = poPatient->neue_behandlung();
  sprintf( sLOID, "%d", zBOID );
  update.commit();
  return( sLOID );
}
```

21.4 Der Entwurf für den Export-Adapter

Strenggenommen gliedert sich der Export-Adapter nochmals in zwei Teile, der Triggersimulation im Angiographiesystem und dem C++ Programm, das die Daten an das FDBS verschickt. Hierzu ist es erforderlich, die O_2-Applikation zu modifizieren. Der Export von Objekten erfolgt in zwei Schritten:

1. Applikation: Transfer der Daten an den Export-Adapter durch Aufruf der Methode `export_object`.
2. Adapter: Verschicken der zuvor empfangen Daten an das FDBS.

21.5 Die Implementierung des Export-Adapters

Wird in der Applikation Material zu einer Behandlung eingegeben, so müssen diese Daten dem DB-Agenten übermittelt werden, der die Daten an das FDBS weiterleitet. Da O_2 keine Trigger anbietet, mußte ein anderer Weg gefunden werden, die Materialdaten aus der Applikation zu exportieren. Die Wahl fiel hierbei auf Unix-Pipes. Den Klassen `CDraht`, `CKatheter` und `CMedikament` wurde dazu die Methode `export_object` hinzugefügt. Diese Funktion öffnet eine festgelegte Pipe und schreibt Klassennamen, OID, die entsprechenden Attributnamen und die dazugehörigen Werte, gefolgt von einer Objektende-Kennung hinein. Das Format entspricht hierbei schon der vom FDBS erwarteten Datenstruktur, so daß auf eine weitere Aufbereitung der übermittelten Daten im DB-Agenten weitgehend verzichtet werden kann. Die eigentliche Arbeit des Export-Adapters besteht nur noch aus einer Schleife, in der Daten aus der Pipe gelesen werden, bis die Objektende- Kennung auftritt, um anschließend diesen Datensatz über die von der COMI zur Verfügung gestellten Funktionen an das FDBS zu verschicken.

Der Quellcode der Triggersimulation

```
method body export_object in class CDraht
{
#include "stdio.h"
#include "stdlib.h"

  FILE *fp;

  if ( !(fp = fopen("/home/pg/pg290/gerding/.angiofdbs", "w+")) )
  {
    fprintf ( stderr, "Couldn't open pipe!\n" );
  }
  else
  {
    fprintf( fp, "CMaterial\n" );
    fprintf( fp, "%d\n", self->OID );
    fprintf( fp, "%d\n", BOID );
    fprintf( fp, "Bezeichnung\n%s\n", self->Produkt );
    fprintf( fp, "Preis\n%f\n", self->Preis );
```

```
    fprintf( fp, "#EOO#\n" );
    fclose( fp );
    sleep(1);
}};
```

Der Quellcode des Export-Adapters

```
//---------------------------------------------------------------------------
//  Datenbankadapter zum Versenden von Objekten
//---------------------------------------------------------------------------

#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
extern "C" {
#include <stdio.h>
#include <stdlib.h>
}
#include "pg290.H"
#include "angio_out.h"

#define DBA_PIPE "/home/pg/pg290/gerding/.angiofdbs"

CSender *poFDBS;
char *sProg;

//--------------------------------------------------------------------------
//  Das Hauptprogramm
//  Returnwert: int
//--------------------------------------------------------------------------

int main(int argc, char **argv)
{
  if( argc != 2)
  {
    cout << "Usage: " << argv[0] << " <server-name>" << endl;
    return( EXIT_FAILURE );
  }

  sProg = argv[0];
  // Adapter initialisieren

  poFDBS = new CSender;
  if( !poFDBS->Init( argv[1] ) )
  {
    cerr << sProg << ": Error while initializing adapter!" << endl;
    delete poFDBS;
    exit( EXIT_FAILURE );
  }

  CAdapter oAdapter;

  // Hauptschleife: Objekt aus Pipe lesen und verschicken
```

```
  while( TRUE )
  {
    oAdapter.get_object();
    oAdapter.send_object();
  }

  // Wird nicht erreicht

  poFDBS->Terminate();
  delete poFDBS;

  return( EXIT_SUCCESS );
}

//---------------------------------------------------------------------------
// Konstruktor
//---------------------------------------------------------------------------

CAdapter::CAdapter(void)
{
  bTransfer = FALSE;
}

//---------------------------------------------------------------------------
//  Objekt aus der O2Look Applikation empfangen
//---------------------------------------------------------------------------

void CAdapter::get_object(void)
{
  int iCount;
  bool b_End_Of_Object;
  char sInput[80];

  ifstream pipe( DBA_PIPE );
  if( !pipe )
  {
    cerr << sProg << ": Unable to open pipe: " << DBA_PIPE << endl;
    exit( EXIT_FAILURE );
  }

  while (pipe.eof()) {}
  pipe.getline(sInput, 80);
  sClass = sInput;
  while (pipe.eof()) {}
  pipe.getline(sInput, 80);
  sLOID = sInput;
  pipe.getline(sInput, 80);
  sAggregate = sInput;

  b_End_Of_Object = FALSE;
  iNumAttrs = 0;
  while( !b_End_Of_Object )
  {
    pipe.getline(sInput, 80);
```

```
    if( strcmp(sInput,"#EOO#") != 0 )
    {
      sAttrName[iNumAttrs] = sInput;
      pipe.getline(sInput, 80);
      sAttrValue[iNumAttrs] = sInput;
      iNumAttrs++;
    }
    else b_End_Of_Object = TRUE;
  }
  pipe.close();

  cout << "Object read:" << endl;
  cout << "----------------------------" << endl;
  cout << "Object " << sClass << ", with OID " << sLOID;
  cout << ", Aggregate " << sAggregate;
  cout << " has " << iNumAttrs << " Attributes"  << endl;
  for( iCount = 0; iCount < iNumAttrs; iCount++ )
  {
    cout << iCount+1 << ". " << sAttrName[iCount];
    cout << ": " << sAttrValue[iCount] << endl;
  }
  cout << "----------------------------" << endl;

  bTransfer = TRUE;
}

//-----------------------------------------------------------------------------
// Objekt an FDBS versenden
//-----------------------------------------------------------------------------

void CAdapter::send_object(void)
{
  CInsert* poInsOp;
  int iCount;

  if( bTransfer )
  {
    poInsOp = new CInsert;

    poInsOp->sSchema = "CSAngio";
    poInsOp->sClass = sClass;
    poInsOp->sLoid = sLOID;
    poInsOp->sAggregateOid = sAggregate;

    for( iCount = 0; iCount < iNumAttrs; iCount++ )
    {
      poInsOp->AddAttribute(sAttrName[iCount], sAttrValue[iCount] );
    }

    poFDBS->Send(poInsOp);
    cout << "Object sent!" << endl;
    bTransfer = FALSE;
  }
  else
```

```
  {
    cerr << sProg << ": (send_object)  No/Wrong Object" << endl;
    exit( EXIT_FAILURE );
  }
}
```

Kapitel 22

Die Entwicklung des Oracleadapters für das Abrechnungssystem

Verfasser: Mischa Lohweber

Der Adapter des Abrechnungssystems teilt sich in zwei Komponenten. In den Importadapter, welcher Daten von Föderierungssystem empfangt und in die *Oracle*-Datenbank des Abrechnungssystems einträgt und den Exportadapter, welcher Daten aus dem Abrechnungssystems an das Föderierungssystem versendet. Beide Adapter laufen als unabhängige Prozesse im Hintergrund und kommunizieren über die COMI mit dem Föderierungssystem.

22.1 Der Importadapter

Der Importadapter dient dazu, Daten, welche vom Föderierungssystem kommen, in die Datenbank des Abrechnungssystems einzutragen.

22.1.1 Anforderungen an den Importadapter

Die Daten, welche der Adapter vom Föderierungssystem erhält sind in Tabelle 22.1 dargestellt. Der Adapter soll diese Daten vom Föderierungssystem erhalten, und dann in die Datenbank des Abrechnungssystems einfügen. Der Adapter empfängt also ein Tupel der Form `('AbrechnungsID', 'Bezeichnung', 'Preis', 'Anzahl')`. Beispiele für solche Tupel wären z.B.:

```
124 Spritze 10.45 1
120 Katheder 12.00 1
124 Schlauch 4.00 1
120 Katheder 15.00 1
```

Das Föderierungssystem liefert ein Material, welches einer bestimmten Abrechnung zugeordnet ist. Die Abrechnung ist eindeutig über den Fremdschlüssel `AbrechnungsID` definiert. Das einzufügende

MATERIAL		NOT NULL	PK/FK	Unique
AbrechnungsID	NUMBER	x	PK/FK	x
Bezeichnung	VARCHAR	x	PK	
Preis	NUMBER	x	PK	
Anzahl	NUMBER	x		

Tabelle 22.1: Importschema MATERIAL.

Material ist durch die Kombination der Attribute `AbrechnungsID, Bezeichnung` und `Preis` eindeutig zuzuordnen. Dies bedeutet, daß es zu einer Abrechnung genau ein Material mit einem eindeutigen Preis gibt, so daß der oben genannten Schlüssel erfüllt wird. Das Feld `Anzahl` besitzt immer den Wert eins. Dies bedeutet, daß Materialien, welche vom Föderierungssystem kommen, immer einzeln in das Abrechnungssystem eingefügt werden. In der Implementation des Adapters wird das Feld `Anzahl` also nicht real föderiert, sondern es wird immer ein Wert von eins angenommen. Wird ein Material vom Föderierungssystem geliefert, gibt es mehrere Möglichkeiten der Verarbeitung:

1. Das Material existiert bereits: Dies bedeutet, daß es dieses Material bereits gibt und einer Abrechnung zugeordnet ist. In diesen Fall wird die Anzahl des Materials in der *Oracle*-Datenbank um eins erhöht.
2. Das Material existiert nicht: Dies bedeutet, daß die Abrechnung mit der ID `AbrechnungsID` dieses Material noch nicht zugeordnet ist. In diesem Fall wird das Material mit Anzahl eins in die *Oracle*-Datenbank eingefügt.
3. Die Abrechnung existiert nicht: Dies bedeutet, daß keine Abrechnung mit der ID `AbrechnungsID` existiert. In diesem Fall wird ein Fehler ausgegeben.

Dadurch, daß der Preis des Materials mit in den Schlüssel eingebunden ist, ist sichergestellt, daß Preisänderungen sich nicht auf schon vorhandene Materialien auswirken, da bei verändertem Preis der Schlüssel nicht mehr derselbe wäre. Dadurch bleiben schon vorhandene Materialien bestehen und Materialien mit neuem Preis werden als neues Material zu der jeweiligen Abrechnung eingefügt.

22.1.2 Realisierung des Importadapters

Der Adapter ist ein C++ Programm, welches als eigenständiger Prozeß läuft. Der eigentliche Code ist in *Pro*C* geschrieben und wird durch einen Pre-Compiler von Oracle in C++ Code umgewandelt. Die Verwendung von *Pro*C* hat den Vorteil, *Embedded SQL* einzusetzen, so daß im C++ Code direkt SQL Statements eingebunden werden können. Der Importadapter enthält zwei Klassendefinitionen (`CShutdownHandler, CInsertHandler`) und einen Hauptteil (`main`). Der Hauptteil ist einmal dafür zuständig, den Adapter beim Föderierungssystem anzumelden und andererseits diesem die Klassen `CInsertHandler` und `CShutdownHandler` zur Verfügung zu stellen. Die Anmeldung geschieht über das Makro `REGISTER_OPERATION`, welches von der COMI zur Verfügung gestellt wird.

```
main {...

 /* Operationen registrieren */
 BEGIN_REGISTRATION;
    REGISTER_OPERATION( CStandardOp( ID_SHUTDOWN ), CShutdownHandler( sFdbs ) );
    REGISTER_OPERATION( CInsert, CInsertHandler );
 END_REGISTRATION;

 /* beim Foederierungssystem anmelden */
 SendAttach( sFdbs, sAdapter );

 /* Operationsempfang starten */
 CReceiver* poRec = CReceiver::Instance();
 if( !poRec->Init( sAdapter ) ){
    return 1;
 }

   poRec->WaitForOperation();

...}
```

Die Klasse `CShutdownHandler` ethält Operationen, die ausgeführt werden sollen, wenn der Adapter beendet wird. Die Klasse `CInsertHandler` stellt dem Föderierungssystem unter anderem die Methode `Handle()` zur Verfügung, welche den eigentlichen Code zum Eintragen der Daten in die Datenbank enthält. Der komplette Quellcode ist in Abschnitt 22.3 zu finden. Für die Implementierung in *Embedded SQL* wurde SQL Methode 3 benutzt, die es erlaubt, auf Ergebnissen einer SQL-Abfrage zu navigieren. Die Ergebnisse können dann in sogenannte HOST-Variablen kopiert werden, die als normale C++ Variablen gelten und weiterverarbeitet werden. Um z.B. die Anzahl eines Materials zu bekommen, wird folgender *Embedded SQL* -Code verwendet:

```
EXEC SQL BEGIN DECLARE SECTION;
#define     EINGABE_LEN    200
#define     SQL_LEN        800
VARCHAR  countstmt[SQL_LEN];
VARCHAR  materialbezeichnung[EINGABE_LEN];
VARCHAR  materialpreis[EINGABE_LEN];
VARCHAR  materialanzahl[EINGABE_LEN];
VARCHAR  materialabrechnungsID[EINGABE_LEN];
VARCHAR  zaehler[EINGABE_LEN];
EXEC SQL END DECLARE SECTION;

....

/* SQL Statement um Anzahl der Materialien zu einer Abrechnung zu bestimmen */
strcpy((char *)countstmt.arr, \
"SELECT COUNT (AbrechnungsID) FROM Material \
WHERE AbrechnungsID = :v1 AND Bezeichnung = :v2 AND Preis = :v3");
countstmt.len = strlen((char *)countstmt.arr);

/* Statement zur Verarbeitung vorbereiten,
   Cursor definieren und Statement ausfuehren. */
EXEC SQL PREPARE S FROM :countstmt;
EXEC SQL DECLARE C CURSOR FOR S;
EXEC SQL OPEN C USING :materialabrechnungsID, /
                      :materialbezeichnung, :materialpreis;

/* Hohle die Anzahl der gefundenen Materialien zu einer Abrechnung */
EXEC SQL FETCH C INTO :zaehler;
EXEC SQL CLOSE C;
```

Der komplette Quellcode ist in Abschnitt 22.3 abgebildet.

22.2 Der Exportadapter

Der Exportadapter dient dazu, Daten, welche im Abrechnungssystem eingetragen oder verändert werden, an das Föderierungssystem weiterzuleiten. Diese Daten werden dann von Föderierungssystem verarbeitet, an die O_2-Datenbank weitergeleitet und dort eingetragen.

22.2.1 Anforderungen an den Exportadapter

Die zu exportierenden Daten bestehen aus zwei Klassen, und sind in den Komponentenschemata in Tabelle 22.2 und Tabelle 22.3 dargestellt. Bei der Neueingabe eines Patienten, oder bei Änderungen an den Stammdaten eines Patienten, werden diese Daten an den Exportadapter weitergeleitet. Ebenso

Klasse: Patient		NOT NULL	PK/FK	Unique
Lokale Objekt ID	NUMBER	x		
Name	VARCHAR	x		
Vorname	VARCHAR	x		
Geburtsdatum	DATE	x		
Strasse	VARCHAR			
PLZ	VARCHAR			
Ort	VARCHAR			
Land	VARCHAR			
Telefon	VARCHAR			
Telefax	VARCHAR			
Staatsangehoerigkeit	VARCHAR			
Krankenkasse	VARCHAR			
Geschlecht	VARCHAR	x		

Tabelle 22.2: Komponentenschema: CSAbrsys, Klasse: Patient

Klasse: Abrechnungsdaten		NOT NULL	PK/FK	Unique
Lokale Objekt ID	NUMBER	x		
PatientenID	NUMBER	x		

Tabelle 22.3: Komponentenschema: CSAbrsys, Klasse: Abrechnungsdaten

werden bei der Neuerstellung einer Abrechnung oder bei Änderungen in einer Abrechnung zu einem Patienten, die geänderten Daten an den Exportadapter weitergeleitet.

22.2.2 Realisierung des Importadapters

Änderungen an den Patientenstammdaten oder an den Abrechnungsdaten müssen erkannt und an den Adapter weitergeleitet werden. Diese Informationsfluß wird durch `Trigger` und `Pipes` realisiert. `Trigger` werden einer Tabelle in der Datenbank zugeordnet und sprechen an, sobald die in ihnen definierte Bedingung erfüllt ist. Die Bedingung kann z.B. eine `INSERT` oder `UPDATE` Operation auf diese Tabelle sein. Da Daten aus den Tabellen `Patientenstammdaten` und `Abrechnungsdaten` föderiert werden sollen, müssen in beiden Tabellen seperate Trigger angelegt werden. Der Trigger für die Tabelle `Patientenstammdaten` heißt `Hole_Patientendaten`. wird eine `INSERT`- oder `UPDATE`-Operaton auf der Tablle ausgeführt, wird der Trigger ausgelöst und arbeitet zeilenweise die geänderten oder neuen Daten der Tabelle ab. So ist gewährleistet, daß bei einem Block von Änderungen jeder einzelne Datensaz separat behandelt wird.

```
CREATE OR REPLACE TRIGGER Hole_Patientendaten
AFTER INSERT OR UPDATE ON Patientenstammdaten
FOR EACH ROW
```

Dies bedeutet, daß Änderungen immer zeilenweise, also für jeden Datensatz erkannt werden, und der Trigger jeweils pro geändertem Datensatz aktiviert wird. Ist der Trigger ausgelöst worden, liest er den neuen oder geänderten Datensatz aus und packt diesen zu einer `Message` zusammen. Diese `Message` wird dann in eine `Pipe` geschrieben, die später vom Exportadapter ausgelesen.

```
  dbms_pipe.pack_message(:new.patientenid);
  dbms_pipe.pack_message(:new.name);
  .
  .
  dbms_pipe.pack_message(:new.Krankenkasse);
```

```
status := dbms_pipe.send_message('FDBS_PIPE');
```

Ebenso wird in der Tabelle `Abrechnungsdaten` ein Trigger definiert, welcher ebenfalls auf `INSERT` und `UPDATE` Operationen reagiert und auch zeilenweise operiert. Die kompletten PL/SQL-Statements für die Trigger sind in Abschnitt 22.3 zu finden.

Nachdem die Daten ausgelesen wurden, wird die Verbindung zum Föderierungssystem hergestellt. Danach wird ein neues Objekt zum Versenden angelegt, mit den eben erhaltenen Daten gefüllt und an das Föderierungssystem versendet.

```
CSender *poFDBS;
poFDBS = new CSender;

/* Kontakt zum FDBS aufnehmen */
if( !poFDBS->Init( "FDBS" ) )
{
  cerr << sProg << ": Error while initializing adapter!" << endl;
  delete poFDBS;
  exit(1);
}

/* Objekt anlegen */
poInsOp = new CInsert;

/* Objekt fuellen */
poInsOp->sSchema = "CSAbrsys";
poInsOp->sClass = "Patient";
poInsOp->sLoid = sPatientID;

poInsOp->AddAttribute("Name",(char *)Vname.arr);
....
poInsOp->AddAttribute("Krankenkasse",(char *)Vkrankenkasse.arr);

/* Objekt an das FDBS senden */
poFDBS->Send(poInsOp);

/* Objekt wieder loeschen */
delete poInsOp;

/* Verbindung zum FDBS loeschen */
delete poFDBS;
```

22.3 Die Implementierung

Es folgt eine Auflistung der ProC*, C++ und SQL Quelltexte zu dem InsertHandler, ExportHandler und den Definitionen der Trigger und Pipes.

22.3.1 Der ImportAdapter

Der Importadapter besteht aus der Datei `AbrSys_InsertHandler.pc`.

```
/* -*-c++-*- */
/*
 * Name:   AbrSys_InsertHandler.pc
 *
 * Typ:    PRO*C SourceDatei - Muss erst durch den Precompiler laufen!
 *
 * Autor:  Mischa Lohweber
 *
 * Beschreibung: Der AbrSys_InsertHandler stellt dem FDBS ueber die Klasse
 *               CInsterHandler einen Mechanismus zum einfuegen von Daten in
 *               die Oracle Datenbank zur Verfuegung.
 *
 */


#include <rw/cstring.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <sqlca.h>
#include <oraca.h>
#include <string.h>
#include <comi.H>
#include <pg290.H>


/* Declare Oracle error handling function. */
void sql_error(char *msg);



/*********************************************************
class: CShutdownHandler
*********************************************************/
class CShutdownHandler : public CHandler
{
public:
   CShutdownHandler( const string i_sFdbs )
      :m_sFdbs( i_sFdbs )
   {};


   CShutdownHandler( const CShutdownHandler* other )
     :m_sFdbs( other->m_sFdbs )
   {}


   virtual CHandler* Clone()
   {
      return (CHandler*) new CShutdownHandler( this );
   }


   virtual bool Handle( COperation* poOp )
   {
      CStandardOp* pTheOp = (CStandardOp*) poOp;

      /* wenn der Shutdown-Befehl nicht vom FDBS kommt */
      if( pTheOp->sSender != m_sFdbs ){
```

```
         SendDetach();
      }

      /* Adapter herunterfahren */
      CReceiver* poRec = CReceiver::Instance();
      poRec->Shutdown();

      return true;
   }


   void SendDetach()
   {
     /* den Namen des Adapters ermitteln */
      CReceiver* poRec = CReceiver::Instance();
      string sMyName = poRec->Name();

      /* Detach-Operation fuer FDBS vorbereiten, */
      /* Sender ist der eigene Adapter */
      CStandardOp oDetachOp( ID_DETACH );
      oDetachOp.sSender = sMyName;

      /* Operation an FDBS verschicken */
      CSender oSender;
      oSender.Init( m_sFdbs );
      oSender.Send( &oDetachOp );
   }

private:
   string m_sFdbs; /* der Name des FDBS */

};



/*********************************************************
class: CInsertHandler
*********************************************************/
class CInsertHandler : public CHandler
{
public:
   CInsertHandler( const CInsertHandler* other )
   { }

   CInsertHandler()
   { }

   virtual CHandler*  Clone()
   {
      return (CHandler*) new CInsertHandler( this );
   }

   virtual bool Handle( COperation* poOp )
   {

     CInsert* pTheOp = (CInsert*) poOp;

     /* Optionen fuer Oracle deklarieren */
     EXEC ORACLE OPTION (ORACA=YES);
```

```
    /* Variablendeklaration fuer Oracle */
    EXEC SQL BEGIN DECLARE SECTION;
    #define     UNAME_LEN      20
    #define     PWD_LEN        40
    #define     EINGABE_LEN    200
    #define     SQL_LEN        800

    VARCHAR  username[UNAME_LEN];
    varchar  password[PWD_LEN];
    VARCHAR  countstmt[SQL_LEN];
    VARCHAR  selectstmt[SQL_LEN];
    VARCHAR  updatestmt[SQL_LEN];
    VARCHAR  insertstmt[SQL_LEN];
    VARCHAR  stmt[SQL_LEN];
    VARCHAR  materialbezeichnung[EINGABE_LEN];
    VARCHAR  materialpreis[EINGABE_LEN];
    VARCHAR  materialanzahl[EINGABE_LEN];
    VARCHAR  materialabrechnungsID[EINGABE_LEN];
    VARCHAR  zaehler[EINGABE_LEN];
    VARCHAR  sMaterialID[EINGABE_LEN];
    EXEC SQL END DECLARE SECTION;


    /* Variablendeklaration fuer den ORB */
    RWCString sSchema, sClass, sLoid;
    RWCString sAbrechnungsID, sBezeichnung;
    RWCString sPreis;


    /* Lokale C++ Variablen */
    bool result = FALSE;
    int  Anzahl;
    int  SatzAnzahl;
    int  dummy;


    /* Hole Daten via Orb */
    result = pTheOp->GetValue("AbrechnungsID", &sAbrechnungsID);
    result &=  pTheOp->GetValue("Bezeichnung", &sBezeichnung);
    result &=  pTheOp->GetValue("Preis", &sPreis);

if (result)
  {

   // Holen der Material-Daten erfolgreich, Daten in  Datenbank eintragen
   EXEC SQL WHENEVER SQLERROR DO sql_error("ORACLE error:");
   oraca.orastxtf = ORASTFERR;
   username.len = strlen(strcpy((char *)username.arr, "PG290"));
   password.len = strlen(strcpy((char *)password.arr, "PG290"));
   EXEC SQL CONNECT :username IDENTIFIED BY :password;
   cout<<endl<<"Connected to ORACLE."<<endl;
   cout<<endl<<"-----------------------------------------"<<endl;

   materialabrechnungsID.len = /
    strlen(strcpy((char*)materialabrechnungsID.arr, sAbrechnungsID));
   cout<<"AbrechnugsID=  "<<sAbrechnungsID<<endl;

   materialbezeichnung.len = /
```

```
   strlen(strcpy((char *)materialbezeichnung.arr, sBezeichnung));
  cout<<"Bezeichnung=   "<<sBezeichnung<<endl;

  materialpreis.len = strlen(strcpy((char *)materialpreis.arr, sPreis));
    cout<<"Preis=         "<<sPreis<<endl<<endl;


/* SQL Statement um Anzahl der Materialien zu einer Abrechnung zu bestimmen */
strcpy((char *)countstmt.arr, \
"SELECT COUNT (AbrechnungsID) FROM Material \
WHERE AbrechnungsID = :v1 AND Bezeichnung = :v2 AND Preis = :v3");
countstmt.len = strlen((char *)countstmt.arr);

/* Statement zur Verarbeitung vorbereiten, Cursor definieren und Statement
   ausfuehren. */
EXEC SQL PREPARE S FROM :countstmt;
EXEC SQL DECLARE C CURSOR FOR S;
EXEC SQL OPEN C USING :materialabrechnungsID, :materialbezeichnung, :materialpreis;

/* Hohle die Anzahl der gefundenen Materialien zu einer Abrechnung */
EXEC SQL FETCH C INTO :zaehler;
EXEC SQL CLOSE C;

/* Werte Anzahl gefundener Datensaetze aus */
zaehler.arr[zaehler.len] = '\0';
SatzAnzahl = atoi((char *)zaehler.arr);

if (SatzAnzahl == 0) {
  /* Es existiert kein solches Material zu der Abrechnung, Material einfuegen */
  cout<<"Material existiert noch nicht, Material wird eingefuegt."<<endl;
  strcpy((char *)stmt.arr, \
  "INSERT INTO Material (AbrechnungsID, Bezeichnung, Preis, Anzahl)/
                        VALUES (:v1, :v2, :v3, TO_NUMBER('1'))");
  stmt.len = strlen((char *)stmt.arr);
  EXEC SQL PREPARE I FROM :stmt;
  EXEC SQL EXECUTE I USING :materialabrechnungsID, :materialbezeichnung, :materialpreis;
  EXEC SQL COMMIT RELEASE;
  cout<<"-------------------------------------"<<endl;

  /* Dem FDBS irgendeine ID (String) zurueckliefern,
     da Material KEINE eigene ID hat. */
  /* Dem FDBS die Material-ID (String) zurueckliefern
     (Tupel: AbrID, Bezeichnung, Preis). */
  sMaterialID.len = strlen(strcpy((char *)sMaterialID.arr, ""));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,(char *)materialabrechnungsID.arr));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,","));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,(char *)materialbezeichnung.arr));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,","));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,(char *)materialpreis.arr));

  pTheOp->sLoid = (char *)sMaterialID.arr;

  } /* Ende von: IF SatzAnzahl == 0 */
```

```
else if (SatzAnzahl == 1) {
  /* Das Material existiert genau einmal, Anzahl um 1 erhoehen */
  cout<<"Material existiert 1mal, Anzahl = Anzahl + 1."<<endl;
  strcpy((char *)stmt.arr, \
  "SELECT Anzahl FROM Material WHERE AbrechnungsID=:v1 /
                              AND Bezeichnung=:v2 AND Preis=:v3");

  stmt.len = strlen((char *)stmt.arr);

  /* Statement zur Verarbeitung vorbereiten, Cursor definieren
     und Statement ausfuehren. */
  /* Anzahl des Materials holen */
  EXEC SQL PREPARE U FROM :stmt;
  EXEC SQL DECLARE CU CURSOR FOR U;
  EXEC SQL OPEN CU USING :materialabrechnungsID, :materialbezeichnung, :materialpreis;
  EXEC SQL FETCH CU INTO :zaehler;
  EXEC SQL CLOSE CU;

  /* Anzahl um 1 erhoehen */
  zaehler.arr[zaehler.len] = '\0';
  Anzahl = atoi((char *)zaehler.arr);
  cout<<"Anzahl =       "<<(char *)zaehler.arr<<endl;
  Anzahl = Anzahl + 1;
  sprintf((char *)zaehler.arr,"%d", Anzahl);
  cout<<"Neue Anzahl =  "<<(char *)zaehler.arr<<endl;
  strcpy((char *)stmt.arr, \
  "UPDATE Material SET Anzahl = :v1 WHERE AbrechnungsID = :v2
                                   AND Bezeichnung = :v3 AND Preis = :v4");
  stmt.len = strlen((char *)stmt.arr);

  /* Neue Anzahl in DB schreiben */
  EXEC SQL PREPARE U FROM :stmt;
  EXEC SQL EXECUTE U USING :zaehler, :materialabrechnungsID, /
                           :materialbezeichnung, :materialpreis;
  EXEC SQL COMMIT RELEASE;
  cout<<"-------------------------------------"<<endl;

  /* Dem FDBS die Material-ID (String) zurueckliefern
     (Tupel: AbrID, Bezeichnung, Preis). */
  sMaterialID.len = /
   strlen(strcpy((char *)sMaterialID.arr, ""));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,(char *)materialabrechnungsID.arr));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,","));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,(char *)materialbezeichnung.arr));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,","));
  sMaterialID.len = /
   strlen(strcat((char *)sMaterialID.arr,(char *)materialpreis.arr));

  pTheOp->sLoid = (char *)sMaterialID.arr;

  } /* Ende von: ELSEIF Satzanzahl == 1*/

else if (SatzAnzahl >= 1)
  {/* Inkonsistenter Zustand, Fehler ausgeben.*/
    cout<<endl<<"Es existieren mehrere gleiche Materialien zu einer/
```

```
                   Abrechnung!!! FEHLER !!!"<<endl<<endl;
      EXEC SQL ROLLBACK RELEASE;
      return false;
    } /* Ende von: ELDEIF SatzAnzahl >= 1 */



  } /* Ende von: IF result */

  else
    {
      /* Holen der Daten nicht erfolgreich, Fehler ausgeben. */
      cout<<endl<<"Fehler beim holen der Daten via ORB!"<<endl;
      return false;
    } /* ELSE result */


      return true;
   }/* ende Handle */
};



void SendAttach( const string i_sFdbs, const string i_sAdapter )
{
  /* ttach-Operation fuer FDBS vorbereiten, */
   CStandardOp oAttachOp( ID_ATTACH );
   oAttachOp.sSender = i_sAdapter;

   /* Operation an FDBS verschicken */
   CSender oSender;
   oSender.Init( i_sFdbs );
   oSender.Send( &oAttachOp );
}


/************************************************************
 Hauptprogramm
************************************************************/
int main( int argc, char* argv[] )
{

  /* falsche Argumentenzahl abfangen */
  /* richtig ist: ProgrammName, FDBS-Name, Adapter-Name */
   if( argc != 3 ){
     cout << "Argumente: FDBS-Name, Adapter-Name"<<endln;
      return 1;
   }

   string sFdbs =    argv[ 1 ];
   string sAdapter = argv[ 2 ];


   /* Operationen registrieren */
   BEGIN_REGISTRATION;
      REGISTER_OPERATION( CStandardOp( ID_SHUTDOWN ), /
      CShutdownHandler( sFdbs ) );
      REGISTER_OPERATION( CInsert, CInsertHandler );
   END_REGISTRATION;
```

```
   /* beim Foederierungssystem anmelden */
   SendAttach( sFdbs, sAdapter );

   /* Operationsempfang starten */
   CReceiver* poRec = CReceiver::Instance();
   if( !poRec->Init( sAdapter ) ){
      return 1;
   }

   poRec->WaitForOperation();

   /* Aufraeumarbeiten nach dem Herunterfahren */
   poRec->Terminate();
   CReceiver::Destroy();

   return 0;
}



/**********************************************************
 Error-Handler Routine fuer Oralce
**********************************************************/
void sql_error(char *msg)
{
/* This is the ORACLE error handler.
 * Print diagnostic text containing error message,
 * current SQL statement, and location of error.
 */
    printf("\n%s", msg);
    printf("\n%.*s\n",
        sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in \"%.*s...\"\n",
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("on line %d of %.*s.\n\n",
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

22.3.2 Der ExportAdapter

Der Exportadapter besteht aus der Datei AbrSys_ExportHandler.pc.

```
/* -*-c++-*- */
/*
 * Name:   AbrSys_ExportHandler.pc
 *
 * Typ:    PRO*C SourceDatei - Muss erst durch den Precompiler laufen!
 *
 * Autor:  Mischa Lohweber
 *
 * Beschreibung: Die Datei liest die Oracle Pipe aus und
```

```
 *               versendet die erhaltenen Daten an das FDBS.
 *
 */


#include <rw/cstring.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <sqlca.h>
#include <oraca.h>
#include <string.h>
#include <comi.H>
#include <pg290.H>
//#include "iso290.h"

EXEC ORACLE OPTION (ORACA=YES);
EXEC SQL INCLUDE SQLCA;

EXEC SQL BEGIN DECLARE SECTION;
#define     UNAME_LEN      20
#define     PWD_LEN        40
#define     EINGABE_LEN    100
#define     SQL_LEN        400
#define     MAX_LEN        2000

VARCHAR  username[UNAME_LEN];
varchar  password[PWD_LEN];

VARCHAR  Vop[MAX_LEN];
VARCHAR  Vtabelle[MAX_LEN];
int      Vid;
VARCHAR  Vname[MAX_LEN];
VARCHAR  Vvorname[MAX_LEN];
VARCHAR  Vgebdat[MAX_LEN];
VARCHAR  Vgeschl[MAX_LEN];
VARCHAR  Vstrasse[MAX_LEN];
VARCHAR  Vhausnr[MAX_LEN];
int      Vplz;
VARCHAR  Vort[MAX_LEN];
VARCHAR  Vland[MAX_LEN];
VARCHAR  Vtelefon[MAX_LEN];
int      Vabrechnungsid;
int      Vpatientenid;
VARCHAR  Vaufnahmedatum[MAX_LEN];
int      status;
VARCHAR  VPATIENT[MAX_LEN];
VARCHAR  VsPID[EINGABE_LEN];
VARCHAR  Vtelefax[MAX_LEN];
VARCHAR  Vkrankenkasse[MAX_LEN];
VARCHAR  Vstaat[MAX_LEN];
EXEC SQL END DECLARE SECTION;

int  Anzahl;
int  SatzAnzahl;
int  n;
CSender *poFDBS;
char *sProg;
char sPatientID[100];
```

```
char sPLZ[20];
char sAbrechnungsID[100];

/* Declare Oracle error handling function. */
void sql_error(char *msg);


//-------------------------------------------------------------------
//  Das Hauptprogramm
//  Returnwert: int
//-------------------------------------------------------------------

int main(int argc, char **argv)
{
  if( argc != 2)
  {
    cout << "Usage: " << argv[0] << " <FDBS-name>" << endl;
    return( 1);
  }

  sProg = argv[0];
  // Adapter initialisieren

  poFDBS = new CSender;
  if( !poFDBS->Init( argv[1] ) )
  {
    cerr << sProg << ": Error while initializing adapter!" << endl;
    delete poFDBS;
    exit(1);
  }

  cout << argv[0] << ": DB-Agent laeuft!" << endl;

  /* DB initialisieren */
  /* Benutzernamen und Passwort festlegen */
  username.len = strlen(strcpy((char *)username.arr, "PG290"));
  password.len = strlen(strcpy((char *)password.arr, "PG290"));

  /* Fehlerbehandlungsroutine festlegen, wenn in einem SQL Statement */
  /* ein Fehler auftritt und die Fehlermeldung merken.*/
  EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle error");
  oraca.orastxtf = ORASTFERR;

  /* Anmeldung */
  EXEC SQL CONNECT :username IDENTIFIED BY :password;
  cout<<endl<<endl<<"Verbindung zu Oracle aufgebaut."<<endl;
  cout<<endl<<"Warte auf Pipe...."<<endl;

  /* InsertObjekt definieren */
  CInsert* poInsOp;
  //poInsOp = new CInsert;

  while( TRUE )
  {

       /* Deklaration von PL/SQL Variablen zum auslesen der Pipe */
    EXEC SQL EXECUTE
      DECLARE
       stat    INTEGER;
```

```
 op      VARCHAR2(2000);
 tabelle VARCHAR2(2000);
 id      NUMBER;
 name    VARCHAR2(2000);
 vorname VARCHAR2(2000);
 gebdat  VARCHAR2(2000);
 geschl  VARCHAR2(2000);
 strasse VARCHAR2(2000);
 hausnr  VARCHAR2(2000);
 plz     NUMBER;
 ort     VARCHAR2(2000);
 land    VARCHAR2(2000);
 telefon VARCHAR2(2000);
 telefax VARCHAR2(2000);
 staat   VARCHAR2(2000);
 krankenkasse VARCHAR2(2000);
 abrechnungsid NUMBER;
 patientenid   NUMBER;
 aufnahmedatum VARCHAR2(2000);
BEGIN
 /* Variablen fuer den Trigger der Patientenstammdaten */
 tabelle := '';
 op := '';
 id := 0;
 name := '';
 vorname := '';
 geschl  := '';
 strasse := '';
 hausnr  := '';
 plz     := 0;
 ort     := '';
 land    := '';
 telefon := '';
 telefax := '';
 staat   := '';
 krankenkasse := '';

 /* Variablen fuer den Trigger der Abrechnungsdaten */
 abrechnungsid := 0;
 patientenid   := 0;
 aufnahmedatum := '';

 /* Auf PIPE warten und dann auslesen. RECEIVE_MESSAGE wartet
    3 Jahre auf Antwort */
 stat := dbms_pipe.receive_message('FDBS_PIPE');

 /* Status der Uebertragung pruefen */
 IF stat = 0 Then
   /* Status ist OK, Message entpacken und Werte
      den HOST Variablen zuweisen */
   dbms_pipe.unpack_message(tabelle);
   dbms_pipe.unpack_message(op);
   IF tabelle = 'PATIENTDAT' Then
     dbms_pipe.unpack_message(id);
     dbms_pipe.unpack_message(name);
     dbms_pipe.unpack_message(vorname);
     dbms_pipe.unpack_message(gebdat);
     dbms_pipe.unpack_message(geschl);
     dbms_pipe.unpack_message(strasse);
```

```
dbms_pipe.unpack_message(hausnr);
dbms_pipe.unpack_message(plz);
dbms_pipe.unpack_message(ort);
dbms_pipe.unpack_message(land);
dbms_pipe.unpack_message(telefon);
dbms_pipe.unpack_message(telefax);
dbms_pipe.unpack_message(staat);
dbms_pipe.unpack_message(krankenkasse);

/* Dieser Werte sind definitionsgemaess immer
   NOT NULL, daher keine Ueberpruefung */
:status := stat;
:Vtabelle := tabelle;
:Vop := op;
:Vid := id;
:Vname := name;
:Vvorname := vorname;
:Vgebdat := gebdat;

/* Ueberpruefung der Werte auf NOT NULL */
IF geschl IS NOT NULL THEN
  :Vgeschl := geschl;
ELSE
  :Vgeschl :=  'm';
END IF;

IF strasse IS NOT NULL THEN
  :Vstrasse := strasse;
ELSE
  :Vstrasse :=  ' ';
END IF;

IF hausnr IS NOT NULL THEN
  :Vhausnr := hausnr;
ELSE
   :Vhausnr := ' ';
END IF;

IF plz Is NOT NULL THEN
  :Vplz := plz;
ELSE
   :Vplz := 0;
END IF;

IF ort IS NOT NULL THEN
  :Vort := ort;
ELSE
  :Vort := ' ';
END IF;

IF land IS NOT NULL THEN
  :Vland := land;
ELSE
  :Vland := ' ';
END IF;

IF telefon IS NOT NULL THEN
  :Vtelefon := telefon;
ELSE
```

```
        :Vtelefon := ' ';
      END IF;

      IF telefax IS NOT NULL THEN
        :Vtelefax := telefax;
      ELSE
         :Vtelefax := ' ';
      END IF;

      IF staat IS NOT NULL THEN
        :Vstaat := staat;
      ELSE
          :Vstaat := ' ';
      END IF;

      IF  krankenkasse IS NOT NULL THEN
        :Vkrankenkasse := krankenkasse;
      ELSE
        :Vkrankenkasse :=  ' ';
      END IF;

    ELSE
      IF tabelle = 'ABRECHNUNG' THEN
       /* Daten aus der Abrechnungstabelle, restliche Messages auslesen */
       dbms_pipe.unpack_message(abrechnungsid);
       dbms_pipe.unpack_message(patientenid);
       dbms_pipe.unpack_message(aufnahmedatum);

       /* Dieser Werte sind definitionsgemaess immer
          NOT NULL, daher keine Ueberpruefung */
       :status := stat;
       :Vtabelle := tabelle;
       :Vop := op;
       :Vabrechnungsid := abrechnungsid;
       :Vpatientenid := patientenid;
       :Vaufnahmedatum := aufnahmedatum;
      END IF; /* von Abrechnung */

    END IF; /* von PATIENT */

  END IF; /* Von Stat */


 END; /* Von BEGIN */
END-EXEC; /* Ende des PL/SQL Statementes */

if (status == 0)
  {
   Vtabelle.arr[Vtabelle.len] = '\0';
   Vop.arr[Vop.len] = '\0';
   VPATIENT.len = strlen(strcpy((char *)VPATIENT.arr,"PATIENTDAT"));
   n = strcmp((char *)Vtabelle.arr,(char *)VPATIENT.arr);
   if (n == 0)
     { /* Tabelle PATEINT, Daten ausgeben und versenden */

       /* Status ist OK, den HOST-Variablen die
          Laenge zuweisen und ausgeben */
       Vname.arr[Vname.len] = '\0';
       Vvorname.arr[Vvorname.len] = '\0';
```

```
Vgebdat.arr[Vgebdat.len] = '\0';
Vgeschl.arr[Vgeschl.len] = '\0';
Vstrasse.arr[Vstrasse.len] = '\0';
Vstrasse.len = strlen(strcat((char *)Vstrasse.arr," "));
Vhausnr.arr[Vhausnr.len] = '\0';
Vstrasse.len = /
 strlen(strcat((char *)Vstrasse.arr,(char *)Vhausnr.arr));
Vort.arr[Vort.len] = '\0';
Vland.arr[Vland.len] = '\0';
Vtelefon.arr[Vtelefon.len] = '\0';
Vtelefax.arr[Vtelefax.len] = '\0';
Vkrankenkasse.arr[Vkrankenkasse.len] = '\0';
Vstaat.arr[Vstaat.len] = '\0';
sprintf(sPatientID,"%d",Vid);
sprintf(sPLZ,"%d",Vplz);
if ((char *)Vgeschl.arr == "w")
  {/* Geschlecht weiblich, also 0 */
      Vgeschl.len = strlen(strcpy((char *)Vgeschl.arr, "0"));
  }
else
  {/* Geschlecht maenlich, also 1 */
    Vgeschl.len = strlen(strcpy((char *)Vgeschl.arr, "1"));
  }


cout<<endl<<"-------------------------------------"<<endl;
cout<<"Tabelle:    "<<(char *)Vtabelle.arr<<endl;
cout<<"Operation:  "<<(char *)Vop.arr<<endl;
cout<<"PatientenID="<<sPatientID<<endl;
cout<<"Name=       "<<(char *)Vname.arr<<endl;
cout<<"Vorname=    "<<(char *)Vvorname.arr<<endl;
cout<<"GeburtsDat= "<<(char *)Vgebdat.arr<<endl;
cout<<"Geschlecht= "<<(char *)Vgeschl.arr<<endl;
cout<<"Strasse=    "<<(char *)Vstrasse.arr<<endl;
cout<<"PLZ=        "<<Vplz<<endl;
cout<<"Ort=        "<<(char *)Vort.arr<<endl;
cout<<"Land=       "<<(char *)Vland.arr<<endl;
cout<<"Telefon=    "<<(char *)Vtelefon.arr<<endl;
cout<<"Telefax=     "<<(char *)Vtelefax.arr<<endl;
cout<<"Krankenkasse="<<(char *)Vkrankenkasse.arr<<endl;
cout<<"Staatsangeh.="<<(char *)Vstaat.arr<<endl;
cout<<"-------------------------------------"<<endl;


/* Hier die Operationen zum versenden der Daten */
/* an das Foederierungssystem. */

/* Objekt anlegen */
poInsOp = new CInsert;

poInsOp->sSchema = "CSAbrsys";
poInsOp->sClass = "Patient";
poInsOp->sLoid = sPatientID;

poInsOp->AddAttribute("Name",(char *)Vname.arr);
poInsOp->AddAttribute("Vorname",(char *)Vvorname.arr);
poInsOp->AddAttribute("Geburtsdatum",(char *)Vgebdat.arr);
poInsOp->AddAttribute("PLZ",sPLZ);
poInsOp->AddAttribute("Ort",(char *)Vort.arr);
```

```
            poInsOp->AddAttribute("Land",(char *)Vland.arr);
            poInsOp->AddAttribute("Staatsangehoerigkeit",(char *)Vstaat.arr);
            poInsOp->AddAttribute("Geschlecht",(char *)Vgeschl.arr);
            poInsOp->AddAttribute("Telefon",(char *)Vtelefon.arr);
            poInsOp->AddAttribute("Telefax",(char *)Vtelefax.arr);
            poInsOp->AddAttribute("Krankenkasse",(char *)Vkrankenkasse.arr);

            /* Objekt an das FDBS senden */
            poFDBS->Send(poInsOp);

            /* Objekt wieder loeschen */
            delete poInsOp;

          } /* von if (Vtabelle = 'PATIENT') */

        else
          {
            /* Status ist OK, den HOST-Variablen die Laenge
               zuweisen und ausgeben */
            Vaufnahmedatum.arr[Vaufnahmedatum.len] = '\0';
            sprintf(sPatientID,"%d",Vpatientenid);
            sprintf(sAbrechnungsID,"%d",Vabrechnungsid);

            cout<<endl<<"--------------------------------------"<<endl;
            cout<<"Tabelle:       "<<(char *)Vtabelle.arr<<endl;
            cout<<"Operation:     "<<(char *)Vop.arr<<endl;
            cout<<"AbrechnungsID ="<<sAbrechnungsID<<endl;
            cout<<"PatientenID=   "<<sPatientID<<endl;
            cout<<"AufnahmeDatum= "<<(char *)Vaufnahmedatum.arr<<endl;
            cout<<"Aggr.:=        "<<sPatientID<<endl;
            cout<<"--------------------------------------"<<endl;

            /* Hier die Operationen zum versenden der Daten */
            /* an das Foederierungssystem. */
            poInsOp = new CInsert;

            poInsOp->sSchema = "CSAbrsys";
            poInsOp->sClass = "Abrechnungsdaten";
            poInsOp->sLoid = sAbrechnungsID;
            poInsOp->sAggregateOid = sPatientID;

            poInsOp->AddAttribute("PatientenID",sPatientID);

            poFDBS->Send(poInsOp);

            delete poInsOp;

          } /* von else (Vtabelle = 'PATIENT') */
        }
      else
        {
          printf("\nFehler beim holen der Message!!\n");
        }

}

/* Wird nicht erreicht */

poFDBS->Terminate();
```

```
  delete poFDBS;

  return( 0 );
}

/**********************************************************
 Error-Handler Routine fuer Oralce
**********************************************************/
void sql_error(char *msg)
{
/* This is the ORACLE error handler.
 * Print diagnostic text containing error message,
 * current SQL statement, and location of error.
 */
    printf("\n%s", msg);
    printf("\n%.*s\n",
        sqlca.sqlerrm.sqlerrml, sqlca.sqlerrm.sqlerrmc);
    printf("in \"%.*s...\"\n",
        oraca.orastxt.orastxtl, oraca.orastxt.orastxtc);
    printf("on line %d of %.*s.\n\n",
        oraca.oraslnr, oraca.orasfnm.orasfnml,
        oraca.orasfnm.orasfnmc);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    exit(1);
}
```

22.3.3 Trigger und Pipes

Die Definitionen der Trigger und Pipes bestehen aus den Dateien `CreateTrigger_Paientenstammdaten.sql` und `CreateTrigger_Abrechnungsdaten.sql`. Diese Dateien enthalten SQL Code und können unter `SQL-Plus` mit bem Befehl `start <Name>` ausgeführt werden.

Die Datei `CreateTrigger_Paientenstammdaten.sql`**:**

```
CREATE OR REPLACE TRIGGER Hole_Patientendaten
AFTER INSERT OR UPDATE ON Patientenstammdaten
FOR EACH ROW
DECLARE
  status NUMBER;
  op     VARCHAR2(20);
BEGIN

  IF INSERTING THEN
        dbms_output.put_line('Patienten_Trigger: /
                             Insert-Operation wurde ausgefuehrt.');
        op := 'INSERT';
  END IF;

  IF UPDATING THEN
        dbms_output.put_line('Patienten_Trigger: /
                             Update-Operation wurde ausgefuehrt.');
        op := 'UPDATE';
  END IF;

  dbms_pipe.pack_message('PATIENTDAT');
  dbms_pipe.pack_message(op);
  dbms_pipe.pack_message(:new.patientenid);
  dbms_pipe.pack_message(:new.name);
```

```
  dbms_pipe.pack_message(:new.vorname);
  dbms_pipe.pack_message(TO_CHAR(:new.Geburtsdatum,'DD.MM.YY'));
  dbms_pipe.pack_message(:new.Geschlecht);
  dbms_pipe.pack_message(:new.Strasse);
  dbms_pipe.pack_message(:new.Hausnummer);
  dbms_pipe.pack_message(:new.PLZ);
  dbms_pipe.pack_message(:new.Ort);
  dbms_pipe.pack_message(:new.Land);
  dbms_pipe.pack_message(:new.Telefon);
  dbms_pipe.pack_message(:new.Telefax);
  dbms_pipe.pack_message(:new.Staatsangehoerigkeit);
  dbms_pipe.pack_message(:new.Krankenkasse);

  status := dbms_pipe.send_message('FDBS_PIPE');

  IF status != 0 THEN
        dbms_output.put_line('Patient_Triggerfehler: /
                              Senden der Message nicht gelungen!!');
  END IF;

END;
/
SET SERVEROUTPUT ON

ALTER TRIGGER Hole_Patientendaten ENABLE;

select trigger_type, table_name, triggering_event from user_triggers;
```

Die Datei `CreateTrigger_Abrechnungsdaten.sql`:

```
CREATE OR REPLACE TRIGGER Hole_Abrechnungsdaten
AFTER INSERT OR UPDATE ON Abrechnungsdaten
FOR EACH ROW
DECLARE
  status NUMBER;
  op     VARCHAR2(20);
BEGIN

  IF INSERTING THEN
         dbms_output.put_line('Abrechnugs_Trigger: /
                               Insert-Operation wurde ausgefuehrt.');
        op := 'INSERT';
  END IF;

  IF UPDATING THEN
        dbms_output.put_line('Abrechnungs_Trigger: /
                              Update-Operation wurde ausgefuehrt.');
        op := 'UPDATE';
  END IF;

  dbms_pipe.pack_message('ABRECHNUNG');
  dbms_pipe.pack_message(op);
  dbms_pipe.pack_message(:new.abrechnungsid);
  dbms_pipe.pack_message(:new.patientenid);
  dbms_pipe.pack_message(TO_CHAR(:new.aufnahmedatum,'DD.MM.YY'));

  status := dbms_pipe.send_message('FDBS_PIPE');

  IF status != 0 THEN
```

```
        dbms_output.put_line('Abrechnungs_Triggerfehler: /
                              Senden der Message nicht gelungen!!');
  END IF;

END;
/

SET SERVEROUTPUT ON

ALTER TRIGGER Hole_Abrechnungsdaten enable;

select trigger_type, table_name, trigger_name from user_triggers;
```

Kapitel 23

Das Benutzungshandbuch

Verfasser: Patrick Koehne

In diesem Kapitel wird beschrieben, welche Komponenten des Systems zur Verfügung stehen müssen und wie diese gestartet werden müssen.

23.1 Das Abrechnungssystem

Verfasser: Betül Ilikli

Das Abrechnungssystem wird folgendermaßen gestartet:

- In das .../Abrsys/Forms-Verzeichnis gehen und dort `f45runm` eingeben.
- Es erscheint das `Forms Runform Options`-Fenster.
- Unter **File** `main.fmx` eingeben.
- Bei **Userid:** und bei **Password:** ebenfalls `pg290` eingeben.

Es erscheint dann das `Hauptmenu`-Fenster.

23.2 Das Angiographiesystem

Verfasser: Ulf Radmacher Um das Angiographiesystem zu starten, sind folgende Schritte notwendig: Zuerst muß das Schema geladen werden.

1. `rlogin auf bern`
2. `setenv DISPLAY eigenerRechner:0.0` (auf bern)
3. `xhost bern` (auf eigenerRechner)
4. `module add O2` (auf bern)
5. `o2server -system SYSTEM` (ab hier alles auf bern)
6. `o2 -system SYSTEM` , soll die graphische Entwicklungsumgebung O_2 *Tools* verwendet werden, so muß der Aufruf `o2 -system SYSTEM -toolsgraphic` heißen.
7. Ist die graphische Entwicklungsebene nicht verwendet worden, so erscheint jetzt die `O2Shell`. Eingaben in dieser Shell werden grundsätzlich mit RETURN und CTRL-D abgeschlossen.
8. Einlesen des Skriptes mit # “`skriptname`”. Anstelle des Skriptnamens muß der Pfad zur Datei `angio.schema`, welche sich zur Zeit im Verzeichnis `radmache/docs` befindet, angegeben werden (Empfehlung: kopiert das Skript in das Verzeichnis aus dem Ihr O_2startet. Dann den Aufruf # “`angio.schema`” ausführen). Das Skript muß nur beim ersten Aufruf eingelesen werden, danach ist es gespeichert und kann von Hand aktiviert werden .

9. Die `O2Shell` kann mit `quit` verlassen werden.

Im nächsten Schritt müssen die Klassen bestätigt werden. Hierzu kann man O_2 mit dem Befehl `o2 -server SERVER -system SYSTEM` starten. Anschließend muß das Schema über die entsprechenden Menu-Punkte gesetzt werden. Die Klassen werden nun über `Confirm classes` bestätigt und die Applikation kann gestartet werden.

23.3 Der FDBS-Kern und die Adapter

23.3.1 Starten und Beenden des CHORUS COOL-ORB

Verfasser: Peter Ziesche

Die Kommunikation zwischen den lokalen Datenbank-Adaptern und dem Föderierungssystem erfolgt über CORBA. Deshalb muß vor dem Start des Föderierungssystems der CHORUS COOL-ORB auf allen Rechnern laufen, auf denen eine Komponente (Föderierungssystem oder Adapter) laufen soll.

Um den COOL-ORB hochzufahren sind folgende Schritte in der angegebenen Reihenfolge auszuführen[1]:

1. Starten des *Domain-Managers* auf dem Rechner `bern`

   ```
   $ coolStart -m
   Starting Domain Manager...Done
   Starting Node Manager...Done
   Starting Group Manager...Done
   $
   ```

2. Starten des *Node-Managers* auf jedem Rechner, auf dem ein Adapter oder das Föderierungssystem laufen sollen

   ```
   $ coolStart
   Starting Node Manager...Done
   Starting Group Manager...Done
   $
   ```

Um den COOL-ORB nach Beendigung aller Komponenten des Föderierungssystems wieder herunterzufahren sind folgende Schritte auszuführen:

1. Zum Herunterfahren des COOL-ORB auf einem einzelnen Rechner ist folgender Befehl vorhanden:

   ```
   $ coolStop node
   Shutting down COOL node
   COOL node is going down...
   $
   ```

2. Zum Herunterfahren des gesamten Systems auf allen Rechner muß folgender Befehl auf dem Rechner `bern` ausgeführt werden:

   ```
   $ coolStop domain
   Shutting down COOL domain
   COOL domain is going down...
   $
   ```

Beim Herunterfahren ist zu beachten, daß der Vorgang nicht unbedingt abgeschlossen ist, sobald die Shell den Eingabe-Prompt wieder anzeigt.

[1] Die Angaben beziehen sich auf das LS10-Netz, in einer anderen Umgebung können für den Start andere Schritte erforderlich sein.

23.3.2 Installation und Aufruf des Föderierungssystems

Verfasser: Peter Ziesche

Dieser Abschnitt listet die Voraussetzungen für den Einsatz des Föderierungssystems auf und gibt Anweisungen zum Starten und zum Beenden.

Das Föderierungssystem muß gestartet werden, nachdem der COOL-ORB (siehe Abschnitt 23.3.1) und bevor ein Datenbank-Adapter (siehe Abschnitte 23.3.3 und 23.3.4) gestartet wurden.

Zum Betrieb des Föderierungssystems werden die Programme

- `fdbsmain` (Das Föderierungssystem) und
- `shutdown` (Ein Utility zum Herunterfahren des Föderierungssystems oder lokaler Datenbank-Adapter)

benötigt.

Der Start des Föderierungssystems erfolgt durch das Kommando

```
$ fdbsmain FDBS_NAME
```

`FDBS_NAME` ist ein Parameter und legt den Namen fest, unter dem sich das Föderierungssystem beim COOL-ORB anmeldet. Alle Datenbank-Adapter müssen diesen Namen angeben, wenn sie mit dem Föderierungssystem kommunizieren wollen.

Das Herunterfahren des Föderierungssystems geschieht durch das Kommando

```
$ shutdown FDBS_NAME
```

`FDBS_NAME` ist dabei der Name, der dem Föderierungssystem beim Start zugewiesen wurde. Das Föderierungssystem sorgt für das korrekte Beenden aller angeschlossenen Datenbank-Adapter, bevor es selbst endet.

Das `shutdown`-Kommando kann auch benutzt werden, um einen lokalen Datenbank-Adapter herunter zu fahren. Es ist dann jeweils dessen Name als Parameter anzugeben.

23.3.3 Installation und Aufruf des Oracle-Adapters

Verfasser: Mischa Lohweber

Dieser Abschnitt listet die Voraussetzungen für den Einsatz des Oracle-Adapters auf und gibt Anweisungen zum Start und zum Beenden des Adapters.

Voraussetzungen:

1. Das Abrechnugssystem muß installiert sein. Das heißt, daß die Oracle-Datenbank laufen muß, der Benutzer `PG290` eingerichtet und mit dem Passwort `PG290` versehen sein muß.
2. Der Benutzer `PG290` muß die Rechte besitzen, in der Oracle-Datenbank Trigger anzulegen und Pipes zu benutzen (Einbinden der Pakete `DBMS_OUTPUT` und `DBMS_PIPE`).
3. Das FDBS-System muß gestartet sein (siehe Abschnitt 23.3.2).
4. Die PL/SQL-Programme `CreateTrigger_Patientenstammdaten.sql` und `CreateTrigger_Abrechnungsdaten.sql` müßen einmal vor dem Start der Adapter in einer interaktiven Shell, wie z.B. `SQLPlus`, gestartet werden, um die Trigger einzurichten.

Starten der Adapter:

Der InsertHandler hat die Syntax `AbrSys_InsertHandler <FDBS-Name> <Adapter-Name>` . Der Aufruf hat dann die Form `AbrSys_InserHandler FDBS Abrsys`.
Der ExportHandler hat die Syntax `AbrSys_ExportHandler <FDBS-Name>`. Der Aufruf hat dann die Form `AbrSys_ExportHandler FDBS`.

Beenden der Adapter:

Der InsertHandler wird vom FDBS benachrichtig, wenn dieses korrekt beendet wurde und beendet sich selbst. Der ExportHandler muß in dieser Version noch mit `CTRL-C` beendet werden. Alle Datensätze, die bis zu diesem Zeitpunkt noch nicht ausgelesen wurden, bleiben in der `DBMS-Pipe` vorhanden und werden beim nächsten Start des Adapters ausgelesen.

23.3.4 Installation und Aufruf des O_2-Adapters

Verfasser: Sven Gerding

Der Agent für das Angiographiesystem besteht aus den Programmen `angio_in` - dem Import-Adapter, der Objekte vom Föderierungssystem empfängt - und `angio_out` - der Export-Adapter zur Übergabe von Objekten an das Föderierungssystem.

Der Start der Adapter geschieht einfach mittels

- `angio_in <FDBS-Name> <Adapter-Name> <Base-Name>`, bzw.
- `angio_out <FDBS-Name>`.

Als Parameter sind für `<FDBS-Name>` `FDBS`, für `<Adapter-Name>` `CSAngio` und für `<Base-Name>` `angio` anzugeben. `<Base-Name>` muß mit der aktuell vom Angiographiesystem verwendeten Base übereinstimmen, da sich ansonsten garantiert Inkonsistenzen im Datenbestand von Angiographiesystem und Abrechnungssystem ergeben!

Abgesehen davon sind lediglich folgende zwei Bedingungen zu beachten:

1. Das Angiographiesystem sollte laufen, bevor `angio_out` gestartet wird.
2. Der Export-Adapter und das Angiographiesystem müssen auf dem selben Rechner laufen, da sonst die Kommunikation von Adapter und Applikation, die über eine Pipe läuft, nicht funktioniert.

Der Import-Adapter wird automatisch gestoppt, wenn das Föderierungssystem heruntergefahren wird. Der Export-Adapter muß jedoch von Hand mittels `Crtl-C`, oder falls er im Hindergrund läuft, mit `kill <PID von angi_out>` beendet werden.

Teil VII

Resumee

Kapitel 24

Die eingesetzten Werkzeuge

Verfasser: Patrick Koehne

In diesem Kapitel werden die eingesetzten Werkzeuge vorgestellt, eventuell Erfahrungen berichtet, Vor- und Nachteile gegenübergestellt und eventuell eine Bewertung zu ihrer Projekttauglichkeit abgegeben.

24.1 CME

Verfasser: Mischa Lohweber

Eine genauere Beschreibung was CME ist und was es kann ist in Kapitel 1.20 zu finden. Die Erfahrung in Rahmen der PG Arbeit haben gezeigt, daß das Werkzeug CME nicht in dem Maße eingesetzt worden ist, wie es anfänglich den Anschein hatte. Die technischen Probleme waren so hoch, daß das Werkzeug nur von wenigen und erst in den letzten Wochen der PG eingesetzt worden ist. Die Probleme bezogen sich hauptsächlich auf die Einrichtung der verschiedenen Fehlerkomponenten und deren Verantwortlichen und die Einrichtung bei den jeweiligen Benutzern der PG. Auf einigen Arbeitsplätzen konnte CME z. B. nicht gestartet werden. Auf den Arbeitsplätzen wo CME lief, stellt das Programm eine gute Unterstützung der Test- und Korrekturphase dar. Durch die anfängliche Fehleranfälligkeit traf CME allerdings nicht auf sehr viel Akzeptanz bei den PG-Mitgliedern, so daß wie oben schon gesagt das Programm nur von wenigen und erst in der letzten Phase der PG eingesetzt wurde. Meine eigenen Erfahrungen mit CME am Fraunhofer Institut für Software und Systemtechnik ([ISS]) sind positiver. Dort wird das Programm seit längerem eingesetzt und stellt ein nützliches Tool bei der Entwicklung und Fehlerbehebung dar. Allerdings läuft es dort auch seit längerer Zeit stabil.

24.2 CVS

Verfasser: Patrick Koehne

Das Versionskontrollsystem CVS hat eigentlich die ganze Zeit durch die Implementierungsphase immer wieder Fragen aufgeworfen. Dies bedeutet nun nicht, daß CVS im ganzen schlecht ist, nein, es ist nur nicht so einfach wie RCS zubedienen. Der Vorteil des System ist sicherlich der, daß jeder Benutzer eine eigene Version aller von ihm benutzten Dateien bei sich im Home-Verzeichnis zur Verfügung hat. Er ist also somit unabhängig von den anderen Entwicklern. Selbst wenn diese eine neuere Version einbringen, so ist der Benutzer nicht gezwungen mit den neuen Versionen zu arbeiten, sondern kann weiterhin auf seinen "lokalen" Versionen weiterarbeiten.

Die Schwierigkeiten mit der Versionskontrolle lagen hauptsächlich in der Bedienung. Wenn wir es nicht geschafft hätten, die Benutzung mit Hilfe des Emacs zu erreichen, wäre die ganze Sache sicherlich völlig konfus verlaufen. Ein weiterer Nachteil liegt natürlich darin, daß jeder Benutzer eine physikalisch Kopie des Verzeichnisbaumes bei sich im privaten Home-Verzeichnis liegen hat. Dies bedeutet, daß bei großen Projekten, an denen auch noch viele Leute mitarbeiten, entsprechend viel Plattenkapazität vorhanden sein muß. Am Ende der PG, als viele ausführbare Programme generiert wurden und dies auch noch von mehreren Leuten, waren unsere Plattenkapazitäten, die bis dahin als unerschöpflich gehalten wurden, schnell völlig ausgeschöpft.

Alles in allem denke ich jedoch schon, daß es mehr Sinn gemacht hat für die Implementierung CVS statt RCS zu verwenden. Dies begründet sich vor allem durch die Benutzung mit vielen, unabhängig arbeitenden Entwicklern. Dies wäre vermutlich mit RCS doch nicht so einfach möglich gewesen.

24.3 Latex

Verfasser: Patrick Koehne

Da ich mich ja nun wohl im Laufe des Projektes am meisten mit Latex habe herumschlagen dürfen, hier nun meine kurze Anmerkung:

Ich bin froh, daß wir Latex als Textsatzprogramm für die Dokumentation benutzt haben. Da die Berichte von mehreren Leuten gleichzeitig bearbeitet und erstellt worden sind, hätten wir wohl mit anderen Programmen erhebliche Probleme gehabt, auch wenn immer wieder Leute berichten, daß auch andere Software das Multi-Benutzer-Entwickeln unterstützen. Mit zusätzlicher Hilfe der Versionskontrolle RCS hatten wir zu keinem Zeitpunkt Probleme bei der Bearbeitung der Dateien, selbst wenn an ein und derselben Datei mehrere Leute fast zeitgleich geschrieben haben. Ich denke nur durch die feine Aufsplittung in sehr viele kleine Abschnitte (jedes Kapitel und jede Section hat ihre eigene Datei) und durch die Übertragung der Kapitelhierachie in die Verzeichnishierachie war es uns auf so einfache Art und Weise möglich, effizient an der Dokumentation zu arbeiten.

Durch die mir vor der PG nicht bekannten Befehle `include` und `includeonly` ist es mit Latex sehr einfach möglich, eine sehr übersichtliche Struktur in die Dokumentation und in die Datei-/Verzeichnisstruktur zu bringen. Zwar war die Einrichtung derartiger Strukturen zu Beginn nicht gerade einfach, aber zur zweiten Hälfte der PG hin kam die Routine durch und es gab in dieser Richtung mit Latex eigentlich nie mehr Probleme.

Alles in allem bin ich nun, nochmehr als vorher, Befürworter für Latex als Dokumentationstool!!

24.4 Oracle

Verfasser: Xi Gao, K.-H. Schulte

24.4.1 Die Datenbank

Im Rahmen unseres Projektes hat sich die Datenbank als sehr zuverlässig und stabil erwiesen. Jedoch ist auch zu berücksichtigen, daß der zugrundeliegende Datenbestand lediglich auf ca. 2000 Datensätze beschränkt war, welche sich auf sieben Tabellen verteilt haben.

Diese relationale Datenbank hat sich als sehr geeignet für die Modellierung von Verwaltungssystemen erwiesen. Die automatische Sicherstellung der Einhaltung der Integritätsbedingungen war stets gewährleistet.

24.4.2 *Oracle-Forms*

Während unserer Entwicklungszeit mit *Oracle-Forms* haben wir etliche Erfahrungen mit diesem Tool sammeln können. Diese sollen im folgenden, gegliedert in positive und negative Aspekte, aufgeführt werden.

positive Aspekte

- Das Tool eignet sich gut zum schnellen Erstellen von Oberflächen, solange die Tabellenstruktur mit der Struktur der grafischen Oberfläche übereinstimmt.
- Insbesondere eignet sich das Tool gut zum Erstellen von Prototypen und Machbarkeitsstudien. Sollen solche Prototypen dann endgültig realisiert werden, so stünde eine Neuimplementierung an, zum Beispiel mit der C++ Anbindung.
- Zur Anwendungsentwicklung ist das Tool immer dann geeignet, wenn in den grafischen Masken jeweils eine Tabellenzeile komplett angezeigt oder auch verändert werden soll. Dann wird der Entwickler auch sinnvoll durch eine automatische Transaktionskontrolle unterstützt.

negative Aspekte

- Das Entwicklungstool hat sich als extrem instabil erwiesen. Selbst bei Farbänderungen des Hintergrundes ist das gesamte Tool häufig abgestürzt.
- Die Online-Dokumentation ist sehr unübersichtlich und bietet dem Entwickler keinen Gesamtüberblick. Desweiteren war keine hilfreiche Stichwortsuche möglich. Darüber hinaus hat sich das Konzept der Online-Dokumentation in den Augen der Entwickler als unbrauchbar herausgestellt.
- Auch die Online-Dokumentation ist sehr häufig abgestürzt.
- Das Tool eignet sich nicht zur Erstellung von Masken, welche sich auf Einträge aus verschiedenen Tabellen beziehen. Auch wenn diese Einträge durch Schlüsselabhängigkeiten verbunden sind, ist *Oracle Forms* nicht in der Lage diese in einem Fenster gemeinsam zu bearbeiten.
- Sinnvoll ist nur dann mit dem Tool zu arbeiten, wenn bereits die Tabellen speziell für die Bearbeitung mit diesem Werkzeug erstellt worden sind. Das heißt, die Architektur des Systems muß gemäß der grafischen Oberfläche erstellt werden.
- Das Tool erweist sich immer dann als unpraktisch, wenn viel Funktionalität zu implementieren ist. Die PL/SQL-Umgebung bietet dafür nicht genügend Strukturierungsmöglichkeiten.
- Ein grosser Nachteil ist, daß keinerlei Sicht auf das Gesamtsystem vorhanden ist. Die Gesamtarchitektur des Systems ist nicht erkennbar, weder in grafischer Form, noch konkret in Quellcode-Form. Wenn sich ein Projektfremder in ein vorhandenes Projekt einarbeiten müßte, so hätte er dazu keinerlei Hilfsmittel. Es gibt keine Möglichkeit das Projekt systematisch kennenzulernen. (Bei einem C-Programm kann er immerhin noch den Quellcode lesen und Reengineering betreiben.) Oracle-Forms bietet nicht einmal die Möglichkeit, die einzelnen Quellcodestücke aus den Triggern komfortabel auszudrucken. Selbst das muß über ein Kopieren in die Zwischenablage geschehen!

24.5 Rose

Verfasser: Peter Ziesche

In der Entwurfsphase einiger Komponenten wurde Rational Rose dazu benutzt Diagramme zu zeichnen, um die Entwurfsideen grafisch zu dokumentieren. Diese Diagramme wurden später als EPS-Dateien in die Dokumentation der Projektgruppe eingebunden. Eine Generierung von Code-Rahmen durch Rose erfolgte nicht. Die folgenden Aussagen beziehen sich auf die Demoversion von Rose 4.0, wie Rational sie über das WWW zum Herunterladen anbietet. Da immer nur Einzelteile des Projektes eingegeben wurden, erwies sich die Beschränkung der Projektgröße in der Demoversion nicht als Hindernis.

Die Bedienung von Rose ist intuitiv und einfach. Die Gestaltung der Oberfläche lehnt sich sehr eng an Microsoft Visual C++ 4.0 an. Ein im Umgang mit Windows-Anwendungen versierter Benutzer benötigt die Online-Hilfe so gut wie nie.

Die Gestaltung von Diagrammen ist einfach und schnell möglich. Rose übernimmt das Layout von Beziehungen zwischen Klassen automatisch. Änderungen sind zwar manchmal nötig, jedoch sehr einfach und schnell durchführbar.

Das Einfügen von Methoden und Attributen in bestehende Klassen ist etwas umständlich. Für jede Methode und jedes Attribut öffnet sich ein neuer Dialog. Soll eine Methode Parameter erhalten, so kommt ein weiterer Dialog ins Spiel. Da sich immer nur wenige Dialoge gleichzeitig auf dem Bildschirm befinden, bleibt die Übersichtlichkeit erhalten. Das Verfahren ist allerding sehr zeitaufwendig. Insbesondere wenn nur Namen von Methoden und Attributen eingegeben werden sollen, stört der Aufruf der Dialoge. Eine Darstellung und Editiermöglichkeit in einer Baumstruktur wäre wünschenswert.

Ein großer Nachteil ist die unvollständige Unterstützung der Notationen. Innerhalb der PG wurden Diagramme in der UML-Notation erstellt. Rose stellt längst nicht alle Bestandteile der Notation graphisch dar, obwohl sie fester Bestandteil der Notation sind und auch textuell eingegeben werden können. Dazu zählen z.B. die Pfeile, die die Richtung einer Assoziation angeben. Ferner erlaubt die UML-Notation eine *null-bis-viele*-Kardinalität durch die Symbole "`*`" oder "`0..*`" darzustellen. Rose unterstützt nur die zweite Schreibweise automatisch, die in umfangreichen Diagrammen jedoch viel Platz benötigt und daher nicht gerade zu mehr Übersichtlichkeit führt.

Wenn diese Fehler behoben sind, kann Rose als Entwurfswerkzeug aber sehr empfohlen werden. Programmabstürze traten während der gesamten Entwicklungszeit nicht auf. Die Online-Hilfe gibt ausführlich Auskunft über alle Programmfunktionen und liefert auch kurze Infos zu den Bestandteilen der unterstützten Notationen.

Es wäre schön gewesen, mit Rose direkt unter UNIX arbeiten zu können. Dadurch wäre die etwas umständliche Übertragung der Dokumente vom PC auf die Workstation entfallen. Die Demoversion gibt es jedoch nur für Windows, so daß dieser Aufwand hingenommen werden mußte.

24.6 XFig

Verfasser: Patrick Koehne

Dieses Tool hat sich wohl als das Allroundwerkzeug, als Tool für alles herausgestellt. Nicht umsonst kam in mehreren PG-Sitzungen, zum Spaß aller, immer mal wieder der Kommentar "Nimm doch XFIG.".

Für das Erstellen einfacher Schaubilder, Tabellen und anderer Präsentationsdiagramme hat sich XFIG als das beste Tool herausgestellt, weil es einfach zu bedienen ist und ziemlich stabil arbeitet. Vor allem vor dem Hintergrund, daß die meisten Schaubilder anschließend in Latex eingebunden werden mußten war das arbeiten mit XFIG besonders effizient, denn aus XFIG heraus kann sofort eine EPS-Datei exportiert werden, die anschließend 1:1 in Latex übernommen werden kann.

Alles in allem: Latex & XFIG - ein tolles Paar!

24.7 Corba

Verfasser: Peter Ziesche

Im Rahmen der Projektgruppe wurde Corba, d.h. der CHORUS COOL ORB, für die Kommunikation zwischen einzelnen Prozessen des Systems benutzt. Die folgende Bewertung des Produktes erfolgt unter der folgenden Einschränkung: Das von der PG entwickelte System ist kein gutes Beispiel für verteilte Objekte. Da die Schnittstellen und die Struktur der zu übertragenden Daten am Anfang noch nicht bekannt waren, wurde eine Kommunikations-Schnittstelle entwickelt, die (Datenbank-) Operationen in Objekte verkapselt. Diese Schnittstelle wird in Kapitel 18 ausführlich dokumentiert. Der Einsatz dieser Schnittstelle hat zur Folge, daß alle Corba-Objekte die gleiche, nur aus einer Methode bestehende Schnittstelle haben. Die Möglichkeiten des COOL ORB wurden dadurch bei weitem nicht ausgenutzt und die Philosophie verteilter Objekte wurde zu einem einfachen RPC-Mechanismus umfunktioniert. Deshalb sind die folgenden Anmerkungen zu dem Produkt für eine "echte" verteilte, objektorientierte Anwendung unter Umständen nicht aussagekräftig.

Zu Beginn der Programmierung mit dem COOL-ORB waren die Mechanismen mit `_ptr` und `_var` etwas verwirrend. Insbesondere die Wirkung von Konstruktoren und Destruktoren waren nicht ganz einfach zu durchschauen. Für die Entwicklung der Kommunikations-Schnittstelle wurde daher auf die Beispielanwendung der COOL-ORB-Dokumentation zurückgegriffen, die sich recht einfach erweitern und anpassen ließ.

Nachdem diese ersten Hürden jedoch genommen waren, erwies sich der COOL-ORB als ein sehr stabiles und ausgereiftes Produkt. Wurde eine Anwendung erst einmal fehlerfrei übersetzt, so lief sie anschließend *sehr stabil* und *überaus zügig*. Ausführliche Performance-Tests wurden zwar nicht durchgeführt, innerhalb des laufenden Systems erwies sich die Interprozeßkommunikation jedoch nie als System-Bremse.

Die Dokumentation der Programmierschnittstelle des COOL-ORB ist ausführlich genug, um die angebotenen Funktionen nutzen zu können. Gute C++ Kenntnisse sind allerding unbedingt nötig. Die Initialisierung eines Corba-Objektes und die Benutzung des `Naming-Service` erfolgen allerdings in guter alter API-Manier. Von Objektorientierung ist dort nicht soviel zu bemerken. Da dieser Teil sich aber sehr gut abkapseln und später wiederverwenden läßt, kann man das gut verschmerzen.

Eine weitere Bemerkung muß noch zum Betrieb des COOL-ORB gemacht werden. Es war anfangs schwierig festzustellen (weil nicht dokumentiert), daß der Domain-Manager auf einem bestimtmen Rechner gestartet werden muß. Nachdem dieses Problem jedoch einmal gelöst war, lies sich der COOL-ORB über vorgegebene Skript-Dateien sehr handlich bedienen. Beim Herunterfahren von einzelnen Knoten wurden in einigen Fällen Prozesse nicht terminiert. Dies fiel erst beim erneuten Hochfahren dieses Knotens auf. Hier half dann nur noch das manuelle Heraussuchen der Prozeß-ID und ein anschließender "`kill -9`"-Befehl.

Alles in allem ist der CHORUS-COOL-ORB (mit den oben genannten Einschränkungen) zu empfehlen.

24.8 Emacs

Verfasser: Peter Ziesche

Der Emacs ist sicherlich die *eierlegende Wollmilchsau*, als die ihn seine Fans beschreiben. Er diente in fast allen Teilprojekten als *Programmeditor* und war auch das Mittel der Wahl beim Schreiben der *Dokumentation*. Die direkte *Unterstützung der Versionskontrollwerkzeuge* RCS und CVS machten ein geordnetes Arbeiten einfacher und schneller. Seine Fähigkeit zum *Syntax-Highlighting für LATEXund C++* sorgte für mehr Übersicht und weniger Fehler. An die Bedienung über mehrstufige Kommandosequenzen konnte man sich auch gut gewöhnen.

Trotz des *sehr guten Gesamteindrucks*, den der Emacs hinterläßt, sollen ein paar Probleme nicht verschwiegen werden. So muß der Emacs erst einmal zu den oben genannten Fähigkeiten überredet werden. Die Konfiguration ist doch recht umfangreich und kompliziert. Ohne die tatkräfige Unterstützung von Klaus Alfert wäre uns das schwerlich so schnell gelungen. Viele nützliche Kommandos kommen erst durch das Lesen der umfangreichen Dokumentation ans Licht. Hier wäre (UNIX-Fans mögen diesen Satz verzeihen) ein Tip-Assistent à la Microsoft Gold wert.

Mit der eingebauten Programmiersprache haben wir uns in der PG aus Zeitgründen nur oberflächlich befaßt. Die geplante Unterstützung der *iso290-Norm* wurde nicht realisiert. Die über Emacs-Lisp gewonnenen Kenntnisse lassen trotzdem auf ein sehr *mächtiges Werkzeug zur Automatisierung von Arbeitsabläufen* schließen. Schade nur, daß es sich nicht um eine imperative Sprache handelt.

Zum Schluß müssen dem Emacs in den Disziplinen Stabilität und Verläßlichkeit noch einemal sehr gute Noten ausgestellt werden. Andere von uns eingesetzte Werkzeuge haben gezeigt, daß sie hier noch großen Nachholbedarf haben.

24.9 O_2

Durch die Arbeit innerhalb der Projektgruppe haben sich Vor- und Nachteile des O_2-System herausgebildet.

Während der Einarbeitungsaufgabe konnte mit O_2 sehr schnell eine einfache Anwendung mit grafischer Oberfläche erstellt werden, aus der ohne viel Aufwand über die standardmässigen C++-Exportfunktionen ein Adapter für das föderierte System erzeugt werden konnte. Probleme traten mit O_2 auf, als wir versucht haben das von Ingo Ullrich entwickelte Angiographiesystem in das föderierte System zu integrieren. Da es während des Reengineerings häufiger nötig war, Änderungen am Schema der Applikation durchzuführen, musste der Export der veränderten Klassen nach C++ immer wieder neu erfolgen, was mit einem hohen Zeitaufwand verbunden war und eine Versionierung des generierten Codes unmöglich gemacht hat. Des weiteren konnte der Föderierungskern leider nicht in O_2 gespeichert werden, da die im C++-Code verwendeten Templates nicht übernommen werden konnten. Die Anbindung der Adapter úber das C++-Binding an die Datenbank hat gut funktioniert. Problematisch war das Einfügen von Objekten in die Datenbank, da diese Änderungen für das Angiographiesystem erst nach einem Neustart sichtbar werden. Der Grund hierfür ist, daß Veränderungen von Zeigern nur innerhalb einer Session aktualisiert werden. Leider war es uns nicht möglich dieses Problem mit Hilfe der O_2-Handbücher in den Griff zu bekommen. Durch die eigenwillige Implementierung des Angiographiesystem, z.B. die Einbindung von grafischen Aufrufen innerhalb der Initialisierungsmethoden, hat die Integration in das FDBS, bzw. die Fehlersuche, unverhältnismäßig lange gedauert.

24.10 RCS

Verfasser: Patrick Koehne

Hierbei handelt es sich um die Erweiterung des Unix-Standard-Versionskontroll-Systems SCCS. Es ist ein einfaches System, welches die Bearbeitung von Dateien durch mehrere Benutzer ermöglicht und dabei in der Hauptaufgabe sicherstellt, daß nicht mehrere Benutzer zur selben Zeit an der selben Datei arbeiten können. Für Aufagen, die sich auf solche (relativ einfachen) Anforderungen beziehen, ist ein Einsatz von RCS sehr sinnvoll und sehr einfach. Es wird auch noch standardmäßig vom Emacs unterstützt. Unsere komplette Dokumentation, die Unmengen an kleinen Dateien umfaßt, wurde mit RCS kontrolliert und es hat sich sehr gut bewährt! Für einen solchen Einsatz (Dokumentation) wäre sicherlich CVS als Alternative unsinnig gewesen, weil der dadurch verursachte Mehraufwand zu groß gewesen wäre und das Ziel für die Versionskontrolle voll erfüllt war.

Ich denke, daß das Tool unbedingt angewandt werden sollte, sobald irgendetwas entwickelt wird, was von mehreren Personen erarbeitet wird. Ab einer bestimmten Komplexität (z.B. umfangreiche Implementierungen) ist sicherlich über eine Alternative nachzudenken.

24.11 Sniff

Verfasser: Mario Ellebrecht

Als 'Sniff-Administrator' ist es nun an mir, die Frustration zusammenzufassen, die uns dieses Programm beschert hat.

Nach einer schwierigen Einrichtung des Projektes und seiner Konfiguration in Unterprojekte, wurde das Tool von vielen Gruppenmitgliedern angenommen und ausprobiert. Manche haben Sniff+ für die Entwicklung von Teilaufgaben eingesetzt, später jedoch nur noch zur Versionskontrolle. Andere Mitglieder konnten Sniff+ überhaupt nicht konstruktiv verwenden.

Dies hatte verschiedene Gründe, von denen der schwerwiegendste wohl die kaum mögliche Integration der von uns benutzten sonstigen Entwicklungstools war. So war es nicht oder nur schwer möglich, die O_2C- und Pro*C-Precompiler einzubinden, mit denen die Datenbank-APIs benutzt wurden. Obwohl Sniff+ angeblich IDL unterstützt, war es nicht möglich die notwendigen Aufgaben damit zu erledigen. Einige Entwicklungsbereiche, wie z.B. die Erstellung der ORACLE-Forms, schieden ohnehin von der Unterstützung aus. Ein weiterer Grund war die nervige Instabilität der Sniff+-Programme.

Die komplizierte Unterprojekteinrichtung erforderte einen zu großen Zusatzaufwand für die Entwicklung. Als sich im zweiten Semester abzeichnete, daß Sniff+ nur noch zur Versionskontrolle eingesetzt wurde, wurde es durch CVS ersetzt.

Alles in allem kann man wohl sagen, daß sich Sniff bei uns nicht bewährt hat. Die eigentliche Stärke der syntaxgesteuerten Tools wurde nur sehr wenig benutzt.

Kapitel 25

Abschlußbemerkungen

25.1 Die Bemerkungen der Projektgruppenteilnehmer

Verfasser: Mischa Lohweber
Abschließend möchte ich noch etwas zur PG sagen. Im Großen und Ganzen hat mir die Projektgruppe gut gefallen und für mich persönlich viel gebracht. Die Einarbeitung in Oracle, PL-SQL, SQLPlus, OracleForms und die Programmierversuche in C++ fand ich sehr interessant und ich denke, daß ich das Wissen weiterverwenden kann. Die Gruppe hat sich meiner Meinung nach gut verstanden und gut zusammengearbeitet. Auf Fragen bekam man gröstenteils konstruktive Antworten oder Verbesserungsvorschläge und Eigenbrödler habe ich (zum Glück) komplett vermisst. Anfangs hatte ich den Eindruck, daß einige Leute nicht wußten was sie taten, aber dies (mein Eindruck oder die Unwissenheit?!) legte sich mit der Zeit. Ein Lob auch an unsere Betreuer über die Hilfe bei trivialen Dingen (wie z.B. Einrichtung von Emacs, Oracle, CME, etc.) bis hin zu Ideen und Unterstützung bei der Projektarbeit. Die Einführung und die Durchsetzung eines Zeitplans finde ich (besonders im Nachhinein) als gut, da man konkrete Fixpunkte vor Augen hatte, und ansonsten die Dokumentation und das Programm wahrscheinlich immer noch nicht fertig wären. Die Abschlußfahrt nach München zu Siemens fand ich sehr informativ und interesannt. Allerdings war mir persönlich die ganze Hin- und Herfahrerei etwas zu stressig. Bleibt zu hoffen, daß nach Abschluß dieser Zeilen der Schein bereit liegt und die PG290 zu einem zufriedenstellenden Abschluß für alle gebracht worden ist.

Verfasser: Peter Ziesche
Jetzt, wo sich die PG dem Ende zuneigt und ich gerade letzte Hand an der Dokumentation anlege, möchte ich auch noch etwas über meine Eindrücke aus den letzten elf Monaten schreiben. Das wichtigste war meiner Ansicht nach der *Teamgeist*, der sich durch die gesamte Arbeit zog. Sowohl die Teilnehmer, als auch die Betreuer, fühlten sich *für das Projekt als ganzes verantwortlich*; Sprüche wie "Dafür bin ich nicht zuständig, ..." gehörten glücklicherweise nicht zum Umgangston. Ich empfand es außerdem als sehr angenehm, in einem Team von *Nichtrauchern* zu arbeiten. Unsere Betreuer haben uns sehr *engagiert unterstützt* und uns viele Hindernisse schon im Vorfeld aus dem Weg geräumt. Die Aufgabenstellung war sehr interessant und verlor ihren Reiz auch im Verlauf der Arbeit nicht. Ich habe über Datenbanken im allgemeinen und föderierte Datenbanken im besonderen *sehr viel gelernt.* Die Präsentation am Tag der offenen Tür verleihte dem Projekt einen *realistischen Touch* in Form einer unbedingt einzuhaltenden Deadline. Wir konnten alle sehen, wie schwierig es offenbar ist, den *Zeitaufwand für ein Softwareprojekt abzuschätzen.* Dies führte gegen Ende des Projektes zu einer (meiner Meinung nach!) ungleichen Verteilung der Arbeit. Dadurch wird aber der *überaus positive Gesamteindruck*, den ich von der PG mitnehme, nicht getrübt.

25.2 Die Bemerkungen der Projektgruppenleiter

Verfasser: Willi Hasselbring

Besonders interessante Aspekte in der Betreuung dieser Projektgruppe waren die Heterogenität in der Entwicklungsumgebung und die Heterogenität in der Gruppe selbst. Die Vorkenntnisse der einzelnen Teilnehmer(innen) waren sehr unterschiedlich, so daß eine Teamorganisation, in der alle Teilnehmer(innen) vergleichbar zum Erreichen der Ziele beitragen, eine schwere Aufgabe darstellte. Im Endeffekt hatten die einzelnen Teilnehmer(innen) dann auch sehr unterschiedliche Anteile am Erreichen der Projektziele, was sich auch beim Lesen dieses Abschlußberichtes zeigen dürfte.

Der Einsatz verschiedener Methoden (ER, Booch, OMT, UML), Werkzeuge (Unix, Rose, xomt, Xfig, Latex, Emacs, RCS, CVS, Sniff) und Systeme (Oracle, O_2, CORBA) erforderte einen erheblichen Einarbeitungsaufwand. Interessanterweise waren keine Vorkenntnisse der Implementierungssprache C++ vorhanden. Trotz dieser Komplexität wurde das Ziel erreicht und die Teilnehmer(innen) dürften einiges gelernt haben.

Die Projektplanung und -kontrolle durch Pert-Charts hat sich sehr gut bewährt, das gilt auch für die Durchführung einer Einarbeitungsaufgabe zum Kennenlernen der Werkzeuge. Problematisch wurde es allerdings, als zu Beginn des zweiten Semesters die Spezifikation etwas vernachlässigt wurde und es dadurch zu einigen Abstimmungsschwierigkeiten zwischen den Teilgruppen kam, ganz nach dem Motto: Die Einarbeitungsaufgabe haben wir geschafft, jetzt wissen wir was wir tun. Ich hoffe, daß die Teilnehmer(innen) dadurch gelernt haben, den Wert von Spezifikationen etwas zu würdigen. Ein weiterer wichtiger Aspekt waren die Deadlines; insbesondere die Präsentation auf dem Campus-Fest. Ohne die Deadlines, die zwar selten genau eingehalten wurden, wären wir wahrscheinlich heute noch nicht so weit. Für die Projektorganisation hatte es sich als Problem erwiesen, daß sich im Kernbereich des Systems praktisch nur eine Person genau auskannte; insbesondere beim prüfungsbedingten Ausfall eben dieser Person. Trotz allem wurde der Zeitplan eingehalten, ohne in die vorlesungsfreie Zeit gehen zu müssen (was wohl auf eine geschickte Wahl der Deadlines zurückzuführen ist). Die Präsentation der Ergebnisse bei der Siemens AG markierte dann auch praktisch das Ende der Projektgruppenarbeit. Die Fertigstellung der letzten Feinheiten des Abschlußberichts zog sich allerdings noch etwas in die Länge.

Insgesamt hat die Betreuung dieser Projektgruppe, auch wegen des stets guten Klimas, viel Spaß gemacht.

Literaturverzeichnis

[Bal96] H. Balzert. *Lehrbuch der Software-Technik.* Spektrum Akademischer Verlag, 1996.

[BL86] C. Batini and M. Lenzerini. Methologies for database schema integration. *ACM Computing Surveys*, (Vol.18, No.4), December 1986.

[Boo94] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, 1994.

[Cat96] R. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2.* Morgan Kaufman, 1996.

[Cla95] Rational Software Corporation Santa Clara. *Rational Rose — User's Guide (Revision 3.0).* September 1995.

[Fai85] R.E. Fairley. *Software Engineering Concepts.* McGraw-Hill Book Company New York, 1985.

[fps96] *Eine Übersicht über die Fallpauschalen und Sonderentgelte.* WWW-Angebot der Universität München unter: `http://www.med.uni-muenchen.de/icd/fp.html`, 1996.

[G+95] E. Gamma et al. *Desing Patterns.* Addison Wesley, 1995.

[GJM91] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Software Engineering*, chapter Management of Software Engineering. Prentice Hall, 1991.

[GMD96] GMD. *Informationen über BSCW.* WWW-Angebot der Gesellschaft für Mathematik und Datenverarbeitung: `http://bscw.gmd.de`, 1996.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, (8), 1987.

[icd96] *Eine Übersicht über die ICD.* WWW-Angebot der Universität München unter: `http://www.med.uni-muenchen.de/icd/icd.html`, 1996.

[icp96] *Eine Übersicht über die ICPM.* WWW-Angebot der Universität München unter: `http://www.med.uni-muenchen.de/icd/icpm.html`, 1996.

[ISS] ISST. *Institut für Software- und Systemtechnik and Fraunhofer Gesellschaft.* Internet: `http://www.do.isst.fhg.de`, Joseph-von-Fraunhofer-Strasse 20, D-44227 Dortmund.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering.* Addison-Wesley, 1992.

[Jos96] N. Josuttis. Schablonage, Die Standard Template Library. *iX*, (6):104–110, 1996.

[McC94] S. McConnell. *Code Complete.* Microsoft Press, 1994.

[O296a] O2. *O2 Documentation.* O2 Technology, 1996.

[O296b] O2. *O2 System Administration Guide.* O2 Technology, 1996.

[O296c] O2. *O2 System Administration Reference Manual.* O2 Technology, 1996.

[O296d] O2. *O2C Beginners Guide.* O2 Technology, 1996.

[O296e] O2. *O2C Reference Manual.* O2 Technology, 1996.

[O296f] O2. *O2Graph User Manual.* O2 Technology, 1996.

[O296g] O2. *O2Kit User Manual.* O2 Technology, 1996.

[O296h] O2. *O2Look User Manual.* O2 Technology, 1996.

[O296i] O2. *O2Tools User Manual.* O2 Technology, 1996.

[O296j] O2. *ODMG C++ Binding Guide.* O2 Technology, 1996.

[O296k] O2. *ODMG C++ Binding Reference Manual.* O2 Technology, 1996.

[O296l] O2. *OQL User Manual*. O2 Technology, 1996.

[Ora94a] Oracle. *Oracle Book on CD*, chapter SQL*Plus 3.2 Documentation. Oracle Corporation, 1994.

[Ora94b] Oracle. *Oracle Book on CD*, chapter Pro*C Documentation. Oracle Corporation, 1994.

[Ora94c] Oracle. *Oracle Documentation*. Oracle Corporation, 1994.

[Phy96] Phyton. *Informationen über Phyton*. Phyton Home Page: `http://www.phyton.org/`, 1996.

[Rah94] E. Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Addison-Wesley, Bonn, 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[SL90] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SL95] A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), Hewlett-Packard-Laboratories, November 1995.

[SP94] Hans Werner Six and Bernd-Uwe Pagel. *Software Engineering Band 1, "Die Phasen der Softwareentwicklung"*. Addison Wesley Verlag, 1994.

[Ull96] I. Ullrich. *Ein Therapiekontrollsystem für die Angiographie*. Diplomarbeit am Lehrstuhl Softwaretechnologie, Fachbereich Informatik, Universität Dortmund, 1996.

[Wei88] G. Weikum. *Transaktionen in Datenbanksystemen*. Addison Wesley Verlag, 1988.

[YD95] Z. Yang and K. Duddy. Distributed object computing with corba. Technical report, CRC for Distributed Systems Technology, University of Queensland, Australia, 1995.