Adaptive Informationsschicht auf der Basis autogenerierender Komponenten als Anwendung in einem Ressourcenmanagement

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplom - Wirtschaftsinformatiker (FH)
an der Fachhochschule für Technik und Wirtschaft Berlin

Fachbereich Wirtschaftswissenschaften II

Studiengang Wirtschaftsinformatik

Betreuer: Prof. Gabriele Bannert
Dr. Thomas Schmidt

Eingereicht von: Alexander Löser

Berlin, den 22. Januar 2001

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Ressourcen stehen in der Regel nicht unbegrenzt oder kostenlos zur Verfügung. Das Management von Ressourcen bietet eine Möglichkeit, die oft knappen und teuren Ressourcen sinnvoll und zielorientiert einzusetzen. Insbesondere die Integration des Ressourcenmanagements in die Plattform des World Wide Web stellt neue interessante Möglichkeiten für die Verteilung, Bereitstellung und Verwaltung von Ressourcen zur Verfügung. Das WWW-basierte Ressourcenmanagement verlangt jedoch Anwendungen, die sich an schnell ändernde Strukturen flexibel anpassen können. Die Programmierung derartiger Client/Server Anwendungen stellt immer noch eine große Herausforderung für Entwickler dar.

Im Rahmen eines Forschungsprojektes an der Fachhochschule für Technik und Wirtschaft (FHTW) Berlin wurde ein spezielles Datenmodell für das WWW-basierte Ressourcenmanagement entwickelt. Dieses Datenmodell implementiert in seinem Ansatz objektorientierte Paradigmen - wie Vererbung und Instanziierung - in einem relationalen DB-Managementsystem.

Aufbauend auf diesem Datenmodell stellt diese Arbeit ein System für die schnelle Entwicklung von leistungsfähigen Anwendungen für das Ressourcenmanagement vor. Über ein spezielles WWW-basiertes Werkzeug lassen sich Ressourcen und ihre Strukturen datenbankseitig definieren, aus denen Anwendungen für

das Ressourcenmanagement generiert werden. Codegenerierung, eine mehrschichtige, komponentenbasierte Softwarearchitektur und Enterprise Java Beans sind einige der dabei benutzten Technologien. Ein ebenfalls auftretender Schwerpunkt ist die Verbindung von relationalen DBMS mit modernen objektorientierten Konzepten. Die Arbeit gliedert sich in folgende Teile:

Kapitel 2 beschreibt Inhalte und Aufgaben des Ressourcenmanagements. Insbesondere wird auf die Anforderungen an Anwendungen für das WWW-basierte Ressourcenmanagement eingegangen.

Kapitel 3 zeigt Konzepte und Modelle für die schnelle Erstellung von WWW-basierten Anwendungen für das Ressourcenmanagement auf.

Kapitel 4 widmet sich der Auswahl eines geeigneten Komponentenmodells für die Anwendungen.

Kapitel 5 dokumentiert die Implementierung der Konzepte und Modelle.

Kapitel 6 stellt die Generierung einer konkreten Anwendung am Beispiel vor.

Die Arbeit endet mit Schlußbemerkungen und einem Ausblick.

Kapitel 2

Ressourcenmanagementsysteme

2.1 Ressourcenmanagement

Im folgenden Kapitel wird ein Überblick über den Begriff der Ressource und der Verwaltung von Ressourcen gegeben. Die folgenden Zitate beschreiben die wesentlichen Eigenschaften von Ressourcen und den Begriff des Ressourcenmanagements unter betriebswirtschaftlichen Gesichtspunkten:

Ressourcen sind nach Gablers Wirtschaftslexikon [1] *Alle Mittel, die im weitesten Sinne in die Produktion von Gütern und Dienstleistungen eingehen.*

Charakteristisch für Ressourcen ist, daß sie dem Produktionsprozeß eine Kapazität zur Verfügung stellen und ihr Einsatz i.d.R. über einen zeitbezogenen Kostensatz verrechnet wird [16]. Die kostenoptimale Gestaltung des Produktionsprozesses fordert die optimale Nutzung der zur Verfügung stehenden Ressourcen [38].

Das Ressourcenmanagement stellt eine Querschnittsfunktion dar, deren Aufgaben von der Planung, über den Einsatz bis hin zum Abbau der Ressourcen reichen und damit den gesamten Lebenszyklus in einem Unternehmen umspannen. [11] ... Ziel ist, die Wirtschaftlichkeit der betrieblichen Leistungserstellungen durch einen effektiven und effizienten Ressourceneinsatz und - umgang zu steigern. [6]

Wird das Wort *Management* in seiner funktionalen Bedeutung gebraucht, so beinhaltet der Begriff *Ressourcenmanagement* die Ausübung aller die Ressourcen einer Einrichtung, eines Unternehmens usw. betreffenden Führungsfunktionen. Für das Ressourcenmanagement lassen sich folgende Funktionen und Aufgaben definieren:

- Viele Ressourcen sind begrenzt verfügbar. Deswegen sollte der der Einsatz von Ressourcen zielgerichtet erfolgen. Eine Aufgabe des Ressourcenmanagements ist es, Ziele des Einsatzes der Ressourcen für eine Steigerung der betrieblichen Leistungssteigerung zu definieren.

- Die Strategie soll einen Weg zum Ziel für den Ressourceneinsatz zeigen. Dazu müssen Ressourcen und ihre Potentiale erkannt werden. Ausgehend von der Ressourcensituation können dann entsprechend der Zielsetzung Ressourcen beispielsweise auf- bzw. abgebaut oder neu verteilt werden.

- Die Planung ist die Fortsetzung der Strategie für den Ressourceneinsatz. Sie dient der konkreten Vorbereitung auf das Ziel, dem der Ressourceneinsatz dient. Zur Planung zählen beispielsweise die Bereitstellung, die Vergabe von Ressourcen und konkrete Möglichkeiten der Ressourcenkombination bzw. -aufteilung.

- Ressourcen stehen nur begrenzt zur Verfügung. Ihre Zuteilung muß nach bestimmten Kriterien geregelt werden. Die Art und Weise der Benutzung der Ressourcen wird durch einzelne Regeln definiert, die der Ressourcenverantwortliche aufstellt. Regeln finden in einem *Account* eines Nutzers der Ressource ihren Niederschlag. Unter dem Begriff *Nutzer* sollen hier sowohl natürliche Personen, als auch (technische) Prozesse verstanden werden. Ein Account ist also eine abstrakte Zusammenfassung von Nutzungsregeln für eine bestimmte Ressource. Gegenstand des Managements von Ressourcen ist die Definition solcher Regeln.

- Teil des Ressourcenmanagements ist ebenfalls die Bereitstellung der Ressourcen über einen nutzerspezifischen Zugang. Dabei können elektronisch verfügbare Ressourcen, beispielsweise Computer oder Netzwerke, auf elektronischem Weg über einen nutzerspezifischen Zugang an der Ressource bereitgestellt werden. Analog ist es möglich, Ressourcen, wie Bücher einer Bibliothek oder Lehrräume einer Hochschule, elektronisch zu verwalten. Die Bereitstellung der Ressourcen erfolgt dann durch eine elektronische, nutzerspezifische Zuteilung der Ressource.

- Benutzte Ressourcen können über einen zeit-, mengen- oder nutzerbezogenen Kostensatz verrechnet werden. Jeder nur begrenzt zur Verfügung stehenden Ressource muß ein solcher Kostensatz zugeordnet werden. Über das Produkt aus dem Verbrauch an dieser Resource und ihrem Kostensatz werden die Kosten für die Nutzung der Resource abgerechnet. Diese Daten könnten beispielsweise Aufschluß über den tatsächlichen Verbrauch der Ressourcen einer Hochschule pro Studenten geben. Eine Aufgabe des Ressourcenmanagements ist also die Abrechnung von Ressourcen.

- Die Kontrolle und Überwachung des Zugriffs der Ressourcen dient einerseits der Bereitstellung von Daten für die Abrechnung der Nutzung von Ressourcen (wenn möglich) als auch der Kontrolle der Auslastung der Ressource und kann somit für eine weitere Optimierung des Ressourceneinsatzes genutzt werden.

Die Zitate zeigen auf, daß Ressourcen einen komplexen Lebenszyklus besitzen. Dieser Lebenszyklus umfaßt die Planung, den Einsatz und den Abbau der Ressourcen. Während dieses gesamten Zykluses können Änderungen der Eigenschaften der Ressourcen erfolgen. So kann der Fall eintreten, das die Faktoren für die Planung der Ressourcen sich während ihres Einsatzes verändern. Ein Beispiel dafür ist der Ausfall eines Servers oder einer wichtigen Netzwerkverbindung. Manchmal wird durch die Auswertung der Zugriffe auf eine Ressource offensichtlich, das

die Regeln für die Vergabe dieser Ressource noch nicht optimal gewählt wurden. Aus diese Erkenntnissen während des Einsatzes der Ressource resultieren neue Möglichkeiten für der Vergabe der Ressourcen. Dazu müssen die Regeln zur Vergabe der Ressource und die Eigenschaften der Ressource neu überdacht und neu definiert werden. Dieser Vorgang kann sich wiederholen.

2.2 Ressourcenmanagementsysteme

Wie im letzten Abschnitt bereits erwähnt wurde, umfaßt das Ressourcenmanagement ein breites Aufgabenspektrum während des Lebenszyklus einer Ressource. Häufig werden die Aufgaben für die einzelnen Ressourcen auf mehrere Manager aufgeteilt. Diese Manager müssen dann auf die gleichen Informationen zur Verwaltung der Ressourcen zurückgreifen können. Das strategische Management eines Unternehmens benötigt eher Angaben zur Auslastung der Ressourcen und Kennzahlen wie die Kosten für einen Zugriff pro Nutzer. Die Ressourcenmanager müssen schließlich konkrete Regeln für den Zugriff auf die Ressourcen definieren.

Immer mehr Ressourcen, komplexere Eigenschaften der Ressourcen, mehr Nutzer und immer schneller sich ändernde Regeln für den Zugang stellen jedoch hohe Ansprüche an den Ressourcenmanager und verlangen Entscheidungen in immer kürzerer Zeit. Es ist deutlich erkennbar, daß mit zunehmender Zahl der Ressourcen und ihrer Nutzer das effiziente Management dieser Ressourcen eine immer größere Rolle einnimmt. Der Preis dafür ist ein komplexes Management der Ressourcen, das nur auf der Basis eines Managements aller Bereiche in einer gemeinsamen IT-Infrastruktur effizient zu bewältigen ist. Eine solche IT-Infrastruktur bildet alle Prozesse des Ressourcenmanagements, ihre zugehörigen Aufgaben und Daten ab. Sie stellt eine gemeinsame Ablaufumgebung für Management-Anwendungen bereit, die diese Daten auswerten und die speziellen Informationen für die unterschiedlichen Ressourcenmanager bereitstellen. Ebenfalls übernimmt sie die Speicherung und Verwaltung sämtlicher Daten für

das Management der Ressourcen. Damit stellt sie die technologische Basis für ein integriertes Ressourcenmanagent dar. Dazu benötigt diese IT-Infrastruktur eine einheitliche technische Plattform. Für diese Arbeit wurde das WWW -*World Wide Web*- als eine solche Plattform identifiziert. Der folgende Abschnitt stellt die Vorzüge dieser Plattform dar.

2.2.1 Das WWW als Plattform für das Ressourcenmanagement

Der Begriff des WWW - oder *das Internet* im allgemeinen Sprachgebrauch - ist schon häufig definiert worden. Im *Oxford Dictionary of Computing* [12] kann man folgende Definition nachlesen:

Das WWW ist ein verteilter Informationsdienst, der am CERN, dem Europäischen Laboratorium für Teilchenphysik in Genf, in den frühen 90iger Jahren entwickelt wurde. Das Web ist ein im großem Umfang verteiltes Hypermedia-System, das auf einem Netzwerk aus miteinander verbundenen Computern, gewöhnlich als Internet bezeichnet, beruht. Es erlaubt den Zugriff auf Dokumente... . Der Zugriff erfolgt über eine Workstation, die mit dem Netzwerk verbunden ist und auf der ein geeignetes Programm für den Zugriff abläuft. (Übersetzung des Autors)

Dabei können die im Zitat erwähnten Dokumente beispielsweise Texte, Grafiken, Videos oder Audioinformationen enthalten. Die Inhalte der Dokumente werden in der *Hypertext Markup Language*, kurz HTML, beschrieben. Über Links kann auf andere solcher Dokumente verwiesen werden.

Für das Ressourcenmanagement wurde die komplexe, interaktive Plattform World Wide Web aus folgenden Gründen ausgewählt:

- Das WWW besitzt einen plattformunabhängigen Zugriff. Dafür wird lediglich ein für viele Betriebssysteme erhältliches Werkzeug, der Web-*Browser*,

benötigt. Somit kann der Zugriff für nahezu jeden vernetzten Rechnerarbeitsplatz realisiert werden. Die Anzahl der Nutzer einer bestimmten Information, eines Dokuments oder eines WWW-basierten Ressourcenmanagementsystems kann dadurch stark vergrößert werden.

- Der Zugriff kann dabei unabhängig vom Ort von einem beliebigen, für das WWW eingerichteten, Computer erfolgen. Somit können Aufgaben unabhängig von einem festen Arbeitsplatz erledigt werden. Dies schließt auch entferntes Arbeiten von zu Hause mit ein. Der ortsunabhänige Zugriff trägt ebenfalls zum Wachsen der Nutzer eines WWW-basierten Ressourcenmanagementsystems bei.

- Von Nutzern und Ressourcenverantwortlichen kann über die gleiche Plattform auf die Ressourcen zugegriffen werden. Dadurch könnten Aufgaben der Ressourcenverantwortlichen auf die Nutzer übertragen werden. Diese Vorgehensweise nennt man *endnutzergesteuertes Ressourcenmanagement*.

- Der Zugriff auf Informationen im WWW ist preiswert. Hochschulen bieten den Zugriff auf das WWW häufig sogar kostenlos an. Die „Verweildauer“im WWW ist bei einem kostenpflichtigem Zugang von den Kosten abhängig. Geringere Kosten ermöglichen eine längere Verweildauer. Somit können auch mehr Informationen und komplexere, zeitintensivere Aufgaben im WWW erledigt werden.

- Durch Erweiterungen des Browsers sind interaktive, plattformunabhängige Anwendungen möglich. Interaktive Anwendungen können sowohl Informationen aus dem WWW abfragen, als auch Daten des Endnutzers zu entfernten Servern übermitteln.

  Unter *Applikationen* - oder auch Anwendungen - wird dabei ein Programm verstanden, welches direkt mit einem Anwender oder auch einem anderem Programm kommuniziert. Eine solche Anwendung könnte beispielsweise auf entfernte Datenbanken zugreifen. Diese Anwendungen benötigen keine

manuellen Installationen von zusätzlichen Treibern oder Programmbibliotheken, sondern werden automatisch durch den Browser installiert. Somit eignet sich das WWW auch als „Anwendungsumgebung“ für Anwendungen, die wenige oder keine technischen Vorkenntnisse bei den Nutzern voraussetzen dürfen.

- WWW-basierte Anwendungen für das Ressourcenmanagement können rollenbasiert bereitgestellt werden. Gewöhnlich besitzt ein Nutzer andere Rechte für den Zugriff auf eine Ressource als der Ressourcenverantwortliche. In diesem Zusammenhang ist ein von Zugriffsrechten abhängiger Zugriff auf die Ressourcen nötig. Eine *UserRole* ist eine Zusammenfassung aller Rechte des Nutzers für eine Ressource oder eine Applikation.

- Auf viele Ressourcen läßt sich elektronisch zugreifen. Beispiele dafür sind Computer, die selbst eine Ressource darstellen. Können die zu verwaltenden Ressourcen digital erhoben werden, so sollte es auch möglich sein, einen Zugriff auf diese Ressourcen vom WWW aus zu realisieren. Über entsprechende WWW-basierte Anwendungen könnte von einem zentralen Punkt auf alle Ressourcen zugegriffen werden.

2.2.2 Szenarien eines WWW-basierten Ressourcenmanagements

An einigen Beispielen werden nun die Besonderheiten des WWW-basierten Ressourcenmanagements aufgezeigt. Diese Beispiele sollen auf die Besonderheiten und Möglichkeiten der Plattform WWW für das Ressourcenmanagement hinweisen.

Anwendungsbeispiel 1: Endnutzergesteuertes Ressourcenmanagement

Häufig werden die personellen Kapazitäten der betroffenen Ressourcenmanager stark bei der Erfüllung ihrer Aufgaben überschritten. Dabei ist ein Unternehmen

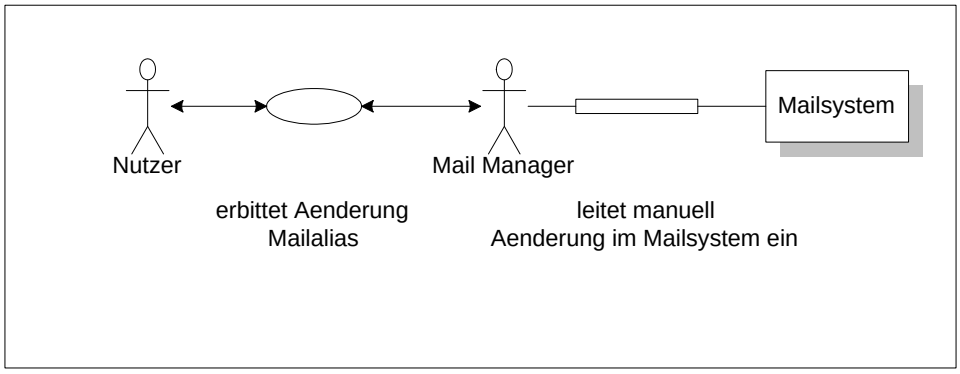

Abbildung 2.1: Änderung des Mail-Aliases durch einen personalisierten Manager

hinsichtlich der Möglichkeit beschränkt, beliebig viele weitere Mitarbeiter zur Lösung dieses Problems heranzuziehen. Eine Möglichkeit dieses Problem zu lösen besteht nun darin, ausgewählte Aufgaben der Ressourcenmanager auf die Nutzer zu übertragen. Durch den gemeinsamen Zugriff auf die Plattform des WWW kann dieser Prozeß technologisch unterstützt werden.

Ein typisches Szenario ist beispielsweise die Vergabe eines Mail-Aliases für einen Nutzer. Der Nutzer nimmt Kontakt zu dem verantwortlichen Mail-Manager auf und erbittet eine Änderung seines Aliases. Der Mail-Manager prüft das Anliegen und ändert gegebenenfalls Einträge im Mail-System (Grafik 2.1). Die Art der Erledigung dieses Geschäftsvorfalles hat den Nachteil, daß ein Mail-Manager ständig präsent sein muß. Änderungswünsche des Nutzers außerhalb der Geschäftszeiten können so nur mit hohem personellem Aufwand berücksichtigt werden. Dabei kann die Qualität und Quantität der Änderung durch den Mail-Manager variieren. Ebenfalls hat der Nutzer nur die Möglichkeit, sich über einen persönlichen Kontakt zu legitimieren und die Änderung vorzutragen. Eine fernmündliche Absprache müßte ein besonderes Verfahren zur Sicherstellung der Identität des Nutzers beinhalten. Grafik 2.2 zeigt einen Lösungsvorschlag für die erwähnten Mängel dieses Szenarios. Der Nutzer ruft eine WWW-Applikation auf und authentifiziert sich. Die Applikation prüft nun die Eingabe des Nutzers für einen Alias nach formalen, vom Mail-Manager vorher einmalig festgelegten Re-

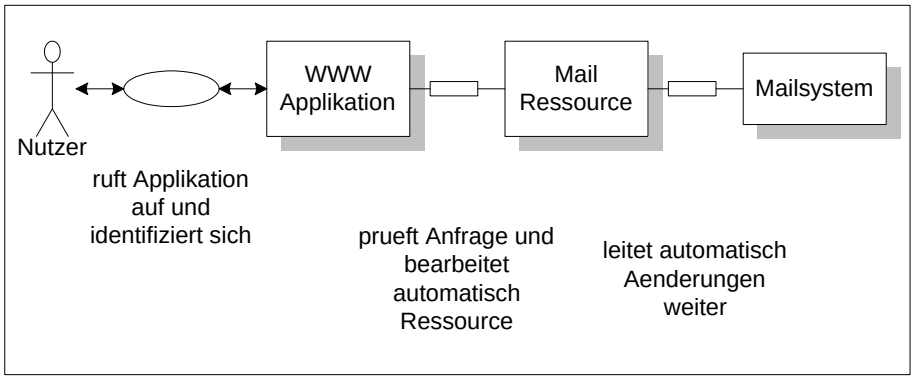

Abbildung 2.2: Änderung des Mail-Aliases über eine WWW-basierte Applikation mit Schnittstelle zur Mail-Ressource

geln. Ist die Überprüfung positiv, bearbeitet die Applikation den Alias des Nutzers in der Ressource *Mail*. Die Ressource gibt wiederum diese Änderungen an das Mail-System weiter.

Damit besitzt das System einen netzwerkweit ständig erreichbaren Zugang. Der Geschäftsvorfall *Mail-Alias ändern* wird in konstanter Qualität ausgeführt. Der Zugriff auf die Ressource `Mail` wird automatisch vom System protokolliert. Aus diesen Daten lassen sich Übersichten für die Auswertung und Abrechnung der Ressource erstellen. Die Präsenz eines personalisierten Mail-Managers ist für die Erledigung dieser Aufgabe nicht mehr nötig. Die Durchführung dieser Aufgabe erfolgt durch den Nutzer mit Hilfe der Applikation.

Durch die Abwicklung durch den Nutzer können die Ressourcenmanager anderen Aufgaben erledigen, bzw. die Zahl der Ressourcenmanager für diese Ressource kann gesenkt werden. Die Kosten für die Einführung dieses Systems könnten über diese Einsparung verrechnet werden.

Diese Art der Abwicklung des Prozesses *Mail-Alias ändern* kann neben der Entlastung des Mail-Managers auch zur Zufriedenheit des Nutzers beitragen. Er erhält das Gefühl, für ihn wichtige Aufgaben auch selbst durchführen zu können.

Dieser Eindruck kann zu einer höheren Akzeptanz des Systems durch den Nutzer führen.

Anwendungsbeispiel 2: Abteilungsübergreifender Geschäftsvorfall

Denkbar wäre ebenfalls das folgende Szenario. Ein Mitarbeiter[1] der Studienverwaltung immatrikuliert einen neuen Studenten für den Studiengang Wirtschaftsinformatik. Der Bearbeiter bestätigt diesen Geschäftsvorfall in einem Informationssystem am Bildschirm. Daraufhin werden nicht nur die entsprechenden Vorgänge der Immatrikulation ausgelöst, sondern für den Studenten wird gleichzeitig eine E-Mailadresse erzeugt, die Zugangsberechtigungen zu verschiedenen Computersystemen und der Bibliothek werden eingerichtet und eine E-Mail des zuständigen Dekans mit generellen Informationen wird an den Studenten zugesandt. Der Bearbeiter der Immatrikulation hat mit seiner formellen Bestätigung weitere, außerhalb seines Sachgebietes liegende Geschäftsvorfälle ausgelöst, ohne die tieferen technischen als auch ablauflogischen Hintergründe kennen zu müssen. Möglich wurde dies durch die Definition von Regeln für den Zugriff auf die entsprechenden Ressourcen (Immatrikulationsamt, Computersysteme, Bibliothek und Mail-Verteiler) und Anwendungen, die den Geschäftsvorfall fachlich in einem Informationssystem abbilden (siehe Grafik 2.3).

Weitere Applikationen

Analog können auch Applikationen entworfen werden, die Routinetätigkeiten, wie beispielsweise die Kontrolle zu lange ausgeliehener Bücher in der Bibliothek, das Heraufsetzen des Druck-Kontingents usw. erledigen. Dabei werden von den entsprechenden Fachkräften für jede dieser Applikationen Regeln formuliert, nach

[1] Im Folgenden wird zur Vereinfachung nur die männliche Schreibweise benutzt. Beispielsweise wird mit Mitarbeiter ebenfalls die Mitarbeiterin assoziiert.

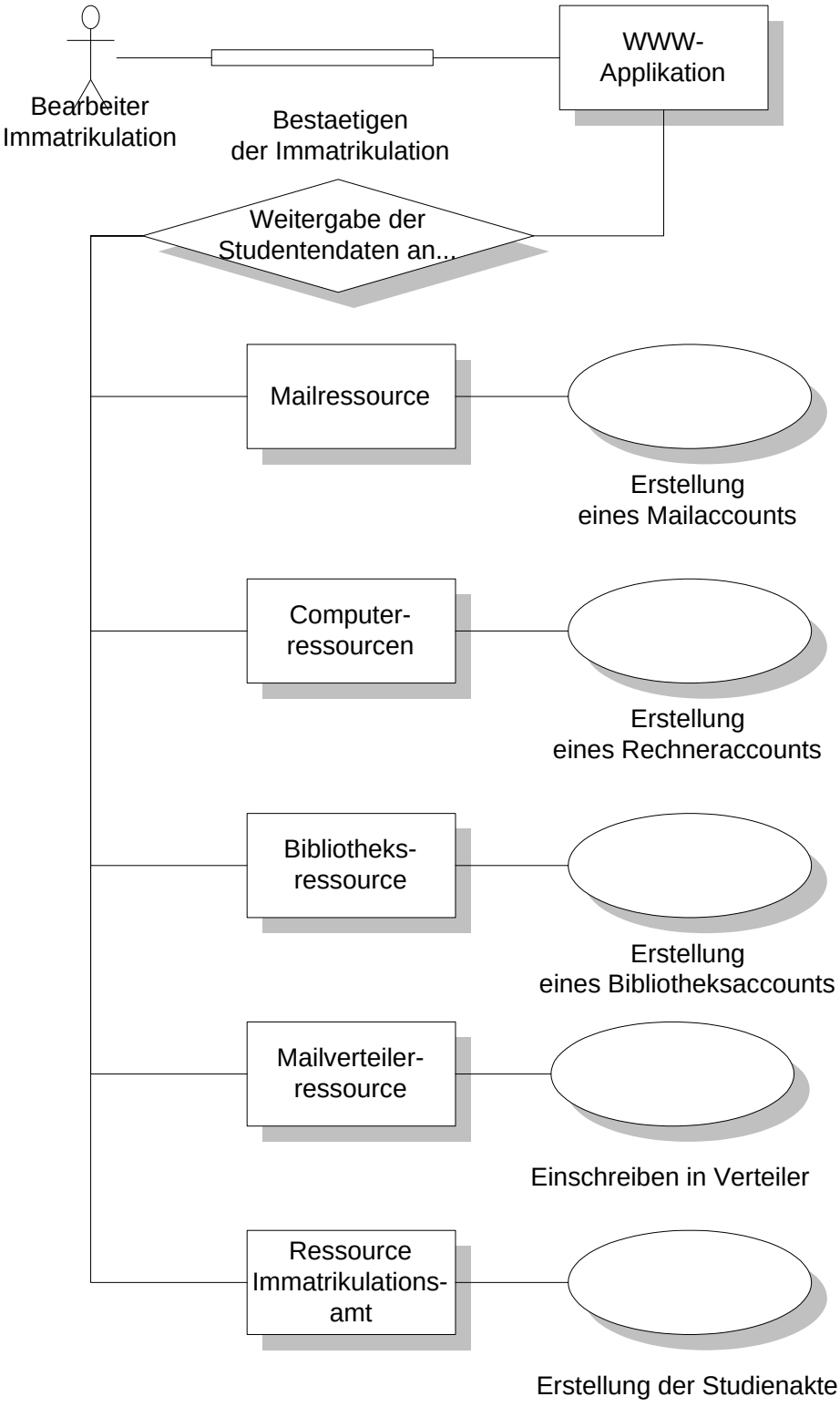

Abbildung 2.3: Vereinfachtes Beispiel des Ablaufes der Immatrikulation eines Studenten

denen die Applikationen funktionieren sollen. Die Applikationen arbeiten dann automatisch diese Regeln ab, ohne eine zusätzliche Steuerung durch eine Fachkraft.

Ebenfalls sind auch Applikationen vorstellbar, die den Fachkräften den Zugriff auf unterliegende Systeme ermöglichen. Oft sind die unterliegenden technischen Systeme komplex und erfordern ein breites Fachwissen. Ein Beispiel dafür ist das System *SendMail*. Von einer Fachkraft einmalig definierte Ressourcen könnten den Zugriff auf *SendMail* übernehmen und Applikationen zur Verfügung stehen, die sowohl die Fachkraft als auch andere Nutzer der Bearbeitung von Mail-Adressen und Mail-Gruppen unterstützen könnten.

2.3 Schlußfolgerungen

Die Beispiele zeigen, das durch die Integration von WWW-basierten Anwendungen in das Ressourcenmanagement viele Vorteile sowohl für Nutzer als auch für Ressourcenmanager entstehen:

- Durch Prozessautomatisierung werden personelle Ressourcen der betroffenen Ressourcenmanager freigesetzt. Ebenfalls erfolgt eine Entlastung der Ressourcenmanager von Routinetätigkeiten. Dies kann zur Steigerung der Arbeitsmotivation und somit der Produktivität führen.
- Der Service wird für die Nutzer erhöht, es kann ein Zugriff 24 Stunden und 7 Tage die Woche von jedem mit dem WWW vernetzten Rechner erfolgen. Diese Tatsache und die gemeinsame Plattform WWW ermöglicht das endnutzergesteuerte Ressourcenmanagement. Neben der Entlastung der Ressourcenmanager kann so die Motivation und Akzeptanz des Systems der Nutzer gesteigert werden. Sie erhalten das Gefühl, selbst die Verantwortung für „ihre“Ressourcen tragen zu können.
- Die Anwendungen können den Ressourcenmanager bei komplexen oder technisch anspruchsvollen Aufgaben unterstützen. Dadurch kann sich der

Ressourcenmanager eher auf die Lösung der Aufgabe konzentrieren, als auf die zur Lösung der Aufgabe benötigten „Werkzeuge“.

- Der Zugriff auf elektronisch verfügbare Ressourcen erfolgt ebenfalls durch ein elektronisches System. Er kann dadurch genauer kontrolliert werden. Ebenfalls lassen sich Auswertungen über die Zugriffe der Ressourcen automatisch erzeugen. Dadurch können sowohl eventuelle bei der Planung der Ressourcen aufgetretene Fehler und Engpässe schneller identifiziert werden, als auch die Nutzung der Ressourcen vollautomatisch auf einen Nutzer bezogen abgerechnet werden.

Um diese vielfältigen Aufgaben zu erfüllen, müssen die Anwendungen beispielsweise folgende fachlichen Funktionen implementieren:

- Definition von Ressourcen und Nutzern
- Implementierung von Regeln für den Zugriff von Nutzern auf Ressourcen
- Implementierung einer *Fachlogik*[2]
- Abrechnung des Verbrauches des Ressource.
- Kontrolle der Auslastung für eine Ressource.

Weiterhin sollen die Anwendungen folgende technische Aufgaben erfüllen:

- Bereitstellung eines WWW-basierten Clients
- Authentifizierung des Nutzers
- Speicherung der Ressourcen und ihrer Daten
- Realisierung des technischen Zugriffs auf die Ressourcen
- Bereitstellung der Ressource

[2] Auf den Begriff der Fachlogik wird in Kapitel 3.4.2 näher eingegangen.

Die letzten Abschnitte zeigten, daß die Integration von WWW-basierten Anwendungen in das Ressourcenmanagement Vorteile sowohl für die Nutzer als auch die Manager der Ressourcen bietet. Für die Implementierung, Konzeption und Einführung solcher Anwendungen ist jedoch ein nicht unerheblichen Aufwand nötig. Die zu entwickelnden Anwendungen müssen dabei zusätzlich die durch sich häufig ändernde Ressourcen bzw. Eigenschaften der Ressourcen entstehende Dynamik verarbeiten können. Die Entwicklung derartiger Applikationen erfordert weiterhin neben einem breiten Informatikwissen auch Kenntnisse über die zu implementierenden betrieblichen Abläufe. Oft besitzen die Entwickler diese Kenntnisse jedoch nicht, bzw. die Ressourcenverantwortlichen verfügen über nicht ausreichende Informatikkenntnisse.Übliche Ansätze der Softwareerstellung genügen diesen Anforderungen nicht. In den folgenden Kapiteln werden Konzepte und Implementierungen vorgestellt, mit denen sich solche Applikationen schnell und in hoher Qualität erstellen lassen.

Kapitel 3

Entwurf eines Systems zur Generierung einer dynamischen Informationsschicht

3.1 Vorbemerkungen

Die hier vorliegende Arbeit ist im Rahmen des Projektes *DORM* entstanden. Das Rahmenkonzept des *Distributed Operational Resource Management* - kurz *DORM*- sieht vor, über eine einzelne Nutzeridentität Nutzern verschiedenste (Rechner-) Ressourcen zur Verfügung zu stellen. Die Kern des Systems ist ein Relationales Datenbank Management System (RDBMS), die *DORM-Datenbank*. Diese Datenbank benutzt ein speziell für das Ressourcenmanagement entwickeltes Datenmodell. Das Projekt *DORM* beinhaltet zum Zeitpunkt der Erstellung der Arbeit weiterhin einige Applikationen für das Ressourcenmanagement.

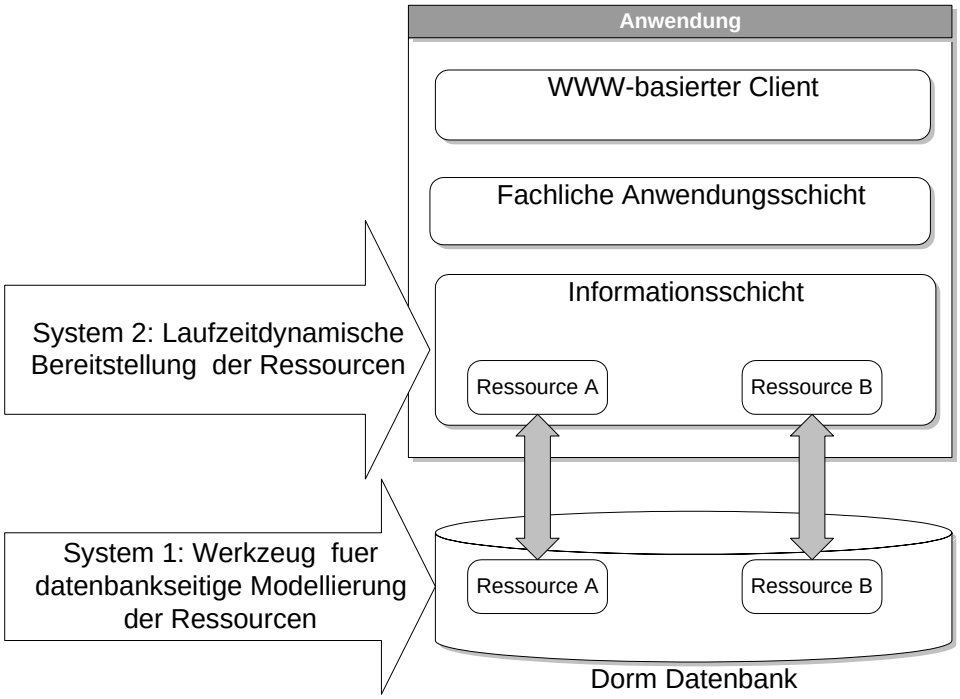

Abbildung 3.1: Systeme zur Bereitstellung von Applikationen

3.2 Zielsetzung

Die Zielstellung der Arbeit ist es, neue Konzepte und Verfahren zu definieren, um Applikationen für das DORM-Projekt schnell und in hoher Qualität entwickeln können. Diese Applikationen sollen dabei einheitlich auf die DORM-Datenbank zugreifen und bereits vorhandene Ressourcen benutzen können. Neue Ressourcen sollen über ein Werkzeug modelliert werden können und dann ebenfalls Applikationen zur unmittelbaren Weiterverarbeitung zur Verfügung stehen. Somit müssen zwei unterschiedliche Systeme entwickelt werden:

1. Ein Werkzeug zur Modellierung der Ressourcen und ihrer Eigenschaften. Die Modellierung umfaßt das Editieren der in den Datenbanktabellen der DORM-Datenbank gespeicherten Ressourcen und Eigenschaften über einen geeigneten Client. Ein ideales Werkzeug - oder System - würde eine plattfor-

munabhängige, netzwerkweite Modellierung gestatten und die Modellierung durch eine intuitive, graphische Benutzerführung unterstützen.

2. Ein System für die Entwicklung von neuen Applikationen für das Ressourcenmanagement innerhalb kurzer Zeit und in hoher Qualität. Die Applikationen sollen auf die bestehende DORM-Datenbank zugreifen können. Abbildung 3.1 zeigt eine solche Applikation. Eine solche Applikation besitzt einen WWW-basierten Client, einen fachlogischen Teil und eine Informationsschicht. Änderungen der Ressourcen und ihrer Eigenschaften sollen für bestehende und neue Applikationen in der Informationsschicht sofort zur Verfügung stehen. Die Applikationen sollen über das WWW aufrufbar sein. Von allen Applikationen gemeinsam genutzte Funktionen sollen in wiederverwendbaren Komponenten[1] implementiert werden.

3.3 Problemerörterung

3.3.1 Das DORM-Datenmodell als Grundlage

Da das DORM-Datenmodell eine wichtige Voraussetzung für diese Arbeit darstellt, möchte ich zuerst mit einen Überblick über dieses Modell beginnen. Grundgedanke des Modells ist die Zuordnung von Nutzern zu Ressourcen. Eine Ressource besitzt mehrere Eigenschaften. So besitzt die Ressource `UnixServer` beispielsweise die Eigenschaften `DiscQuota` und `UserPassword`. Die Werte für diese Eigenschaften können für jeden Nutzer individuell festgelegt werden. Diese Werte werden im Folgenden mit *Belegungen* (im Datenbank-Modell als *settings* definiert) bezeichnet. Ein Beispiel für eine Belegung ist das Paßwort des Nutzers `aloeser`. Es wird über die Eigenschaft `UserPassword` der Ressource `UnixServer` abgefragt. Für jede Eigenschaft können mehrere Belegungen erfol-

[1] Auf den Begriff der Komponente wird in Abschnitt 3.4.4 eingegangen.

gen. Ein Beispiel dafür ist eine Liste ausgeliehener Bücher für einen Nutzer der Ressource `Bibliothek`.

Grafik 3.4 illustriert die Relationen eines *Accounts* zu Ressourcen, Belegungen und Nutzern in einem Entity-Relationship-Modell (ER-Modell). Dieses ER-Modell stellt einen wichtigen Ausschnitt aus dem Dorm-Datenmodell dar und wird von nun an vereinfachend als *DORM-Datenmodell* bezeichnet.

Entitäten sind abgrenzbare (Daten-) Objekte , die eindeutig identifizierbar ein reales Objekt oder eine Abstraktion darstellen. In einer Entitäten-Menge werden alle Entitäten mit gemeinsamen Eigenschaften zusammengefaßt. Zwischen Entitäten, die unterschiedlichen Entitäten-Mengen angehören, können Beziehungen (Relationships) bestehen. Quelle: Thalheim, Entity-Relationship Modeling [37]

Die Zuweisung eines Nutzers zu einer Ressource erfolgt durch einen Account. Unter einem *Account* werden im Folgenden alle Belegungen eines Nutzers für eine bestimmte Ressource verstanden. Ein Account verweist auf die *Entitäten* `Nutzer`, `Ressource` und `Belegungen`. Eine Nutzer kann mehrere Accounts besitzen. So besitzt ein Student einen Account für das System `rz-unix-login` und einen weiteren für den Zugang zur Bibliothek. Jeder Account dieses Nutzers weist mehrere Belegungen für eine Ressource auf. Eine Ressource besitzt Accounts von mehreren Nutzern.

Die Abbildung der Ressourcen in einem DBMS ist ein weiterer wichtiger Schwerpunkt des DORM-Datenmodells. Benötigt werden abstrakte Strukturen zur Definition der Ressourcen, die flexibel anpaßbar sind. Zur Abbildung von Ressourcen im DORM-Datenmodell wurde deshalb ein Metastruktur-basiertes Datenbank-Design gewählt. Die Anpassung der Datenbank an neue Ressourcen erfolgt nicht durch das Hinzufügen von Tabellen für neue Ressourcen und Spalten für neue Eigenschaften, sondern durch das Hinzufügen von Eigenschaften und Ressourcen als Zeilen in den Tabellen `Eigenschaften` und `Ressourcenklassen`. Eine Ressour-

ce wird damit flexibel beschrieben. Ihre auf Metadaten basierte Definition kann so schnell an neue Anforderungen angepaßt werden.

Der Zugriff auf die Entitäten des DORM-Datenmodells erfolgt auf einer höheren logischen Abstraktionsschicht über vordefinierte Prozedur- bzw. Funktionsaufrufe, die vom RDBMS selbst ausgeführt werden, sogenannte *Stored Procedures*. Die Erzeugung, Löschung, Zuweisung, Auslesung und die Modifikation geschehen unter vollständiger Kapselung mit Hilfe von Stored Procedures. Diese Kapselung verhindert den direkten Zugriff auf die Tabellen der Datenbank.

Im DORM-Datenmodell werden Ressourcen mit gleichen Eigenschaften zu einer *Ressourcenklasse* zusammengefaßt. Jede Eigenschaft ist einer Ressourcenklasse zugeordnet. Jede Ressourcenklasse besitzt wiederum Eigenschaften. Grafik 3.2 zeigt die entsprechenden Datenbanktabellen. Die Abbildung der Beziehung `Ressourcenklasse-Eigenschaften` erfolgt in der Tabelle `Eigenschaften`. Diese Tabelle besitzt die Spalten `EigenschaftsID`, `Name`, `TypID` und `RessourcenKlassenID`. Weiterhin existiert die Tabelle `Ressourcenklassen`. Diese Tabelle besitzt die Spalten `RessourcenKlassenID` und `Name`. Ebenfalls existiert die Tabelle `Typen`. Diese Tabelle besitzt die Spalten `TypID` und `Name`. Jeder Eigenschaft ist über die `TypID` ein Datentyp zugeordnet. Jede Eigenschaft wird über die `RessourcenKlassenID` mit einer Ressourcenklasse assoziiert. Damit besitzt jede Eigenschaft einen Typ und ist einer Klasse eindeutig zugeordnet. Damit weist das DORM-Datenmodell, ähnlich dem objektorientierten Klassenmodell, ebenfalls Klassen und Eigenschaften auf.

Die Tabelle `Ressourcenklassen` besitzt ebenfalls die Spalte `Elternklasse`. Werte dieser Spalte sind Primärschlüssel anderer Ressourcenklassen. Auf diese Tabelle kann man nur mittels der bereits oben erwähnten Stored Procedures zugreifen. Dabei lösen die Stored Procedures eventuelle Referenzen zu Elternklassen auf und stellen neben den Eigenschaften der Ressourcenklasse auch die Eigenschaften der referenzierten Elternklasse bereit. Durch diese Kapselung und die Referenz auf

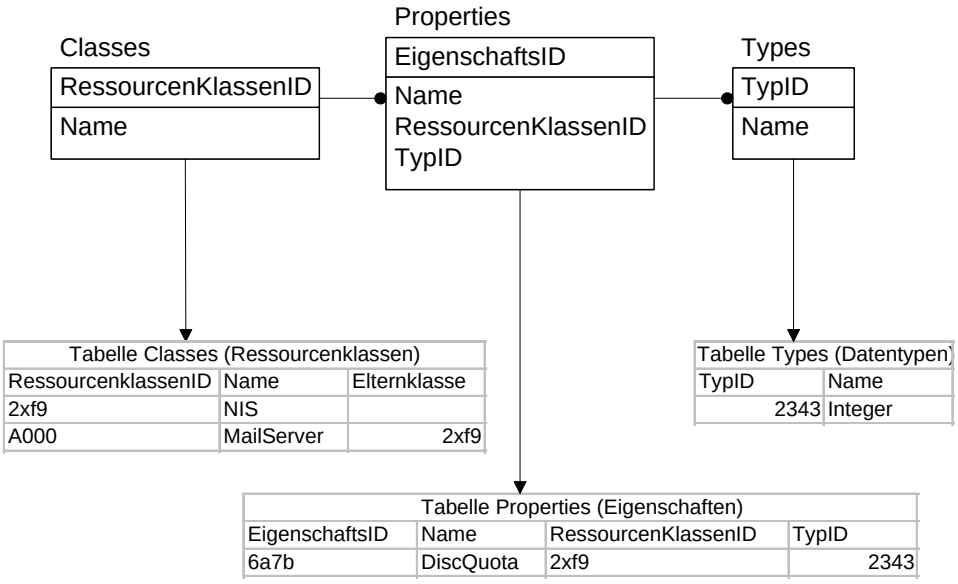

Abbildung 3.2: Klassen und Eigenschaften im DORM-Datenmodell

eine andere Klasse wird es möglich, die „Vererbung“von Eigenschaften in einem relationalen Datenbanksystem zu simulieren (siehe Grafik 3.2).

Diese abstrakten Ressourcenklassen werden den Ressourcen zugeordnet. Die Umsetzung erfolgt über die Tabellen `Ressourcen` und `Ressourcenklassen`. In der Tabelle `Ressourcen` wird zu einer Ressource ihr Primärschlüssel und ihr Name gespeichert. Weiterhin muß zu jeder Ressource in der Spalte `RessourcenklassenID` der Schlüssel ihrer übergeordneten Klasse gespeichert werden (siehe Grafik 3.3). Über diesen Fremdschlüssel wird jede Ressource eindeutig einer Ressourcenklasse zugeordnet. Zusammen mit den Stored Procedures, die den Zugriff und die Struktur der Tabellen kapseln, kann somit ein weiteres wichtiges objektorientiertes Prinzip in einem relationalen DBMS simuliert werden, die Instantiierung.

Die Erzeugung einer bestimmten Instanz einer Objektklasse, generischen Einheit .. wird mit Instantiierung bezeichnet. Quelle: Oxford Dictionary of Computing [12] (Übersetzung des Autors)

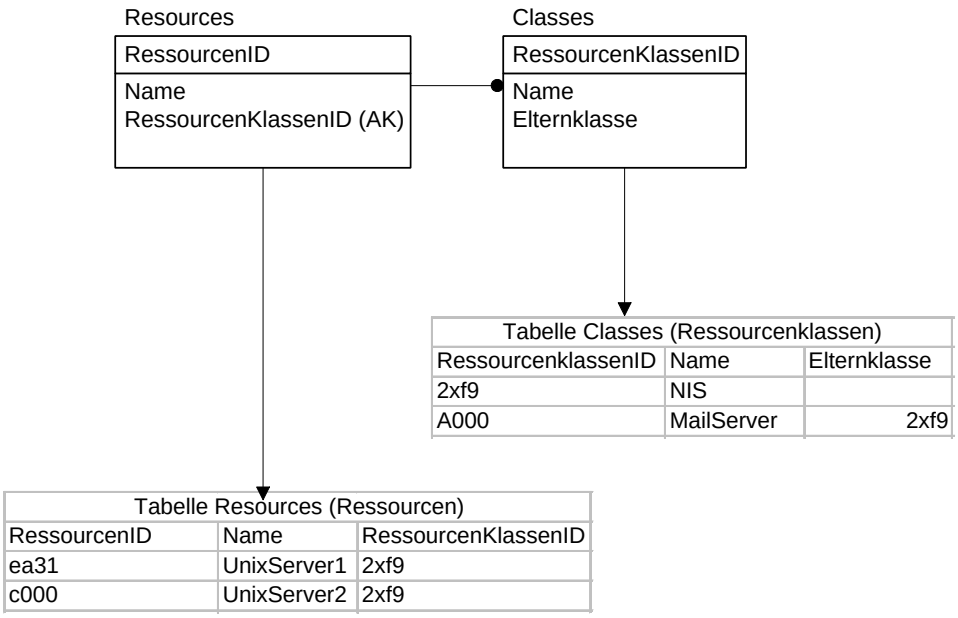

Abbildung 3.3: Instantiierung von Ressourcen aus Ressourcenklassen

Die Ressource instanziiert alle Eigenschaften dieser Ressourcenklasse. Ein Beispiel: Es wird eine Ressourcenklasse `NIS` definiert, die die Eigenschaft `Quota` des UNIX-Dateisystems besitzt. Durch den Zugriff auf die Ressourcen über Stored Procedures wird eine Instantiierung dieser Ressourcen simuliert. Die Ressourcen `UnixServer1` und `UnixServer2` stellen somit „Instanzen“der Ressourcenklasse `NIS` dar. Somit besitzen die Ressourcen `UnixServer1` und `UnixServer2` ebenfalls die Eigenschaft `Quota`.

Durch die Kapselung des Zugriffs via Stored Procedures wird zusätzlich eine Erhöhung der Performance erreicht. Weiterhin bleibt die Implementierung der komplizierten Abfragelogik einem Anwender ebenso verborgen, wie die Struktur der darunterliegenden Tabellen und der Dialekt der verwendeten Abfragesprache.

Grafik 3.4 zeigt den wichtigsten Ausschnitt aus dem Datenmodell. Eine Klasse kann mehreren Ressourcen zugeordnet werden. Ebenfalls besitzt eine Klasse mehrere Eigenschaften. Über diese Relationen, Ressourcen zu Klassen, Klassen zu Eigenschaften, wird die Relationen N Ressourcen besitzen M Eigenschaften

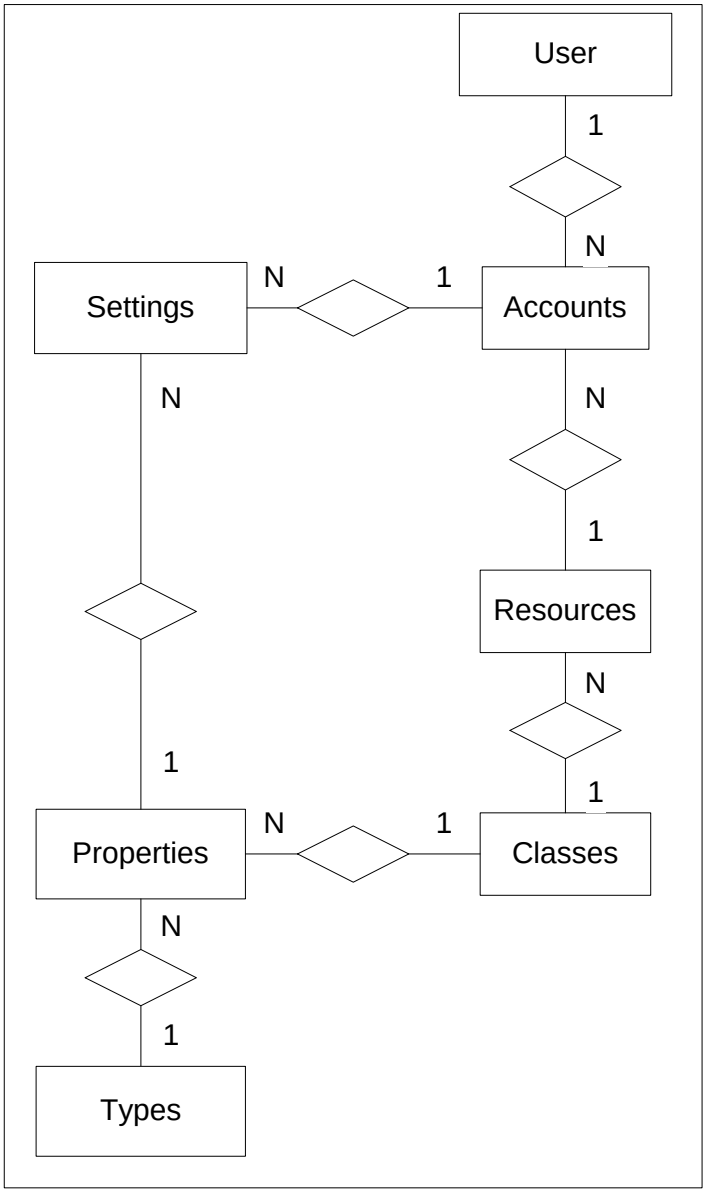

Abbildung 3.4: Entitäten und Relationen des DORM-Datenbank Modells (Ausschnitt)

indirekt hergestellt. Über die `Settings` werden die nutzerspezifischen Werte zu jedem Account zugeordnet. Ein Account hat mehrere Settings, einer Eigenschaft können wiederum mehrere Settings zugeordnet werden. Jede Ressource hat mehrere Accounts. Ein Nutzer besitzt mehrere Accounts.

3.3.2 Werkzeug zur Modellierung der Ressourcen und ihrer Eigenschaften

Momentan existiert zur Bearbeitung der datenbankseiten Tabellen ein auf UNIX - Systemen lauffähiges Kommandozeilen-Interface. Dieses Interface greift dabei auf die schon erwähnten *Stored Procedures* zurück, um Datenbanktabellen zu ändern. Diese Schnittstelle ist durch die *Man Pages* dokumentiert. Dieses in ANSI-C programmierte Anwendung ist jedoch eher als Schnittstelle für Entwickler anzusehen und nicht als vollwertige Anwendung für die Modellierung der Ressourcen.

Eine Zielsetzung der Arbeit ist die Implementierung eines Werkzeuges zur Modellierung der Klassen, Ressourcen und Eigenschaften des DORM-Datenmodell (siehe Abschnitt 3.3.1). Ein solches System müßte folgenden Funktionen bieten:

- Hinzufügen, Editieren und Löschen von Klassen und ihren Eigenschaften
- Zuweisung von Ressourcen zu Klassen
- Darstellung der bereits vorhandenen Klassen, Ressourcen und Eigenschaften
- Zuweisung von Typen, Beziehungen, Constraints, also verbindlichen Kriterien zum Setzen von Eigenschaften sowie [2]
- Zuweisung von Beziehungen, Standardnutzern, Eignern und MetaRessourcen zu Ressourcen
- Evaluierung der Benutzereingabe

[2] Einschränkungen des Wertebereiches

- Übersichtliche Navigation und Benutzerführung

Aufgrund der in Kapitel 2.2.1 vorgestellten Vorzüge des WWW als Plattform für verteile Anwendungen wird in Kapitel 3.4.1 ein System vorgestellt, daß die WWW-basierte Modellierung der Ressourcen erlaubt.

3.3.3 Laufzeitdynamische Bereitstellung der Ressourcen

Ein weiterer wichtiger Aspekt der Zielstellung dieser Arbeit ist es, die datenbankseitig definierten Ressourcen laufzeitdynamisch einer Applikationsschicht bereitzustellen. *Laufzeitdynamisch* bedeutet hier, daß die zugreifenden Applikationen automatisch auf neue, während ihrer Ausführung generierte Ressourcen zugreifen können. Das Problem der laufzeitdynamischen Bereitstellung gliedert sich auf in

- die Abbildung der datenbankseitig definierten Ressourcen in geeigneter Form auf Objekte einer höheren Programmiersprache und
- die Propagierung diese Objekte dynamisch zur Laufzeit an bestehende Applikationen bzw. der Zugriff von bestehenden, ausgeführten Applikationen auf diese Programmstrukturen

Für die laufzeitdynamische Bereitstellung der Ressourcen gibt es zwei Varaianten:

Variante 1. Eine zentrale „generische“Ressource stellt den Zugriff auf alle Ressourcen und die nutzerspezifischen Werte für diese Ressource bereit.

Variante 2. Jede Ressource wird durch eine eigene Java-Klasse repräsentiert. Dabei repräsentieren die Methoden der Java-Klasse die Eigenschaften der Ressourcen

Beide Varianten haben das gemeinsame Ziel, alle benötigten Informationen der Datenbank auf Objekte und Klassen einer höheren objektorientierten Programmiersprache abzubilden. Insbesondere sollen sie das Problem der Zuordnung von Ressourcen zu Eigenschaften auf einer höheren Ebene lösen. Auf den folgenden Seiten werden die beiden Varianten vorgestellt und diskutiert.

Variante 1: Generische Ressource

Die Datenbank kapselt den Zugriff auf die Ressourcen durch wenige Stored Procedures. Im Abschnitt 3.3.1 wurde bereits erwähnt, daß die Datenbank in Stored Procedures gekapselte Funktionen zum Erzeugen und Löschen, Zuweisen, Auslesen und Modifizieren der Ressourcen und ihre Eigenschaften bereitstellt. Für jeden Datentyp gibt es teilweise unterschiedliche Funktionen (Tabelle 3.1 zeigt alle möglichen Funktionen in Abhängigkeit vom Datentyp.)

Eine Signatur kann nun aus der entsprechenden Funktion (lies, schreibe... usw.) für eine Eigenschaft einer Ressource gebildet werden. Dabei stehen für jeden in der Datenbank vorkommenden Datentyp (String, Integer, Date...) auch alle für seinen Zugriff notwendigen Funktionen (get, set, insert...) zur Verfügung. Innerhalb der Methode werden eventuelle Konvertierungen zwischen den Datentypformaten der Datenbank und der Programmiersprache vorgenommen. Das folgende Listing zeigt eine Beispielimplementierung einer solchen Methode.

Listing 3.1: Java Klasse einer GenericRessource

```
1 public class GenericRessource{
2     Account a;
3     // Konstruktor
4     public GenericRessource(Account myAccount){
5         this.a=myAccount;
6     }
7     // Methode schreib für den Datentyp Date
8     public void schreib(String ressourceName,String eigenschaftsName,Date value){
```

Dorm Modell Datentyp	Methode
String	getString setString
boolean	getBoolean setBoolean
date	getDate setDate
int	getInteger setInteger
float	getFloat setFloat
list	insertItem deleteItem listItems
option	getOption setOption listOptions
vector	insertItem deleteItem listItems

Tabelle 3.1: Methoden für den Zugriff auf die Datenbank geordnet nach Datenbankdatentypen

```
9            ...
10           Datenbank.setDate(a,ressourceName, eigenschaftsName,convertDateToDatabaseFormat(value));
11       }
12       // Methode schreib für den Datentyp Integer
13       public void schreib(String ressourceName, String eigenschaftsName, Integer value){
14           ...
15           Datenbank. setInteger (a,ressourceName, eigenschaftsName, convertIntegerToDatabaseFormat(value));
16       }
17       // weitere Methoden für andere Datentypen
18       ...
19   }
```

Durch Überladen von Methoden besteht die Möglichkeit, den Datentyp in der Methoden-Signatur zu kapseln. Im Beispiel existieren zwei Methoden mit gleichem Namen. Sie unterscheiden sich jedoch im Datentyp ihrer Parameter. Eine solche Methode, die zwar den gleichen Namen besitzt jedoch unterschiedliche Parameter bzw. Rückgabewerte hat, heißt *überladen*. Häufig unterscheidet sich die Abbildung von Datentypen in einer Datenbank von der Abbildung von Datentypen in einer Programmiersprache. So wird beispielsweise das Datum in der Datenbank als `ThuNov1618:58:41MEZ2000` dargestellt, in Java jedoch als eine Zahl vom Typ `LongInteger`, beispielsweise `252564998400`. Im Beispiel ist mit den Methoden `GenericRessource.convertIntegerToDatabaseFormat(value)` und `GenericRessource.convertDateToDatabaseFormat(value)` eine Möglichkeit zur Konvertierung der Datentypen der Datenbank in die Programmiersprache Java aufgeführt. Eine solche Signatur besitzt folgende Nachteile:

1. Der Datentyp der Eigenschaft ist nicht ersichtlich. Alle Anwendungen, die diese Methode aufrufen, benötigen somit zusätzlich die Funktionalität, den Datentypen der Eigenschaft zu ermitteln. Der Programmierer der Anwendung muß somit zusätzlich eine spezielle Abfrage an die Datenbank implementieren, die den Datentyp ermittelt.

2. Die Parameter `eigenschaftsName` und `ressourceName` muß beim Aufruf der

Methode bekannt sein. Der Programmierer der Anwendung muß also zum Zeitpunkt der Implementierung des Aufrufs der Methode einen Überblick über die in der Datenbank vorhanden Eigenschaften der Ressourcenklassen und die Beziehung der Ressource zu einer Ressourcenklasse kennen. Insbesondere benötigt er einen Überblick über alle Eigenschaften für eine spezielle Ressource. Oft kennt der Programmierer jedoch weder diese Relationen noch das zugrundeliegende Datenmodell sowie seine Abfragemöglichkeiten.

Durch eine Auflistung aller Ressourcen ihrer Eigenschaften und der Datentypen könnten diese Nachteile beseitigt werden. Dazu stellt die Klasse `GenericRessource` in einer internen Struktur alle Ressourcen, ihre Eigenschaften und Datentypen bereit. Ebenfalls müßte eine einfache und verständliche Programmierschnittstelle zum Auslesen dieser Struktur implementiert werden. Die Programmierschnittstelle muß dabei die Eigenschaften, ihre Datentypen sowie die Ressourcen einerseits für andere Java-Klassen als auch für den Anwendungsprogrammierer lesbar machen. Der Programmierer soll nicht mehr Kenntnisse des Aufbaus und der Abfragelogik der Datenbank benötigen.

Vorstellbar wäre, das der Anwendungsprogrammierer beispielsweise eine Methode `GenericServer.listAllRessourcesAndProperties()` aufruft. Diese Methode würde dann in tabellarischer Form alle Ressourcen auflisten. Ebenfalls denkbar währe eine grafisch orientierte Anwendung, die nach Eingabe einer Ressource die Ressourcen, Eigenschaften und ihre Datentypen auflistet.

Eine solche Vorgehensweise besitzt den Vorteil, das im wesentlichen eine Klasse (hier `GenericRessource`) für das Setzen und Lesen aller Datentypen sowie für die Erstellung einer Übersicht über die Ressourcen, Eigenschaften und ihre Datentypen `GenericRessource` implementiert werden muß. Weiterhin muß in weiteren Klassen eine Schnittstelle zur Datenbank definiert werden. Ebenfalls wird ein geeignete Anwendung zum Auffinden der Eigenschaften für eine Ressource benötigt.

Der Nachteil ist, daß ein Programmierer ebenfalls nur über einen bestimmten

Client Informationen über die Eigenschaften, Ressourcen und Datentypen erhalten kann. Die metastrukturierte Datenbank wird auf einer ebenfalls metastrukturierten Programmierschnittstelle abgebildet. Damit brauchen die Programmier zwar nicht die Abfragelogik der Datenbank zu beherrschen, sind jedoch gezwungen, sich sowohl mit dem metastrukturbasierten Design dieser Schnittstelle auseinanderzusetzen, als auch mit einem zusätzlichen Werkzeug zum Auffinden der Ressourcen und Eigenschaften.

Variante 2: Abbildung der Ressourcen auf mehrere ressourcenspezifische Klassen

Da das DORM-Datenmodell ebenfalls zwischen Klassen und Eigenschaften unterscheidet, soll der Versuch unternommen werden, diese Struktur für die objektorientierte Modellierung von Ressourcen zu benutzen. Das folgende Zitat zeigt eine hilfreiche Definition eines *Objektes* auf.

Ein Objekt weist einen Zustand (Daten) und ein Verhalten (Verarbeitungsmethoden) auf. ... Objekte, die dasselbe Verhalten aufweisen und dieselbe Menge von Zuständen speichern können, gehören ein und derselben Klasse an. (siehe auch Bannert [3], Seite 5).

Objekte, die gleiche Eigenschaften besitzen, gehören derselben Klasse an und stellen Exemplare dieser Klasse dar (siehe auch Courd/Yourdan [7]).

Ressourcen im DORM-Datenmodell sind in der Tabelle `Resources` über ihren Primärschlüssel und ihren Namen definiert. Ihre Eigenschaften werden von genau einer übergeordneten Ressourcenklasse abgeleitet. Somit kann für jede Ressource ein Objekt definiert werden. Dieses *Ressourcenobjekt* besitzt die Eigenschaften der übergeordneten Ressourcenklasse. In Grafik 3.5 besitzt das Ressourcenobjekt `UnixServer1` beispielsweise die Eigenschaft `Quota`. Diese Eigenschaft wird aus der Ressource `UnixServer1` ermittelt, die von der übergeordneten Ressourcenklasse `NIS` bereitgestellt wird. Jede Ressource wird also durch eine eigene

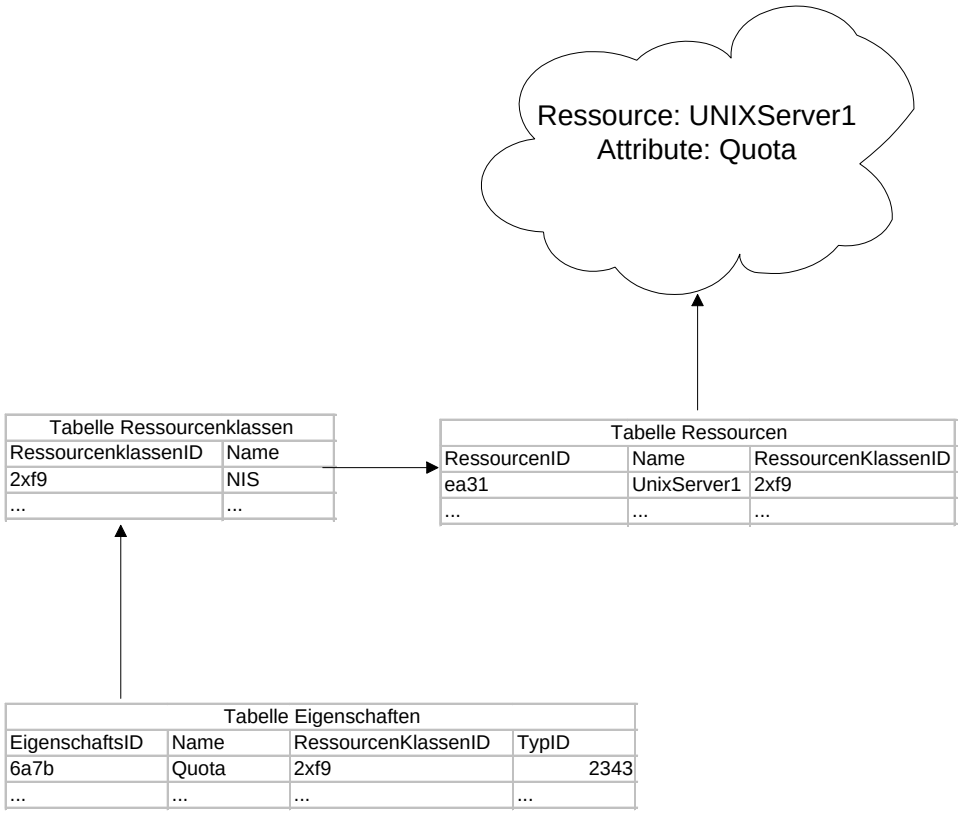

Abbildung 3.5: Ressourcenobjekt UnixServer1

Java-Klasse repräsentiert. Die Signatur einer Methode einer solchen Java-Klasse kapselt jeweils eine Eigenschaft und eine Möglichkeit des Zugriffs auf diese Eigenschaft. Dazu wird aus der Eigenschaft und den Funktionen für ihren Zugriff (lies, schreibe..) eine Methoden-Signatur gebildet. Das folgende Listing zeigt diese Idee am Beispiel der Eigenschaften `Quota` und `LastLoginDate` der Ressource `UnixServer1`.

Listing 3.2: Java Klasse der Ressource UnixServer1 mit Methoden zum Lesen der Eigenschaften Quota und LastLoginDate

```
1 public class UnixServer1 {
2       ...
3       //  Konstruktor
4       ...
```

```
5       // Methode lies für den Datentyp Date
6       public java.util.Date liesLastLoginDate(){
7           java.util.Date setting;
8           static String eigenschaftsName = "LastLoginDate";
9           ...
10          setting = convertToDate(Datenbank.getSetting(ressourceName, eigenschaftsName));
11          return setting;
12      }
13      // Methode lies für den Datentyp Integer
14      public java.lang.Integer liesQuota(){
15          java.lang.Integer setting;
16          static String eigenschaftsName = "Quota";
17          ...
18          setting = convertToInteger(Datenbank.getSetting(ressourceName, eigenschaftsName));
19          return setting;
20      }
21      // weitere Methoden für andere Eigenschaften
22      ...
23  }
```

Über die Signatur der Methode wird somit der Java-Datentyp der Eigenschaft definiert. Innerhalb der Methode erfolgt der Zugriff auf die Datenbank und die Konvertierung des Datentypes aus dem Datenbankformat in das Java-Format. Damit wird die Datenbank komplett in Java-Strukturen gekapselt.

Bewertung der Varianten

Um alle benötigten Informationen der Datenbank auf Objekte und Klassen einer höheren objektorientierten Programmiersprache abzubilden, wurden in diesem Abschnitt zwei Varianten näher diskutiert. Für die Erstellung einer Software mit diesem Ziel gibt es also zwei Möglichkeiten:

- Die Erstellung eines Frameworks mit einer „generischen“Ressource - vorgestellt in Variante 1

- jede Ressource wird einem eigenem Objekt abgebildet - vorgestellt in Variante 2

Viele Frameworks haben den Vorteil, daß viele Details direkt vom Entwickler kontrolliert werden können. Sie sind für die Integration in größere Anwendungen konstruiert. Anwendungen, die das Framework benutzen wollen, müssen von Hand angepaßt werden. Ein Framework gibt normalerweise in einer bestimmte Weise die Möglichkeiten der Interaktion mit dem Framework vor. Die Qualität des Frameworks wird nun in hohem Maße durch die Anpaßbarkeit des Frameworks an die verschiedenen Anwendungen bestimmt. Ein weiteres Problem ist der Grad des (Vor-)Wissens über das Framework, das ein Entwickler besitzen muß. Je mehr das Framework konfiguriert werden kann, umso besser kann es sich an Anwendungen des Entwicklers anpassen. Andererseits ist ein sehr komplexes und umfassendes Framework schwer zu erlernen, schwer korrekt zu benutzen und zu unterhalten. Die Möglichkeit der Balance zwischen der Komplexität und der Benutzbarkeit liegt jedoch nicht beim Anwendungsentwickler, sondern in der Hand des Entwicklers des Frameworks. Für ein Framework ist es damit fast unmöglich, einerseits für eine unbegrenzte Anzahl von Anwendungen bereitzustehen als auch eine geringe Komplexität zu wahren. Somit ist mit der Implementation eines Frameworks immer noch viel Code per Hand zu implementieren und anzupassen.

Für die Arbeit wurde die zweite Variante ausgewählt. Dies wird auf den ersten Blick überraschen. Eine solche Vorgehensweise hat jedoch folgende Vorteile :

1. Sie vereinfacht die Schnittstelle zur Datenbank. Einer Ressource sind nun unmittelbar ihre Eigenschaften zugeordnet. Diese Zuordnung erfolgt auf der Ebene einer objektorientierten Programmiersprache. Dazu wird jede Ressource als eigenständige (Java-) Klasse abgebildet. Sie besitzt *Get*- und *Set*-Methoden für den Zugriff auf die Eigenschaften der Ressource und damit ein für Java-Entwickler verständliches Format. Der Anwendungsentwickler kann dadurch ohne Kenntnisse der Datenbank und ihrer metabasierten Struk-

tur direkt auf die Ressourcen und auf die nutzerspezifischen Werte für diese Ressource über eine hochentwickelte Programmiersprache zugreifen .

2. Anhand des Namens der Methode kann der Programmierer sofort den Zweck der Methode ablesen. Weiterhin werden Methoden-Signaturen definiert, aus denen sowohl die Datentypen als auch die Eigenschaftsnamen ersichtlich sind. Mit Hilfe von *JavaDoc*-einem Werkzeug zur WWW-basierten Dokumentation von Java-Quellcode- können dadurch automatisiert HTML-Seiten erstellt werden, die Ressourcen, ihrer Eigenschaften und der Zugriff auf diese Eigenschaften zeigen. Ein Programmierer kann aus einer solchen Übersicht über einen Web-Browser die entsprechenden Ressourcen und ihre Methode für seine Anwendung auswählen, ohne auf die Datenbank zugreifen zu müssen, oder eine zusätzliche Anwendung zum Auffinden der Ressourcen benutzen zu müssen.

3. Um Gegensatz zur ersten Variante eignen sich die so entstandenen Ressourcenobjekte sehr gut für das automatisierte Auslesen ihrer Schnittstellen durch andere Programme oder Entwicklungsumgebungen. Dieser Vorgang ist in der Java-Programmierung insbesondere durch das Java-Bean Modell auch unter den Begriffen Introspection und Reflection bekannt geworden. Bei der *Reflection* werden zur Laufzeit die Eigenschaften und Attribute einer Java-Bean über ihre Methodensignaturen ermittelt. Damit können zur Laufzeit des Ressourcenobjektes seine Methoden und somit auch die Typen und Zugriffsfunktionen der Eigenschaften der Ressource automatisch ermittelt werden. Java stellt dazu die Bibliothek `java.lang.reflect` bereit. Das folgende Codebeispiel (entnommen aus [20]) zeigt eine Methode, die via Reflection die Typen, Methoden und Parameter einer Klasse ausliest:

Listing 3.3: Auslesen der Methoden einer Klasse via Reflection

```
1 import java.lang. reflect .*;
2     static void showMethods(Object o) {
```

```
3       Class c = o.getClass();
4       Method[] theMethods = c.getMethods();
5       for (int i = 0; i < theMethods.length; i++) {
6           String methodString = theMethods[i].getName();
7           System.out.println("Name: " + methodString);
8           String returnString =
9             theMethods[i].getReturnType().getName();
10          System.out.println("   Return Type: " + returnString);
11          Class[] parameterTypes = theMethods[i].getParameterTypes();
12          System.out.print("   Parameter Types:");
13          for (int k = 0; k < parameterTypes.length; k++) {
14              String parameterString = parameterTypes[k].getName();
15              System.out.print(" " + parameterString);
16          }
17          System.out.println();
18      }
19  }
```

Java-Beans, die über ein zusätzliches Interface, die `BeanInfoClass`, ihre Information bereitstellen, werden über *Introspection* ausgelesen. Während also bei der Reflection direkt auf die Klasse zurückgegriffen wird, kann bei der Introspection auf eine externe Klasse zurückgegriffen werden. Die Technik der Introspection wird insbesondere zum Einlesen von Klassen in virtuelle Programmierumgebungen oder sogenannte RAD[3]-Entwicklungsumgebungen genutzt, um die Programmierung zu vereinfachen. Eine solche Programmierumgebung liest das Ressourcenobjekt automatisch aus und visualisiert es. Der Programmierer hat nun die Möglichkeit, sowohl die Vorzüge der visuellen Programmierung, als auch des *Rapid Prototyping* auf die Ressourcenobjekte und auf sie aufbauende Anwendungen anwenden zu können. So könnte der Aufwand für die manuell zu programmierende Fachlogik durch den Einsatz visueller Programmierung wesentlich erleichtert werden.

[3]Rapid Application Development

Mit den ermittelten Schnittstellen des Ressourcenobjektes läßt sich ebenfalls entscheiden, mit welcher graphischer Komponente die Attribute des Ressourcenobjektes dargestellt bzw. auf sie zugegriffen werden sollen. Dabei wird ein zusätzliches Objekt definiert, der *Mediator*. Es handelt sich dabei um ein Entwurfsmuster, welches das Zusammenspiel einer Menge von Objekten, in diesem Fall zwischen den Ressourcenobjekten und den Elementen der graphischen Benutzerschnittstelle, den *Widgets*, kapselt (siehe auch [8]). Dazu liest das Mediatorobjekt eine Ressource über Reflection aus und stellt dabei den Datentyp der Eigenschaften der Ressource fest. Es besitzt Kenntnis über mögliche Widgets und ihre Schnittstellen und entscheidet mit welchem konkretem Widget sich eine Eigenschaft der Ressource setzten und auslesen läßt. Werden beispielsweise Widgets des `java.awt` Pakets benutzt, eignet sich ein `java.awt.Textfield` zur Darstellung der Datentypen `Float`,`String`,`Date`, für den Datentyp *Boolean* kann zur Darstellung der Belegung das graphische Element `java.awt.Checkbox`, zur Darstellung einer Auswahl kann die Klasse `java.awt.Choice` verwendet werden usw. Grafik 3.6 zeigt dazu eine Ressource mit einer Eigenschaft `DiscQuota` vom Typ Integer. Für diesen Datentyp wurde das Widget `java.awt.TextBox` vom Mediator ausgesucht.

Das eben vorgestellte Szenario kann und soll nicht alle Möglichkeiten der Reflection und Introspection der Ressourcenobjekte zeigen. Dies würde den Rahmen der Arbeit deutlich überschreiten. Der Mediator in Grafik 3.6 stellt deswegen eine im Rahmen dieser Arbeit nicht näher beschriebene „Black Box“dar, die weitere Möglichkeiten der grafischen Visualisierung der Ressourcen offen läßt.

Ein gravierender Nachteil der zweiten Variante ist jedoch die nötige Progammierung von vielen Ressourcenobjekten. Ein mögliches Verfahren wäre eine manuelle Implementierung aller Ressourcen und Methoden. Ein wichtiges Kriterium für eine solche Vorgehensweise ist der Umfang der Implementierung. Zum Zeitpunkt

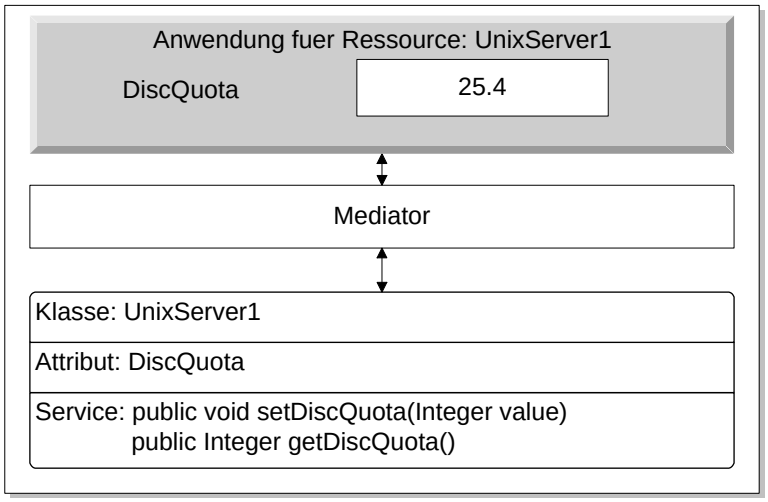

Abbildung 3.6: Mediator zum Auslesen der Eigenschaften einer Resource und der grafischen Repräsentation ihrer Eigenschaften

der Erstellung dieser Arbeit speicherte die Datenbank zehn Ressourcen mit durchschnittlich 30 - 40 Eigenschaften. Für jede Eigenschaft müßte mindestens je eine Methode zum Lesen und Schreiben der Belegungen für einen Nutzer bereitgestellt werden. Damit würde eine manuelle Implementierung zwischen 600 und 800 Methoden umfassen. Schon bei dieser kleinen Anzahl von Ressourcen und Methoden erscheint es sehr mühevoll, die Ressourcenobjekte manuell erzeugen zu wollen. Eine laufzeitdynamische Bereitstellung der Ressourcenobjekte ist nicht möglich.

3.3.4 Generierung der Ressourcen

Seit langer Zeit steht jedoch mit der Codegenerierung eine Technologie bereit, mit der sich unter wirtschaftlich sinnvollem Einsatz von Mitteln große Mengen von Quellcode erstellen lassen. Codegenerierung wird beispielsweise bei der Erzeugung von CORBA-IDL-Proxy Klassen verwendet. Hierbei ist der Codegenerator ein Teil des Object Request Brokers (ORB). Unter *Codegenerierung* wird im Folgenden die automatische Erstellung von Quellcode durch ein Werkzeug verstanden

(vgl. Rösch [23]). Für eine Codegenerierung der Ressourcen sind jedoch einige Voraussetzungen zu erfüllen:

- Für die Codegenerierung werden verschiedene Werkzeuge benötigt. Das wichtigste Werkzeug ist der *Codegenerator*. Benötigt wird dabei ein Codegenerator, der alle Klassen- und Methodenimplementierungen generieren kann. Da der Generator Quellcode erzeugt, wird ein Werkzeug zur Umwandlung des Quellcodes in Java Bytecode benötigt - der *Java-Compiler*. Eventuell wird weiterhin ein Werkzeug benötigt, das die kompilierten Programme automatisch in die Laufzeitumgebung einer Java Virtual Machine integriert und damit anderen Applikationen zugänglich macht - ein *Deployer*. Da der Codegenerierungsprozeß vollautomatisch ablaufen soll, wird schließlich eine zentrale Steuerung für die verschiedenen Werkzeuge benötigt. Abschnitt 3.4.4 stellt ein Systemkonzept vor, das die Steuerung der Generierung und Modellierung der Ressourcen erlaubt.

- Eine typische Eingabe für einen Codegenerator ist ein Modell der zu generierenden Klassen und Methoden. Sollen die Ressourcen durch Codegenerierung erstellt werden, so muß sich auch ein Modell der Ressourcen und ihrer Eigenschaften erstellen lassen. Abschnitt 3.4.3 zeigt ein Konzept auf, um aus einem gegebenen Modell durch einen Generator Quellcode zu erzeugen.

- Da die generierten Ressourcen den bestehenden Applikationen automatisch zur Verfügung stehen sollen, muß ein Konzept entwickelt werden, das eine dynamische Bereitstellung der generierten Ressourcen erlaubt. Abschnitt 3.4.2 zeigt ein Konzept auf, wie die generierten Ressourcen in bestehende Applikationen integriert werden können.

Im letzten Abschnitt wurde erwähnt, das die Erstellung eines Frameworks für die laufzeitdynamische Bereitstellung der Ressourcen eine schlechte Alternative darstellt. Momentan jedoch scheint der Aufwand für die Erstellung eines Generators

ungleich größer zu sein. In dieser Arbeit werden jedoch in den nächsten Kapiteln Möglichkeiten aufgeführt, wie ein Generator durch ein passendes Modell der Ressourcen den Aufwand für eine Erstellung der Ressourcen signifikant verringern kann. Dieses Modell erlaubt es, alle notwendigen Attribute der in der Datenbank definierten Ressourcen über einen ebenfalls in dieser Arbeit entwickelten Codegenerator auf Java-Code zu übertragen. Zu den im letzten Abschnitt erwähnten Vorteilen der zweiten Variante (siehe 3.3.3) kommen nun noch die durch den Einsatz eines Codegenerators entstehenden Vorteile hinzu:

1. Die Codegenerierung besitzt im Gegensatz zur manuellen Erstellung der Ressourcenobjekte den Vorteil, daß sich innerhalb sehr kurzer Zeit die wesentlichen Informationen der Ressourcen auf Java-Code übertragen lassen. Dadurch enstehen **wiederverwendbare, in Java abgebildete und vorkompilierte Ressourcen**.

2. Dabei bleibt die Qualität der erzeugten Objekte konstant. Die Produktion des Codes zur Java-seitigen Abbildung der Ressourcen wird wesentlich fehlerfreier.

3. Die Codegeneratoren können durch die in der Arbeit verwendeten Generatorsprache leicht abgeändert werden. Dadurch wird es beispielsweise möglich, Java-Beans als Clients der Ressourcenobjekte zu erzeugen. Diese Java Beans kapseln clientseitig den Zugriff auf eine Ressource. Sie eignen sich für den Aufruf in einer dynamischen Java Server Page. Der Entwickler der Java Server Page benötigt kein Wissen mehr um die Datenbank, deren Abbildung der Ressourcen und ihrer Eigenschaften sowie deren Zugriffe auf die über mehrere Rechner verteilten Ressourcenobjekte.

3.4 Konzeption

3.4.1 Architekturmuster eines Modellierungssystems

Bis vor kurzem wurden Client/Server-Anwendungen in aller Regel als zweischichtige Anwendungen (*Two-Tier Applications*) realisiert. Dabei umfaßt die erste Schicht die Benutzerschnittstelle und den größten Teil der Verarbeitungsschicht (auch *Anwendungsschicht* genannt). In der zweiten Schicht ist die Datenhaltung in Form von Datenbankservern lokalisiert, kleinere Teile der Anwendungsschicht werden mit Hilfe des Datenbanksystems implementiert. Heutzutage werden mehrschichtige Client/Server-Anwendungen (*Multitier Applications*) wesentlich öfter entwickelt. Dies hatte folgende Gründe:

- In mehrschichtige Softwareanwendungen kann sich jede Schicht auf eine Aufgabe spezialisieren. *Barocca und Hall* definieren in *Software Architectures*[4] fünf Schichten: Ablauf-, Präsentations-, (Geschäfts) Entitäten-, Datenzugriff- und Datenhaltungsschicht.

- Durch eine Abgrenzung der Aufgaben innerhalb der Schichten, können diese für ähnliche Systeme wiederverwendet werden. Ebenfalls ist es möglich, einzelne Schichten der Anwendung auszutauschen.

- Die einzelnen Schichten teilen sich die Arbeit. Diese Trennung ermöglicht es sogar, bei Bedarf jeden Teil der Anwendung auf einem eigenen Computer auszuführen, so daß sie von mehren Computern bearbeitet werden kann.

- Durch Vielschichtigkeit erlaubt eine bessere Pflege der Anwendungen und des Systems. Fehlfunktionen können ebenfalls leichter in den einzelnen Schichten identifiziert werden.

Grafik 3.7 zeigt eine solche Client/Server - Architektur für das benötigte Modellierungssystem. Die Darstellung und Interaktion wird von einem appletbasierten

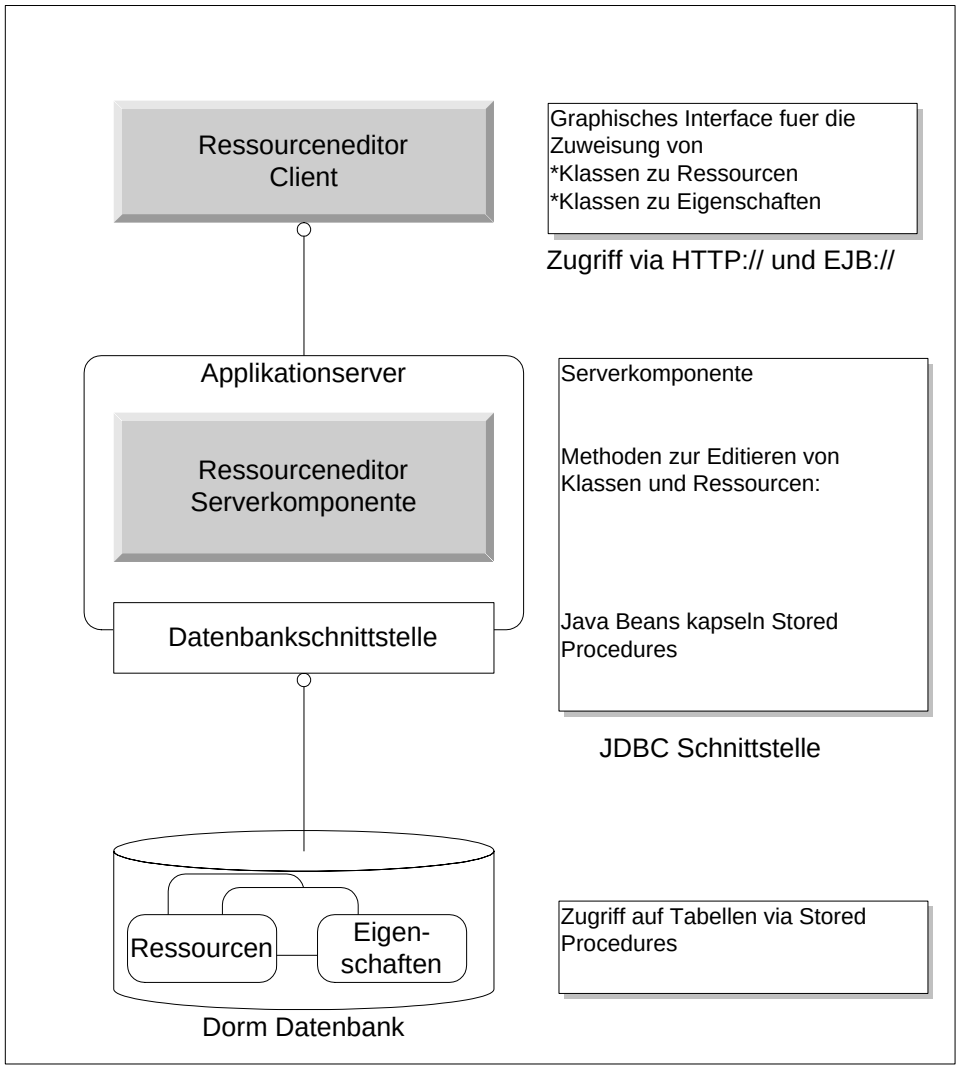

Abbildung 3.7: Architekturmuster für das Modellierungssystem *Ressourcenmanager*

Client übernommen. Über vom *java.awt Framework* bereitgestellte Dialogklassen können Ressourcenklassen und ihre Eigenschaften sowie Ressourcen definiert werden, wobei eine Plausibilitätsprüfung der eingegebenen Daten erfolgt. Über eine sitzungsorientierte Verbindung werden diese Daten an die serverseitige Komponente weitergegeben. Dabei wird sowohl das Protokoll HTTP *(Hypertext Transfer Protokoll)* für die Übertragung des Clients und einer HTML Seite, als auch das Protokoll EJB *(Enterprise Java Bean Protokoll)* für den Austausch von Java-Datenobjekten verwendet. In einer 3-Schichten-Architektur steht typischerweise diese Komponente zwischen der Sicht des Clients und der zur tatsächlichen Objektspeicherung genutzten Datenbank. Im Gegensatz zum Applet besitzt diese Komponente keine Restriktionen für den Zugriff auf das RDBMS. Damit wird das im Abschnitt 3.3.2 besprochene Problem des Zugriffs eines Applets auf die Datenbank gelöst. Da viele der zur Modellierung notwendigen Datenbankoperationen auch von anderen Komponenten wiederverwendet werden sollen, wird der Zugriff zur Datenbank in einer separaten Datenbankschnittstelle gekapselt und ist somit von der Zugriffslogik der serverseitigen Komponente unabhängig. In Kapitel 5.2 wird speziell auf diese Schnittstelle eingegangen. Die letzte Schicht stellt die Datenbank dar, die die benötigten Ressourcenklassen, ihre Eigenschaften und die Ressourcen speichert.

Das hier vorgestellte System zur Modellierung der Ressourcen stellt nur ein grobes Schema für eine spätere Umsetzung dar. Auf besondere, bei der Imlementierung verwendete, objektorientierte Muster, wie Modell-View-Controller, Beobachtermuster und Aufrufe von entfernten Methoden, wird im Rahmen dieses Abschnitts nicht näher eingegangen.

3.4.2 Systemarchitektur zur Trennung von Fachlichkeit und technischer Architektur

Im Abschnitt 3.3.3 wurden schon Ansätze vorgestellt, wie man die datenbankseitig leicht generierbaren Ressourcen ebenso laufzeitdynamisch Clients oder Applikationen zur Verfügung stellen kann. Dabei wurde aufgezeigt, daß sich die datenbankseitige Modellierung der Ressourcen analog auf eine Modellierung in einer objektorientierte Programmiersprache übertragen läßt. Weiterhin wurde aufgezeigt, daß mit Hilfe der Codegenerierung ein Konzept existiert, um mit wirtschaftlich geringem Aufwand, bei konstanter Qualität eine beliebige Anzahl dieser Ressourcen zu erstellen. Offen blieb die Frage, inwieweit sich diese generierten Ressourcenobjekte in eine Client/Server Architektur integrieren lassen. Das folgende Zitat beschreibt das Problem genauer:

Anwendungssysteme weisen oft noch einen hohen Grad von technischer Architektur und fachlichem Programmcode auf. Technische Aufgaben - wie Transaktionsmanagement, Persistenz und Objektaktivierung - sind mit fachlicher Programmlogik vermischt. In diesen Fällen ist es dann sehr schwierig, die technische Architektur weiterzuentwicklen, ohne die programmierte Fachlogik zu verändern... Quelle: Object Spectrum 2/2000, siehe auch [18]

Benötigt wird also eine besondere Softwarearchitektur, die innerhalb einer Applikation die technischen Anwendungsteile von einer zusätzlichen fachspezifischen Programmlogik trennt. Grafik 3.8 zeigt eine solche Architektur. Dabei werden grundsätzlich vier Schichten unterschieden:

- Einen ersten Eindruck von einer Applikation erhält der Nutzer durch den **Client**. Dabei gibt der Client nach Aufforderung durch den Nutzer Anfragen an den Server weiter und präsentiert die Ergebnisse dem Nutzer. Der Client stellt dem Nutzer vielfältige Schnittstellen für eine Interaktion zur Verfügung. Er besitzt die Fähigkeit, entfernte Methoden aufzurufen.

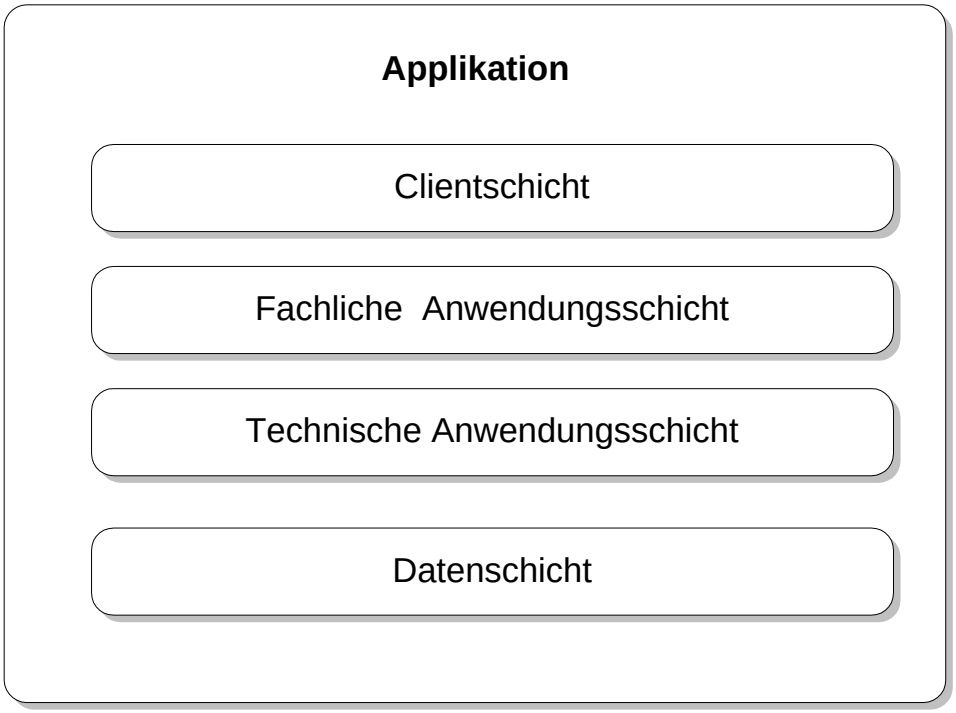

Abbildung 3.8: Mehrschichtige Applikation

- Die **fachliche Anwendungsschicht** besteht aus den fachlichen Operationen der Applikation. Die programmierte Fachlogik kapselt dabei die gesamte fachliche Komplexität der Anwendung. Die Schnittstellen der Objekte dieser Schicht besitzen Methoden, deren Aufruf die Abarbeitung von Geschäftsprozessen und fachlich gesteuerte Abläufe anstößt.

- Aufgaben wie Client/Server Kommunikation, Transaktionsverarbeitung, Objektaktivierung, Persistenz und Replikation werden in der **technischen Anwendungsschicht** gekapselt und damit vollständig vom programmierten Teil der Fachlogik getrennt. Damit ist diese Schicht frei von spezifischer Fachlogik und kann in beliebigen Projekten und Applikationen mit neuen fachlichen Anforderungen wiederverwendet werden (siehe auch [18]).

- Ein gutes Beispiel für die **Datenschicht** ist eine im System vorhandene Datenbank. Aufgabe dieser Schicht ist es, vorhandene Daten zu speichern und

sie auf Abfrage schnell zugänglich zu machen. Beim Zugriff auf die Daten arbeitet diese Schicht eng mit der technischen Anwendungsschicht zusammen.

Der Sinn einer solchen Trennung von Client-, fachlicher und technischer Anwendungsschicht sowie Datenschicht ist es, neben dem Erlangen der in Abschnitt 3.4.1 aufgezeigten Vorteile auch die Generierungsfähigkeit der Architektur zu steigern. Damit hat man die Möglichkeit, für jede Schicht entweder eine generierte oder manuelle Implementation festzulegen. Weiterhin können komplette Schichten ausgetauscht und in anderen Projekten wiederverwendet werden. Folgendes Zitat unterstreicht diese Ideen:

Wenn es gelingt, wiederverwendbare und kombinierbare Elemente im Systemaufbau zu identifizieren und zu isolieren, besteht eine große Chance, die Erstellung der Elemente zu automatisieren und damit die Softwareerstellung wesentlich effizienter zu gestalten. Quelle: Object Spectrum 2/2000 siehe auch [18]

Neben der Erstellung eines Editors für die datenbankseitigen Ressourcen und Ressourcenklassen ist ein weiterer Kernaspekt dieser Arbeit, ein Konzept für eine sich automatisch anpassende Informationsschicht auf der Basis autogenerierender Komponenten zu erstellen.

Als Komponente bezeichnen wir im Folgenden einen wiederverwendbaren Programmabschnitt. Komponenten werden mit anderen Komponenten verbunden, um eine Applikation zu formen. (Übersetzung des Autors aus [39])

Die Idee der komponentenbasierten Entwicklung ist es, anwendbare Software aus wiederverwendbaren Komponenten zusammenzufügen. Jede Komponente ist derart konstruiert, daß sie in unterschiedlichen Umgebungen arbeiten und mit anderen Komponenten agieren kann.[4] *(Übersetzung des Autors aus: Software Architectures [4])*

[4]Weitere Definitionen zum Begriff der Komponente erfolgen in Abschnitt 3.4.4.

Ein solcher wiederverwendbarer Programmabschnitt ist die objektorientierte Implementierung einer datenbankseitig definierten Ressource. Die in der Grafik 3.8 gezeigte technische Anwendungsschicht könnte aus einer Vielzahl solcher Komponenten implementiert werden. Sie stellt ein Beispiel für eine *Informationsschicht* dar. Durch Verwendung von Codegenerierung ist es möglich, bei Veränderungen der Ressourcen eine schnelle Anpassung dieser Architekturschicht zu erreichen.

Eine Applikation kann auf die generierten Ressoucenkomponenten der technischen als auch auf die Komponenten der fachlichen Anwendungsschicht unterschiedlich zugreifen. Grafik 3.9 zeigt mögliche Varianten. Die *Anwendung X* orientiert sich in ihrem Aufbau an dem in Grafik 3.8 vorgestellten Architekturmodell. Ressourcen A,B,C der als Datenschicht verwendeten Datenbank werden in der Informationsschicht durch Generierung als objektorientierte Komponenten abgebildet. Ein vorhandener Client hat die Möglichkeit, direkt auf die Komponente C zuzugreifen und generische Operationen, wie Setzen oder Holen von Belegungen einer Eigenschaft dieser Ressource, auszuführen. In einer weiteren Variante werden die Ressourcenkomponenten A und B von einer zusätzlichen Fachlogik benötigt, um ein Ergebnis aus den Belegungen dieser Ressourcen abzuleiten. Ein Beispiel wäre die Berechnung der Überziehungszinsen für einen Nutzer der Bibliothek. Dabei erhebt die Fachlogik aus der Ressource A die Anzahl der ausgeliehenen Bücher, die Dauer der Bereitstellung sowie aus der Ressource B die Überziehungsgebühr an dieser Bibliothek für ein Buch pro Tag. Ein in der Fachlogik gehaltener Algorithmus berechnet aus diesen Angaben die Schuld des Nutzers. Dieses Ergebnis kann von einem entfernten Client abgefragt werden.

3.4.3 Modell eines Generators

Im Abschnitt 3.3.3 wurde bereits festgestellt, daß die Codegenerierung gegenüber der manuellen Implementierung der Ressourcenobjekte Vorteile aufweist. In die-

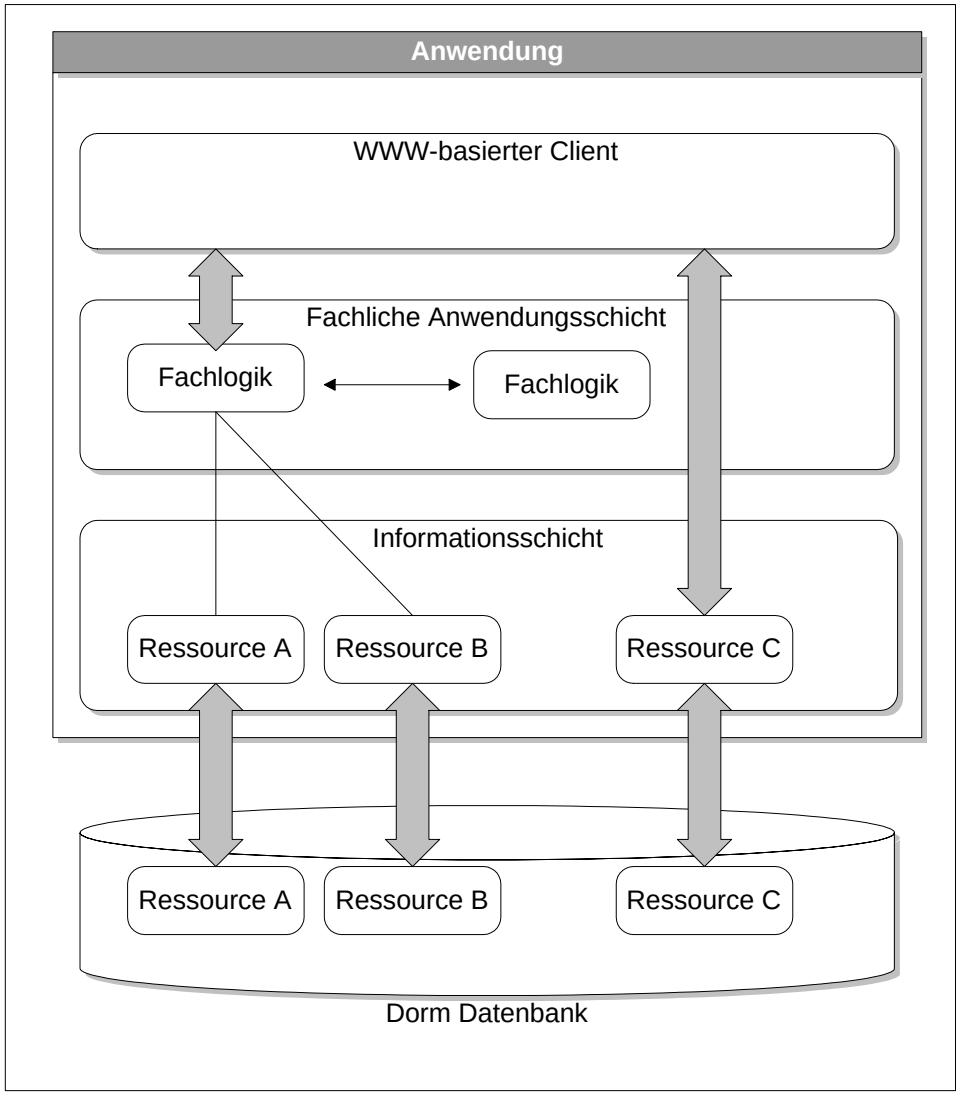

Abbildung 3.9: Applikation, die Varianten des Zugriffs auf fachliche und technische Komponenten der verschiedenen Schichten demonstriert.

sem Abschnitt soll geklärt werden, wie aus einem gegebenen Modell durch einen Generator Quellcode erzeugt werden könnte.

Um variable Teile einer Klasse zu generieren hat es sich als günstig erwiesen, die konstanten Teile wie gewohnt in eine Datei zu schreiben und zusätzliche Markierungen einer Generatorsprache in den Quellcode einzubetten. Eine solche Datei wird von nun an als *Schablone (engl. Template)* bezeichnet. Das folgende Listing zeigt ein Beispiel für eine Schablone:

Listing 3.4: Template für das Entity-Home Interface

```
1 package dorm.server. resources ;
2 public  interface <RESOURCENAME>HomeInterface extends javax.ejb.EJBHome
3 {
4     public <RESOURCENAME>RemoteInterface
5         findByPrimaryKey(dorm.generator. utils .DormPk primkey)
6         throws java.rmi.RemoteException, javax.ejb.FinderException;
7 }
```

Dabei steuern die Markierungen (z.B. `<RESOURCENAME>`) den Generator, während die konstanten Teile unverändert ausgegeben werden. Entsprechende Ersetzungen werden auf Grundlage eines aus der Datenbank generierten Modells vorgenommen. So kann der Name der Ressource später in einen Platzhalter der Schablone eingefügt werden. Grafik 3.10 zeigt das Schema eines solchen Generators. Die in der Datenbank enthaltenen Informationen über Klassen, Eigenschaften sowie Ressourcen (siehe Abschnitt 3.3.1) bilden die Grundlage des für die Codegenerierung benötigten Modells. Soll aus diesem Modell Java-Code erzeugt werden, wird eine Instanz dieses Modells für die Java Programmiersprache benötigt. Offen beleibt die Frage, woher der Generator diese Informationen erhält.

Die Informationen könnten beispielsweise ebenfalls in der Schablone gespeichert werden. Es hat sich jedoch als ungünstig erwiesen, das Modell direkt in der Schablone zu hinterlegen. Die resultierende Schablone wird unübersichtlich und ist

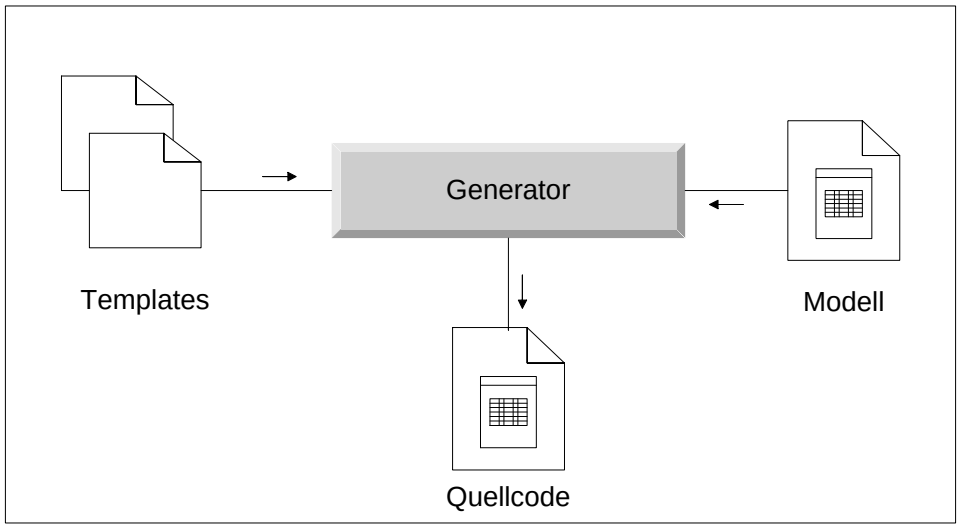

Abbildung 3.10: Schema eines Codegenerators

schwer zu warten. Weiterhin ist es nicht möglich, die Schablone in einem anderen Projekt oder einem weiterem Generator wiederzuverwenden (siehe auch [18]).

Ein weiterer möglicher Lösungsvorschlag ist die Metamodell-Transformation [18]. Diese wurde in der Implementierung aufgegriffen. Hierbei werden die Modellinformationen aus der Datenbank ausgelesen und in ein weiteres (Zwischen)-Modell außerhalb der (Java Code) Schablone abgebildet. Dieses Modell nennt man *Metamodell*. Es folgt ein Ausschnitt eines solchen Metamodells:

```
<RESOURCE>
    <RESOURCENAME>UnixServer1</RESOURCENAME>
    <SUPERCLASSNAME>NIS</SUPERCLASSNAME>
    <PROPERTY>
        <PROPERTYNAME>DiscQuota</PROPERTYNAME>
        <PROPERTYLABEL>Plattenspeicher für einen Nutzer</PROPERTYLABEL>
        <PROPERTYTYPE>int</PROPERTYTYPE>
        ...
    </PROPERTY>
```

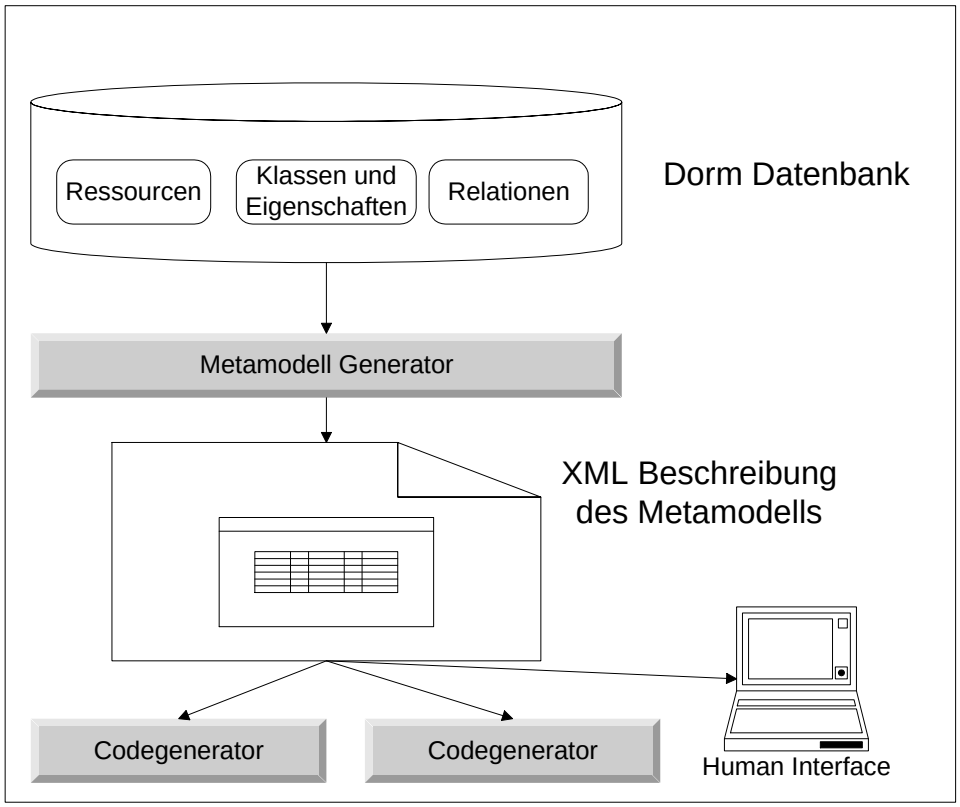

Abbildung 3.11: Metamodell Generator

```
 ... weitere Eigenschaften
</RESOURCE>
... weitere Ressourcen
```

Ein einmal gebildetes Metamodell kann mehrmals verwendet werden. Weiterhin existiert eine für andere Codegeneratoren offene Schnittstelle auf ASCII-Basis, die darüber hinaus XML konform gestaltet wurde. Diese Schnittstelle kann zum Export der Modelldaten genutzt werden (siehe Grafik 3.11). Die textbasierte Struktur des Metamodells ist durch das verwendete XML-Format für Menschen leicht lesbar. Damit ist das Metamodell auch eine Hilfe bei der Programmentwicklung und Fehlersuche.

3.4.4 Konzept eines Systems zur Generierung von Komponenten

In den letzten Abschnitten wurde ein Konzept für einen Codegenerator beschrieben. Der Generator verwendet als Eingabe spezielle, vordefinierte Schablonen sowie ein Metamodell, um Quelltext auszugeben. Dieses Modell wird aus den in der Datenbank gehaltenen Einträgen über Ressourcen und ihren Eigenschaften erzeugt. Ein Client/Server basierter Editor für die Modellierung der Ressourcen wurde bereits in Abschnitt 3.4.1 vorgestellt.

Komponenten können über verschiedene Computer innerhalb eines Netzwerkes verteilt werden (Deployment) und kommunizieren innerhalb dieses Netzwerkes miteinander. Die Komponente wird innerhalb eines Conatiners ausgeführt, der den Kontext der Komponente beinhaltet. Container sind beispielsweise WWW Seiten, Web Browser oder Applikationserver[5]*. Quelle: Übersetzung des Authors aus [39]*

Ziel dieses Abschnittes soll es sein, das bereits in der Problemerörterung erwähnte, zentrale Steuerungssystem für diese und weitere Werkzeuge zu entwerfen. Ein solches System zeigt Grafik 3.12. Ein Benutzer kann dabei von seiner Workstation einen WWW-basierten Client aufrufen, mit dem einerseits eine Modellierung der Ressoucren, aber auch die Steuerung des Generierungs- und Bereitstellungsprozesses ermöglicht wird. Dabei wird durch den bereits vorgestellten Metamodellgenerator ein textbasiertes Abbild der Ressourcen und ihrer Eigenschaften erzeugt. Dieses textbasierte Metamodell dient neben den Schablonen als Eingabe für den Codegenerator (siehe Abschnitt 3.4.3). Der generierte Quellcode wird einem Java-Compiler übergeben. Die so erstellten Komponenten werden schließlich über einen Verteilmechanismus in den Applikationsserver eingefügt und stehen bereits bestehenden und neuen Applikationen als Teil einer Informationsschicht zur Verfügung.

[5] Der Begriff des Applikationserver wird in Abschnitt 3.4.5 erklärt.

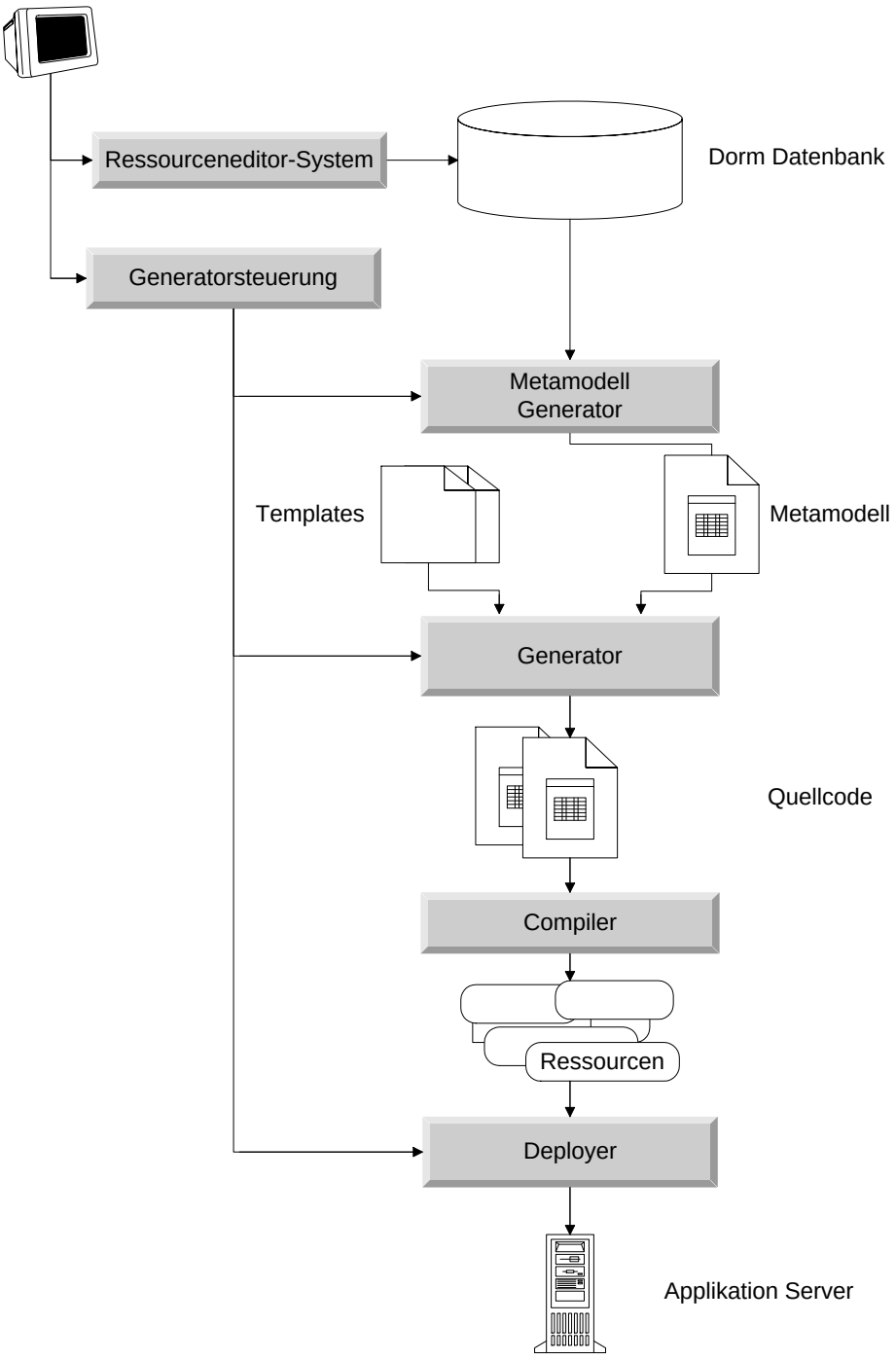

Abbildung 3.12: Schema eines Systems zur Generierung und Bereitstellung der Ressourcenkomponenten

3.4.5 Bereitstellung der Komponenten durch einen Container

Die so erzeugten Komponenten werden in einem Container für einen externen Zugriff bereitgestellt. Nach Suns *Java Bean* Architekturkonzept und Mircosofts *Component Object Model* ist ein *Container* ein ausführbares Programm oder Teilsystem in welchem die Komponente ausgeführt wird. Der Container erlaubt die netzwerkweite rechnerübergreifende Ausführung und Verteilung der Komponenten. Insbesondere die jüngste Entwicklung neuer komponenten- und objektorientierter Architekturen hatte zur Folge, daß spezielle Container für die besonderen Anforderungen von effizienten Infomationssystemen entwickelt wurden. Solche Systeme sind unter dem Begriff *Middleware* oder *Applikationserver* in den IT Sprachgebrauch eingeflossen.

In der IT Branche ist Middleware eine allgemeine Bezeichnung für jede mögliche Art der Programmierung, die eine Vermittlung zwischen normalerweise bereits existierenden Programmen ermöglicht. Eine übliche Middleware-Anwendung erlaubt typischerweise den Zugriff auf Daten von anderen (verteilten) Applikationen auf eine Datenbank. Quelle: Übersetzung des Autors aus [39]

Ein Applikationserver ist ein spezielles Computerprogramm (Server) in einem verteilten Netzwerk, der die Geschäftslogik für Applikationsprogramme bereitstellt. Er wird oft als Teil einer vielschichtigen Applikation angesehen. Oft verbindet der Applikationserver seine Dienste mit einem Webserver. Als Webserver wird ein Programm bezeichnet, das Anfragen zu dynamischen als auch statischen Dokumenten, Bildern etc. über ein spezielles Protokoll abarbeiten kann.

Applikationsserver stellen meist Containerfunktionalitäten für spezielle Applikationsmodelle bereit. Eines der momentan am meisten genutzten Modelle ist das *J2EE Application Modell* [33] der Firma *Sun Microsystems*. Das *J2EE Modell* definiert für *Enterprise Applications* drei fundamentale Teile: Komponenten, Container und Schnittstellen (an Abbildung 3.13 als Connectors bezeichnet). Komponenten bilden den Focus für Entwickler, während Systemhersteller Contai-

Abbildung 3.13: Container, Komponenten und Schnittstellen im J2EE Modell

ner und Schnittstellen implementieren, um deren Komplexität vor dem Entwickler zu verbergen. Container vermitteln dabei zwischen Clients und Komponenten. Sie stellen die benötigten Services, wie Transaktionsunterstützung [6] und *Instance Pooling*[7] für beide Seiten bereit. Die Vermittlung durch einen Container erlaubt vielen Komponenten ihr (Transaktions-)Verhalten erst zum Zeitpunkt ihrer Bereitstellung durch den Applikationsserver zu definieren. Auf den Begriff der Schnittstelle wird im nächsten Kapitel eingegangen.

3.4.6 Schnittstellen

Die Middleware oder der Applikationsserver bietet über Schnittstellen des Containers Möglichkeiten zur Kommunikation mit der Komponente von einem entfernten Client.

Eine Schnittstelle ist die Spezifikation zwischen zwei Programmteilen. Die sorgfältige Definition von Schnittstellen einer Programmeinheit erlaubt ihre Be-

[6] siehe Kapitel 4.2.3
[7] siehe Kapitel 4.2.5

nutzung ohne Kenntnisse ihrer internen Umsetzung. Quelle: Oxford Dictionary of Computing
Eine Instanz der im Container gespeicherten Komponente kann damit zur Laufzeit bereitgestellt werden. Moderne Container erlauben die Bereitstellung einer Instanz einer Komponente über einen eindeutigen Schlüssel (Primary Key) und verbinden damit ein wichtiges Merkmal eines RDBMS mit modernen objektorientierten Prinzipien.

Weiterhin bieten Middleware oder Applikationsserver ebenfalls Schnittstellen zu RDBMS oder Verzeichnisdiensten an. Häufig sind diese Systeme für die Java-Plattform optimiert und können damit den großen Funktionsumfang der *Java API* (beispielsweise das JNDI Interface, die Java Transaction API oder das J2EE Frame Work [33]) nutzen. Diese Schnittstellen können bei der Entwicklung von verteilten Applikationen genutzt werden. Dabei spielt es keine Rolle, auf welcher Plattform die Server (Datenbankserver, Applikationsserver) laufen. Durch diese Erweiterungen der Programmiersprache *Java* müssen Entwickler nicht mehrere Programmiersprachen beherrschen, um auf verschiedene unternehmensweite Services zurückzugreifen.

3.4.7 Prototyping

In den letzten Abschnitten wurde eine Softwarearchitektur für ein Ressourcenmanagementsystem vorgestellt. Dabei wurde gezeigt, daß bei sorgfältiger Trennung zwischen fachlicher Logik und technischen Anforderungen wiederverwendbare und kombinierbare Elemente im Systemaufbau isoliert werden können.

Die technische Architektur dieses Systems kapselt die technischen Aufgaben des Systems wie die Client/Server Kommunikation, Transaktionsmanagement, den Lebenszyklus einer Komponente sowie Persistenz und Replikation vollständig von einer zusätzlich (manuell) programmierten Fachlogik. Eine kleine Anzahl von

Standardoperationen stellt die Verbindung von Fachlichkeit und technischer Architektur her.

Weitere Überlegungen führen zu einem System, welches es erlaubt, aus diesen Elementen generierbare Komponenten zu erstellen. Diese autogenerierten adaptiven Komponenten bilden dabei eine hochdynamische Informationsschicht in einem Ressourcenmanagementsystem. Diese Komponenten werden durch einen Generator erzeugt. Der Generator benötigt konstante Teile des Quelltextes und ein Modell der Ressourcen und ihrer Eigenschaften. Die konstanten Teile des Quelltextes sowie Befehle zur Steuerung des Generators sind in Schablonen zusammengefaßt[8].

Momentan ist noch unklar, wie die benötigten Schablonen bereitgestellt werden. Dazu wird eine Komponente manuell implementiert, die alle Funktionen der später zu generierenden Komponenten definiert. Ein solches Exemplar, das essentielle Funktionen der späteren Komponenten implementiert, wird *Prototyp* genannt[9]. Ein solcher Prototyp für das Ressourcenmanagement sollte folgende Funktionen implementieren:

- eindeutige Prüfung des Zugangs eines Nutzer über seinen Account
- Referenzimplementationen aller in Tabelle 3.1 aufgeführten Methoden. Diese Methoden berücksichtigen alle momentan von der Datenbank unterstützten Datentypen und Funktionen für ihren Zugriff.
- eine vollständige Implementierung aller konstanten Teile der Komponente, wie Komponentenkopf-, Rumpf - und Fußteil, zur Komponente gehörende separate Dateien
- Schnittstellen der Komponente, z.B zu ihrem Container

[8] Der Begriff der *Schablone* wurde in Abschnitt 3.4.3 erklärt.

[9] Im englischen Sprachgebrauch wird das Word *Prototyping* für den Vorgang der Erstellung eines solchen Exemplars benutzt.

Die konstanten Teile der Schablonen richten sich nach dem verwendeten Komponentenmodell. Kapitel 4 widmet sich detailliert der Auswahl eines für diese Arbeit geeigneten Modells.

Nach der Implementierungs- und Testphase des Prototypes werden Steuerungsbefehle für den Generator eingefügt. Schließlich werden die so entwickelten Schablonen dem Generatorsystem zur Verfügung gestellt.

3.5 Resultierende organisatorische Veränderungen

Eine Einführung einer neuen Technologie oder eines neuen Softwaresystems ist fast immer mit Änderungen in der Ablauf - und Aufbauorganisation verbunden. Die in diesem Kapitel vorgestellte Technologie eignet sich dazu in ganz besonderem Maße. Durch die Trennung zwischen einer Präsentationsschicht, Fachlogikschicht, Informationsschicht und Datenschicht können Kompetenzen zwischen den einzelnen Spezialisten wie Softwareentwicklern, Datenbankspezialisten, Fachleuten, die die Ressourcen modellieren, Client-Interfacespezialisten und Gestaltern klar getrennt werden. Dies führt dazu, daß Projekte durch eine parallele Arbeitsweise schneller abgewickelt werden können. Mitarbeiter können sich auf einzelne Technologien konzentrieren. So werden von einem Middleware-Spezialisten nicht mehr Fachkenntnisse der Datenbankprogrammierung oder des Datenmodells benötigt. Entwickler der Clients benötigen kein umfassendes Know-How über verteilte Softwarearchitekturen, sondern greifen über eine Schnittstelle auf darunterliegenden Schichten zurück.

Weiterhin werden mit diesem Ansatz eines neuartigen Systems für das Ressourcenmanagement Entwickler von Routineprogrammieraufgaben entlastet. Dies kann zu Motivationssteigerungen bei den betroffenen Mitarbeitern führen. Die klassische Aufgabe des Entwicklers, die Abbildung einer Spezifikation in konkretem Programmcode, tritt durch die angewandte Codegenerierung zunehmend in den Hintergrund.

Im Hinblick des aktuellen Trends der personell schlecht abgedeckten IT-Infrastruktur und der steigenden Komplexität von derzeitigen und zukünftigen Technologien stellen diese Eigenschaften der vorgestellten Architektur und Technologien einen nicht zu unterschätzenden wirtschaftlichen Wert insbesondere für kleine Projektgruppen dar.

3.6 Überblick über ähnliche Ansätze

Das in dieser Arbeit beschriebene Informationssystem ist ein möglicher Ansatz für die Erstellung einer anpassungsfähigen, generierbaren Informationschicht. In letzter Zeit, insbesondere während der Erstellung dieser Arbeit und des zugrundeliegenden Softwaresystems, wurde dieses noch sehr neue Thema von mehreren kommerziellen und wissenschaftlichen Arbeitsgruppen aufgegriffen. Die Bezeichnungen der Systeme und der Focus der Anwendung variieren dabei beträchtlich. Im Folgenden werden daher zwei Systeme näher vorgestellt, die sich bezüglich Schwerpunkt, Idee und Umsetzung nicht unbedingt mit dem vorgestellten Architekturkonzept für ein Ressourcenmanagement decken, dafür aber einzelne Aspekte der Codegenerierung einer Informationsschicht oder der Abbildung eines relationalen Datenmodells auf Objekte anders oder vielleicht besser lösen.

So wird das System *RC Generator der Rösch Consulting* [24] vorgestellt - als Beispiel für den Einsatz eines Codegenerators zur Kapselung einer Informationsschicht. Das System *Avantis Unisuite for EJB* [2] - als Komplettlösung für ein UML basiertes objektrelationales Mapping für Enterprise Java Beans. Alle Ansätze werden kurz erklärt und in ihren besonderen Eigenschaften mit dem in dieser Arbeit beschriebenen System verglichen.

3.6.1 RC Generator der Rösch Consulting

Der *RC Generator der Rösch Consulting* ist ein Werkzeug, das, ähnlich dem in dieser Arbeit vorgestellten Ansatz, die Idee verfolgt, Zeitersparnis und verbesserte Qualität durch die Aufteilung der Verantwortungsbereiche „Betriebliche, d.h. fachliche Ablauflogik und Verarbeitungsregeln“und „Technische Softwarearchitektur“zu erreichen und die Ergebnisse aus den so gewonnenen parallelen Arbeitsabläufen durch einen Generator in funktionierendem Anwendungscode automatisiert abzubilden. Dabei konnte die *Rösch Consulting* ihr Generierungsverfahren unter anderem in der Versicherungsgruppe *Signal IDUNA* erfolgreich einsetzen.

Der Generator benötigt ein OOA Modell, ergänzt um OOD und Implementierungsinformationen. Das OOA Modell wird dazu in UML erstellt. Die OOD-und Implementierungsinformationen umfassen beispielsweise Datentyp und Längenangaben der Parameter, Namen für Klassen, Attribute und Methoden sowie Kennzeichnungen von Attributen als Schlüsselfelder.

In einer generierungsfähigen Softwarearchitektur wird aus diesen Angaben ein großer Teil generischer Methoden einer Informationsschicht erzeugt, wie Methoden zum Lesen und Setzen von Attributen, dem Anlegen und Löschen von Objekten, dem Anlegen von Beziehungen. Zusätzlich kaspelt der gesamte technische Code den Zugriff von Objekten auf die Datenbank, das Transaktionsverhalten, sowie die Client/Server Kommunikation. Fachlicher Code für die individuelle Anpassung von Objektklassen wird zusätzlich manuell hinzugefügt. Der Generator benutzt Templates um Quellcode zu erzeugen. Folgender Text zeigt ein solches Template:

```
package #thePackage->getName() ;
public class #theClass->getName()
    extends #theClass->getSuperClass()->getName();
{
```

```
#FOR theOperation IN theClass ->getOperations()
#INCLUDE "Operation.template"
#endfor
```

Ein vorangestelltes Doppelkreuz stellt die Markierung für einen Befehl des Generators dar. Der vom Generator erzeugte Quellcode wird dann mit Hilfe von üblichen Compilern oder Datenbankwerkzeugen in eine ausführbare Form gebracht.

Der Ansatz der *Rösch Consulting* und die Publikationen der Mitarbeiter des Unternehmens und des Unternehmers *Rösch* in der Fachzeitschrift *Objekt Spectrum* (siehe [23] und [18]) waren für diese Arbeit eine wichtige Grundlage für den Aufbau einer generierungsfähigen Softwarearchitektur. Der Gedanke, eine solche Architektur durch eine Trennung von Fachlichkeit und technischer Architektur aufzubauen, wurde in dieser Arbeit speziell auf das Ressourcenmanagement angewandt. Bei der Konzeption eines geeigneten Generators war das bei *Rösch Consulting* vorgestellte Prinzip der Nutzung von Templates und eines Metamodells für die vorliegende Arbeit maßgeblich.

3.6.2 Avantis Unisuite for EJB

Die auf dem *Java Forum Stuttgart* [14] vorgestellte Softwarelösung *Unisuite for EJB* der *Avantis GmbH* [2] ist ein Werkzeug, mit dem Entwickler Enterprise Java Beans im UML-Modell entwerfen. Integriert mit dem EJB-Applikationsserver bietet die *Persistency-Bridge for EJB container-managed persistency* ein objektrelationales Mapping zur flexiblen und performanten Anbindung von Entity Beans an relationale Datenbanken.

Die *UML Bridge for EJB* generiert ausführbare Enterprise Java Beans (siehe [31] und Kapitel 4.2) direkt aus einem UML-Modell eines wählbaren Modellierungswerkzeuges. Dabei erweitern *Avantis Add-ins* das Modellierungswerkzeug um Funktionen, die es dem Entwickler erlauben, alle Informationen für eine

Quellcodeerzeugung im Modell zu definieren. So lassen sich relationale Schemata, Finder- und Erzeugermethoden, Primärschlüsseleigenschaften und das Transaktionsverhalten abbilden. *UML Bridge for EJB* generiert Remote-Interfaces, Implementierungsklassen, Home-Interfaces, Primärschlüsselklassen, Klassen für die „Transmission per Value“, einfache Java-Klassen und Interfaces sowie die benötigen „Deployment-Deskriptoren“.[10]

Mit der zweiten Komponente *Persistency Bridge for EJB* versucht *Avantis* dem Problem der (meist) fehlenden Trennung von logischem Objektmodell und physikalischen Datenmodell zu begegnen. Diese Trennung wird durch eine objektorientierte Abbildung aller Standarddatentypen einer relationalen Datenbank erreicht. Weiterhin wird durch eine spezielle Abbildung der Objektklassen in Datenbanktabellen für Ober- und Unterklassen ein Vererbungsmechanismus in einem RDBMS simuliert. Anfragen sowie Transaktionen werden auf Objektebene realisiert (siehe Abbildung 3.14) .

Avantis GmbH deckt mit ihrer Softwarelösung die wesentlichen Ansprüche an das Ergebnis der im Rahmen dieser Arbeit vom Autor vorgestellten Systemlösung gut ab. So liegt ein Schwerpunkt auf der schon erwähnten Trennung von Fachlichkeit und Technischer Architektur. Die Abbildung eines (UML-) Modells wird zur Erstellung von Code benutzt. Der Zugriff auf die Datenbank erfolgt über eine höhere objektorientierte Ebene. Das Problem der objektrelationalen Abbildung von in RDBMS definierten Tabellen auf Objekte einer anderen Ebene wird über ein eigenes Werkzeug gelöst. Weiterhin wird das der Generierung zugrundeliegende Modell aus einem manuell definierten UML-Modell abgebildet. Es werden Komponenten vom Typ *Entity Enterprise Java Beans* erzeugt. Jedoch bietet die Software keine Möglichkeit des bei diesem Komponententyp notwendigen Deployments für die in dieser Arbeit benutzte Middleware *Sybase Enterprise Application Server [35]*.

[10] Quelle: Übersetzung des Autors aus [2]

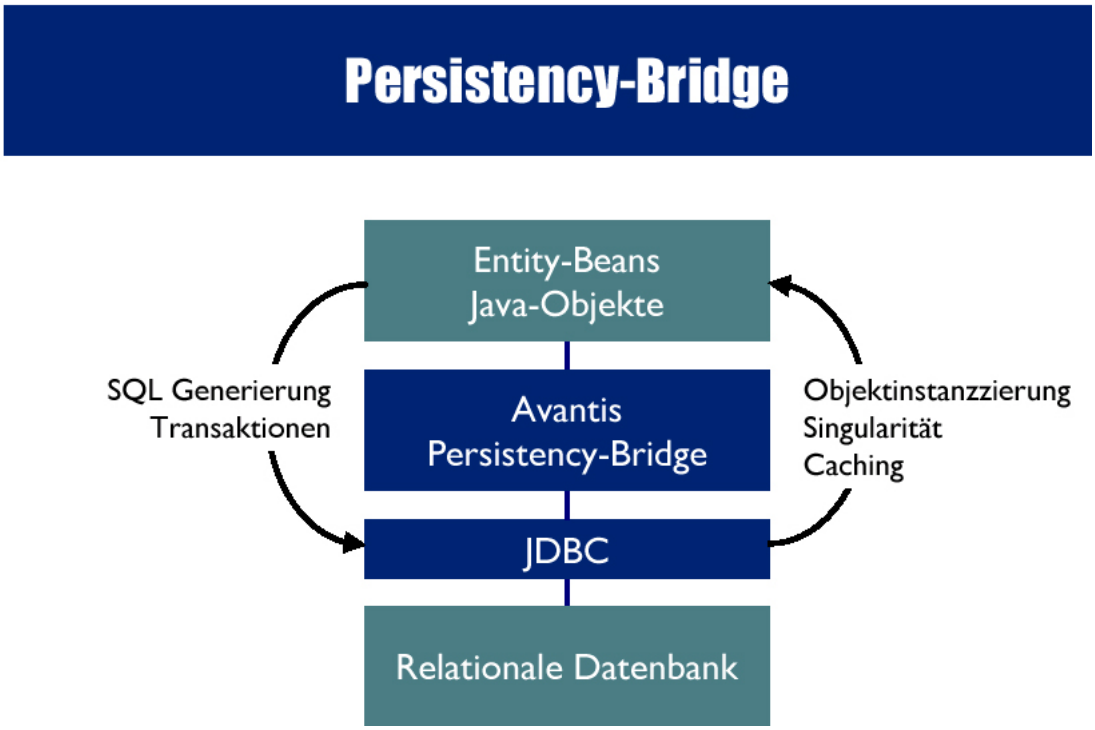

Abbildung 3.14: Avantis Persistency Bridge

Kapitel 4

Die Auswahl des Komponentenmodells

Die Auswahl der Komponenten ist für die Erstellung eines Prototyps für die anschließende Codegenerierung entscheidend. Dieses Kapitel ist dieser Tatsache gewidmet. Zuerst sollen einige der aktuellen Komponentenmodelle kurz vorgestellt werden. Dabei werden zwei Komponentenmodelle als besonders geeignet identifiziert. Diese beiden Ansätze werden dann genauer nach bestimmten Kriterien untersucht. Schließlich wird eine begründete Entscheidung zugunsten eines Modells gefällt.

4.1 Eine Auswahl aktueller Komponentenmodelle

Der Softwaremarkt stellt schon seit geraumer Zeit verschiedenste Komponentenmodelle bereit. Ein wesentliches Kriterium bei der Auswahl der Modelle war die Unterstützung durch die bereits verwendete Middleware *Sybase Enterprise Application Server Jaguar 3.5* [35]. Sie unterstützt folgende Modelle:

- mit ActiveX/COM programmierte Objekte (Mircosoft)

- CORBA-kompatible Komponenten (OMG) [1]
- Java Beans, Enterprise Java Beans (Sun Technology)

Darüber hinaus existierende, jedoch nicht unterstützte Komponentenmodelle für verteilte Systeme, wie beispielsweise die *Microsoft Transaction Services (MTS)*, werden im Rahmen der Arbeit nicht berücksichtigt. Nähere Informationen zu diesem Komponentenmodell sind den Web-Seiten von Microsoft [19] und Sun Microsystems [27] zu entnehmen.

Auf *ActiveX/COM* basierende Komponenten können via DCOM - *Distributed Component Object Model*- oder IIOP - *Internet Inter-ORB Protocol* - Informationen mit entfernten Clients austauschen. ActiveX und die damit verbundene Client/Server Kommunikation über DCOM/COM wurde aus den folgenden Gründen nicht in die nähere Wahl eines geeigneten Komponentenkonzepts einbezogen:

- *ActiveX Controls* als Antwort von Microsoft auf Suns *Java Applet* Konzept sind nur auf den Plattformen Windows 95/98/NT und auf dem *Macintosh Operating System* lauffähig. Damit könnten beim Einsatz des Ressourcenmanagementsystems insbesondere UNIX Plattformen - die in der künftigen Arbeitsumgebung dominieren - clientseitig nicht unterstützt werden.
- Das Datenbanksystem und die verwendete Middleware arbeitet ebenfalls auf einer UNIX Plattform. Die bekannte schlechte Adaptivität von Produkten aus dem Haus *Microsoft* zu nicht von *Microsoft* hergestellten Systemen, insbesondere zu UNIX Systemen, ergab einen weiteren Anhaltspunkt.
- Das Sicherheitskonzept von *ActiveX Controls* wird durch sein offenes Konzept den hohen Anforderungen für WWW-basierte Clients nicht gerecht.

Somit kommen nur CORBA-kompatible Komponenten oder (Enterprise) Java Beans als Komponentenmodell in Frage.

[1] CORBA (*Common Objekt Request Broker Architecture*) ist ein von der OMG (*Object Management Group*) definierter Standard für Anwendungen in verteilten, heterogenen Systemen.

4.2 Auf Java Beans basierende Komponenten im Vergleich zu Enterprise Java Beans

Serverseitige Applikationen sind eine echte Stärke von Java. Leider wurde mit den Namen Enterprise Java Beans (EJB) und Java Beans eher Verwirrung erzeugt, anstatt eine klare Trennung der Konzepte auch durch den Namen zu definieren. Im Folgenden wird kurz der Unterschied zwischen EJB und Java Beans erklärt. Weiterhin werden die zwei Komponentenkonzepte näher nach spezifischen Kriterien untersucht. Ziel ist es, eine Entscheidung herbeizuführen, welches Konzept verwendet wird bzw. sich in die vorhandene Architektur gut einfügt.

Java Beans sind gemäß Sun Microsystems Komponentensoftware in der Programmiersprache Java. Die Beans befinden sich dabei in einem Container, der zusätzliche Funktionalität zur Verfügung stellt und die Interaktionen mit anderen Objekten regelt. Technisch gesehen handelt es sich jedoch bei *Java Beans* um eine Möglichkeit, den Eigenschaften von Objekten eine portable Schnittstelle zu geben. Von verschiedenen Entwicklungswerkzeugen und anderen Java Beans kann über diese Schnittstellen auf die Bean zugegriffen werden. Java Beans entsprechen dem Konzept von ActiveX, bieten jedoch Plattformunabhängigkeit.

Enterprise Java Beans sind im Gegensatz zu Java Beans für Clients nicht sichtbare serverseitige Komponenten, die mit dem Client kommunizieren, um ihnen einen Dienst zur Verfügung zu stellen. Sie können aber auch eine Funktion ausüben und z.B. fortwährend eine Berechnung durchführen, ohne dabei in Interaktion mit anderen Komponenten treten zu müssen. Der Client kann beispielsweise ein Java Programm, eine Java Bean oder sogar ein C++ Programm sein. Enterprise Java Beans werden zwischen sitzungsbasierten Session Beans und permanenten Entity Beans unterschieden. *Session Beans* sind EJB Komponenten, die kurzlebige Services implementieren. Sie werden vorwiegend benutzt, um Programm- und Fachlogik über ihre Schnittstellen Clients zur Verfügung zu stellen. Eine Session Bean ist häufig nur an einen Client gebunden. *Entity Beans* stellen Daten einer Daten-

bank dar. Jede Entity Bean besitzt eine eindeutige Identität und repräsentiert genau einen Datensatz. Sie können von mehreren Clients gleichzeitig aufgerufen werden.

4.2.1 Fähigkeit zum Aufruf entfernter Methoden

Die Informationsschicht soll sowohl von der fachlichen Anwendungsschicht als auch von einem entfernten Clients aufgerufen werden können. Dies erfordert von den Komponenten zwingend die Eigenschaft, ihre öffentlichen Methoden entfernten Clients und Komponenten zu propagieren. Java Beans bieten nicht die Möglichkeit, mit entfernten Clients zu kommunizieren. Deshalb ist es erforderlich, zusätzliche Funktionen zur Kommunikation der Bean mit einem Client, beispielweise über CORBA, RMI oder IIOP, manuell zu implementieren.

EJBs sind mit ihren speziellen Schnittstellen (Interfaces) gut für eine verteilte Kommunikation ausgerüstet. Dem Client präsentieren sich EJB-Komponenten zunächst über ein *Home-Interface*, das Operationen, wie das Erzeugen (`create`), Löschen (`remove`) und eine gezielte Suche über den Primärschlüssel erlaubt (`findByPrimaryKey()`, `findByLastName()` ...). Eine Referenz auf das Home-Interface einer Komponente wird durch die Programmierschnittstelle des Verzeichnisdienstes *JNDI* [2] zur Verfügung gestellt. Das *Remote-Interface* bietet die Möglichkeit, nach bereits gelungener Instantiierung (also z.B dem Auffinden einer Entity Bean über einen vorhandenen Primärschlüssel) auf die öffentlichen Methoden der Komponente zuzugreifen.

Der Lebenszyklus einer Entity Bean wird über den Container der EJB kontrolliert. Graphik 4.1 illustriert die verschiedenen Abschnitte im Lebenszyklus einer

[2] Das JNDI - *Java Naming Directory Interface*- ist eine Standarderweiterung der Java Plattform, die es auf der Java Technologie beruhenden Applikationen erlaubt, über ein gemeinsames Interface auf die verschiedensten Namens und Verzeichnisdienste innerhalb eines Unternehmens zurückzugreifen. Als Teil der *Java Enterprise API* erlaubt JNDI ein nahtloses Verbinden zu heterogenen unternehmensweiten Namens - und Verzeichnisdiensten.
Quelle: SUN Microsystems, Übersetzung des Autors

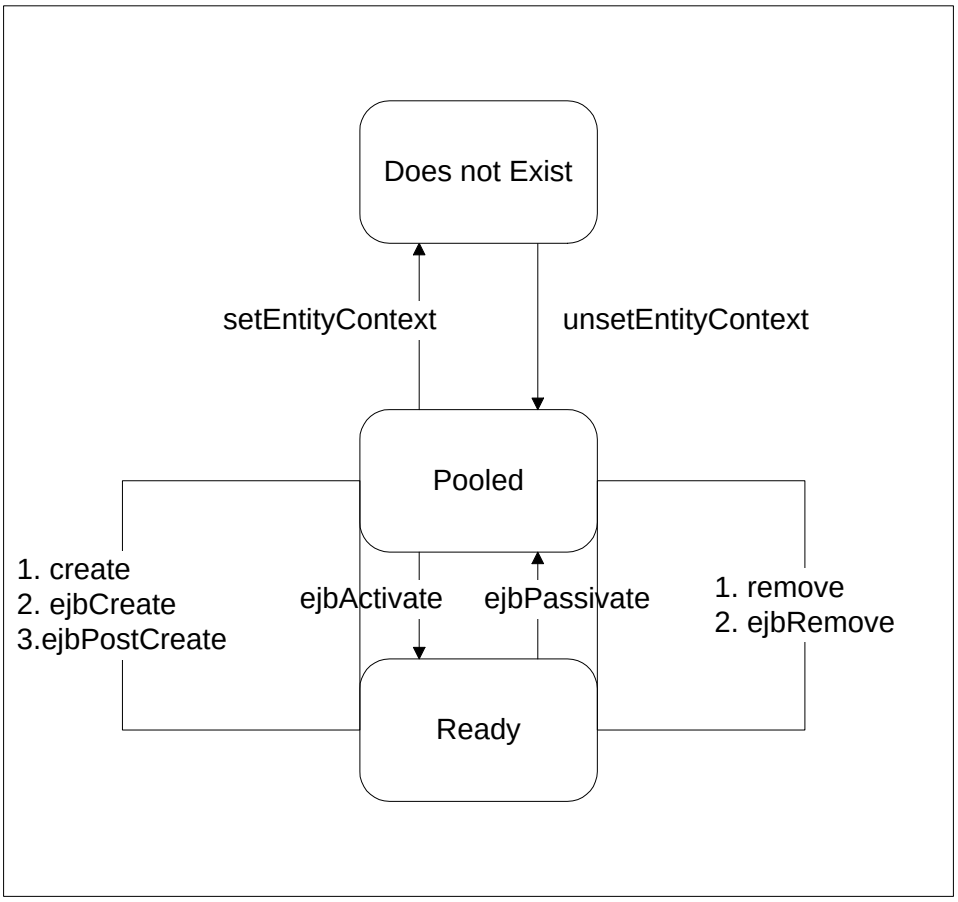

Abbildung 4.1: Lebenszyklus einer Entity EJB

Entity Bean. Nachdem der EJB Container die Instanz der Entity Bean erzeugt hat, wird die Methode `setEntityContext` der Entity Bean aufgerufen. Diese Methode gibt den Kontext des EJB-Containers an die Bean weiter. Nach der Instantiierung wird die Entity Bean in einen „Pool“der verfügbaren Instanzen gehalten. Solange sich die Bean in diesem Zustand (*pooled stage*) befindet, ist die Instanz nicht an ein spezielles, einer bestimmten Identität zugeordnetes EJB-Objekt gebunden. Der EJB Container weist eine Identität einer Entity Bean zu, wenn diese in den nächsten Zustand (*ready stage*) übergeht. Dabei gibt es zwei Möglichkeiten, diesen Zustand zu erreichen. Einerseits kann der Client eine „Erzeugermethode“aufrufen, wie `ejbCreate` oder `ejbPostCreate`. Andererseits kann der EJB-Container die

Methode `ejbActivate` aufrufen. Solange die EJB sich in diesem Zustand befindet, können die im Remote-Interface definierten öffentlichen Methoden aufgerufen werden. Um wieder in den nächst tieferen (*pooled*) Zustand zu gelangen, sind ebenfalls zwei Wege möglich. Der Client kann eine Methode zum Entfernen aufrufen - `ejbRemove`- oder der EJB-Container ruft die Methode `ejbPassivate` auf. Am Ende des Lebenszyklus der EJB wird die Instanz der EJB durch den Container aus dem Pool entfernt und die Methode `unsetEntityContext` aufgerufen.

Zusammenfassend kann bemerkt werden, daß Entity Beans aufgrund ihres vordefinierten standardisierten Schnittstellenkonzeptes für den Aufruf entfernter Methoden und des damit verbundenen geringeren Implementierungsaufwandes einen klaren Vorteil gegenüber Java Bean basierten Komponenten in Verbindung mit CORBA-Technologien für den Client/Server-Einsatz besitzen.

4.2.2 Migrationsfähige und austauschbare Komponenten

Eine weitere wichtige Eigenschaft der Komponenten ist die Fähigkeit, in anderen Laufzeitumgebungen bzw. Plattformen zu funktionieren. Da der Markt der Middlewaresoftware sehr schnellebig ist und man häufig mit neuen Produkten und Versionen konfrontiert wird, sollten die Komponenten sich gut auf andere Systeme übertragen lassen.

Das Java-Bean-Konzept ist durch seine auslesbaren Komponenten und dem plattformübergreifenden „Pure-Java“ Ansatz sehr gut migrierbar. Bei einem Wechsel des Applikationsservers zu einer höheren Version könnte es jedoch eventuell durch fehlende Standardisierung des Aufrufes entfernter Methoden zu Inkompatibilitäten kommen bzw. ein Produkt eines anderen Herstellers würde keinen fremden Standard (in diesem Fall den des *Sybase Enterprise Application Server*) unterstützen.

EJBs entsprechen dem J2EE Standard (siehe [33]) und werden von mehreren Middlewaresystemen unterstützt (siehe auch [25]). Da die EJB in einem Contai-

ner abläuft, sind Aufgaben wie Transaktionssteuerung, *Interception*[3] und diverse Rückrufmethoden, wie das Aktivieren und Deaktivieren der Komponente durch den Container standardisiert. Selbst der Zugriff auf Funktionen des Containers durch die Komponente findet über ein definiertes Kontextobjekt Unterstützung.

Auch hier besitzt das Konzept der *Enterprise Java Beans*, insbesondere durch seinen vom Markt akzeptierten Standard und damit eine damit verbundene Austauschbarkeit des Containers über verschiedene Hersteller einen deutlichen Vorteil.

4.2.3 Transaktionsmanagement

Transaktionssteuerung - oder Transaktionsmanagement - bezieht sich auf die Fähigkeit eines relationalen Datenbank-Managementsystems, Datenbanktransaktionen auszuführen.

Transaktionen sind Arbeitseinheiten, die als Gruppe in einer logischen Reihenfolge auszuführen sind. Der Begriff Arbeitseinheit bedeutet, daß eine Transaktion einen Anfang und ein Ende hat. Wenn eine Fehlfunktion während der Transaktion auftritt, läßt sich die gesamte Arbeitseinheit bei Bedarf abbrechen. Ist die Transaktion erfolgreich, wird die gesamte Arbeitseinheit in der Datenbank gespeichert.

Die zu generierenden Komponenten führen Datenbankzugriffe aus und müssen ein Transaktionsmanagement implementieren. Das Transaktionsmanagement kann vom Client, von der Middleware oder von der Komponente gesteuert werden. Java Beans implementieren keine speziellen Schnittstellen für Transaktionen. Jedoch kann manuell ein `Transaktionshandling` implementiert werden. Da das `Transaktionshandling` jedoch sehr komplex zu behandeln ist, wäre es wünschenswert, diese Funktionalität von einem Produkt bereitgestellt zu bekommen.

[3] Bereitstellung der Laufzeitumgebung und Delegierung des Client - Aufrufs an die Implementierungsklasse.

Tatsächlich stellt der Softwaremarkt vielfältige Transaktionsmanagementsysteme für den Einsatz mit Java und JDBC bereit. Jeder große Hersteller hat mindestens ein System in seinem Portfolio. Jedoch beinhaltet der Kauf eines weiteren Produktes gleichzeitig oft weitere Schwierigkeiten mit der Adaption und Anpassung an eine bestehende meist sogar historisch gewachsene Architektur. Deshalb hat sich der Autor für EJBs entschieden, die ein Transaktionsmanager-Interface gemäß der J2EE Spezifikation von SUN implementieren. Dieses Interface wird von den Middlewareherstellern benutzt, um die Transaktion zu steuern. Das EJB-Transaktionsmodell basiert auf dem OTS (*Object Transaction Service*)[4] Model der CORBA Spezifikation. Es ist vereinfacht worden, um die Spezifikationen für die Lizenznehmer der Spezifikation nicht zu schwergewichtig zu gestalten (und somit auch möglichst viele Unternehmen zu einer Lizenznahme zu bewegen). Dies bedeutet jedoch auch, daß einige Features (z.B. verschachtelte Transaktionen) nicht enthalten sind.

Das EJB Transaktionsmodell unterstützt Transaktionen über die JTS *(*Java Transaktion Services API), die eine Abbildung des CORBA-OTS darstellen. Das Transaktionsverhalten wird im EJB- Deploymentdescriptor spezifiert (siehe auch Kapitel 5.5.4). Die Transaktion kann entweder vom Container der EJB oder von der EJB selbst behandelt werden. Tabelle 4.1 zeigt für die verschiedenen EJB Spezifikationen mögliche Attribute und deren Auswirkung[5].

[4] System zur Steuerung verteilter Transaktionen. Das OTS basiert auf dem vom X/Open veröffentlichten Distributed Transaction Protocol (DTP). Dabei regelt ein zentraler Transaktionsmanager die Transaktionskontrolle und -abwicklung. Bei Abschluß der Transaktion wird nach dem 2-Phase-Commit-Protokoll verfahren.

[5] Die verwendete Middleware Jaguar CTS Version 3.5 unterstützt in dieser Version nur die EJB Spezifikation 1.0. Der Autor erwartet jedoch, daß Sybase noch innerhalb der Erstellung dieser Arbeit seine Software auf den Stand der EJB Spezifikation 1.1 hebt. Deshalb kann die hier beschriebene Implementierung des Deployments von der tatsächlichen geringfügig abweichen.

EJB 1.1 Spezifikation	EJB 1.0 Spezifikation	Erläuterung
REQUIRED	TX_REQUIRED	Container basierte Transaktion. Der Server startet entweder eine neue Transaktion auf Anfrage des Nutzers oder führt eine gerade benutzte Transaktion fort, die beim Start der Bean begonnen wurde.
REQUIRESNEW	TX_REQUIRED_NEW	Container basierte Transaktion. Der Server startet eine neue Transaktion auf Anfrage des Nutzers. Wenn eine existierende Transaktion diese Transaktion aufgerufen hat, wird die existierende Transaktion solange ausgesetzt, bis die aktuelle beendet ist.
Specified as Bean transaction-type in deployment descriptor	TX_BEAN_MANAGED	Bean basierte Transaktion. Über den Zugriff auf den Transaktionskontext wird der Beginn, Commit, oder Rollback gesteuert.
SUPPORTS	TX_SUPPORTS	Wenn das Programm, das die Bean aufgerufen hat, gerade in einer Transaktion abläuft,wird diese Bean in die Transaktion eingefügt.
NEVER	TX_NOT_SUPPORTED	Wenn das Programm, das die Bean aufgerufen hat, gerade in einer Transaktion abläuft,wird diese Transaktion solange ausgesetzt, bis der Methodenaufruf in der aufgerufenen Bean beendet ist. Ein Transaktionskontext wird nicht erzeugt.
MANDATORY	TX_MANDATORY	Das Transaktionsattribut für diese Bean wird gesetzt, wenn eine andere Bean eine Methode für diese Bean aufruft. In diesem Fall übernimmt die Bean das Transaktionsattribut der aufrufenden Bean. Sollte die aufrufende Bean kein Attribut gesetzt haben, wird von der aufgerufenen Methode eine Ausnahme vom Typ TransactionRequiredexception erzeugt.

Tabelle 4.1: Attribute der Transaktionssteuerung für verschiedene EJB Spezifikationen, Quelle : Sun Microsystems[34]

4.2.4 Isolation Level Descriptions

Eine EJB benutzt einen *Isolation Level*, um Informationen über ihr Zusammenspiel mit gemeinsam genutzten Daten und anderen ausgeführten Prozessen, die ebenfalls auf die Daten zugreifen, zu erhalten. Wie der Name impliziert, gibt es mehrere Stufen, wobei `TRANSACTION_SERIALIZABLE` die höchste Stufe der Datenintegrität bereitstellt. Isolation Level sind von der verwendeten Datenbank abhängig. Häufig werden nur einige Isolation Level von der Datenbank unterstützt. Deswegen folgt nun eine Analyse der von *Enterprise Java Beans* gemäß der Spezifikation 1.0 und der verwendeten *Sybase* Datenbank *Adaptive Server Enterprise (ASE) Version 11.9.2* **gemeinsam unterstützten** Isolation Levels. Die Analyse erfolgt auf Basis der Datenbank *Manuals* von *Sybase* (siehe auch [13]) und der EJB-Spezifikation von Sun Microsystems (siehe auch [26]) durch den Autor. Zuerst wird der Isolation Level gemäß der EJB-Spezifikation aufgeführt. Dann wird versucht, ihm einen von der Datenbank unterstützten Isolation Level zuzuordnen.

- `TRANSACTION_SERIALIZABLE`: Diese Stufe gewährleistet maximale Datenintegrität. Die Bean erhält den exklusiven Zugriff auf die Daten. Keine andere Transaktion kann weder lesend noch schreibend auf die Daten zugreifen. *Serialisierbar* in diesem Kontext bedeutet die serielle Abarbeitung aller Anweisungen, dabei ist die Reihenfolge der Abarbeitung der Anweisungen für das Endergebnis der Transaktionen unwesentlich. Diese Einstellung stellt die langsamste aller Einstellungen dar. Sie sollte gewählt werden, wenn Leistung nicht im Vordergrund steht.

  Diese Stufe entspricht dem ASE Isolation Level 1.

- `TRANSACTION_REPEATABLE_READ`: In dieser Stufe können die Daten während der Transaktion zwar von anderen Transaktionsprozessen gelesen, jedoch nicht modifiziert werden. Der gelesene Wert bleibt solange unverändert, solange die zuerst initiierte Transaktion ihn nicht verändert zurückschreibt.

Für diese Stufe wird kein ASE Isolation Level unterstützt.

- `TRANSACTION_READ_UNCOMMITTED`: In dieser Stufe können Daten von anderen Prozessen gelesen werden und zwar unabhängig, ob die erste Transaktion ein `rollback` oder `commit` ausführt. Andere Transaktionen erfahren nicht, ob der Vorgang abschließend mit `rollback` oder `commit` beendet wurde.

  Diese Stufe entspricht dem ASE Isolation Level 0.

- `TRANSACTION_READ_COMMITTED`: Daten können in dieser Stufe von anderen Transaktionen erst dann wieder gelesen werden, wenn der erste Transaktionsprozess der Transaktion entweder ein `rollback` oder `commit` ausführt.

  Diese Stufe entspricht dem ASE Isolation Level 3.

Die Analyse zeigt auf, daß die Datenbank drei der vier Isolation Level der EJB-Spezifikation unterstützt. Es kann nun ein Isolation Level für das Transaktionsverhalten der EJB ausgewählt werden. Dies wird in Abschnitt 5.2.3 beschrieben.

Zusammenfassend kann gesagt werden, das EJBs ein komplexes Transaktionsmanagement bereitstellen, das entweder von einer Middleware kontrolliert-*container managed*- oder in der Bean -*bean managed*- vom Entwickler implementiert wird. Letzteres bedeutet einen ähnlichen Aufwand wie für eine Java Bean mit externer Transaktionssoftware.

4.2.5 Skalierbarkeit und Hochverfügbarkeit der Komponenten

Die Skalierbarkeit und Hochverfügbarkeit hängt im wesentlichem vom Applikationsserver bzw. der serverseitigen Software ab. Eine Hochverfügbarkeit wird beim hier verwendeten Applikationsservers *Jaguar CTS Server 3.5* über Cluster erreicht, die sich bei Ausfall eines Servers gegenseitig ergänzen. Die *Skalierbarkeit*, also die Fähigkeit sehr viele Instanzen eines Komponententypes (10.000 und mehr) zu managen, wird durch zwei Funktionen erreicht:

- Der Möglichkeit der vorzeitigen Deaktivierung einer Komponente. Der Vorteil in einer vorzeitigen Deaktivierung liegt in der frühen Freigabe der von der Komponente benutzten Ressourcen, z.B. einer Datenbank. Da die Anzahl der gleichzeitigen Zugriffe auf Tabellen oder Entitäten in einer Datenbank normalerweise limitiert ist, erreicht man, daß andere Komponenten diese Ressourcen „vorzeitig“benutzen können.

- Der Optimierung des Verbindungsaufbaus zur Datenbank durch Instance Pooling. Das Öffnen einer neuen Verbindung zur Datenbank nimmt sehr viel Zeit in Anspruch. Um trotzdem zeiteffizient auf Datenbanken zugreifen zu können, benutzt die EJB-Spezifikation das Verfahren des *Instance Pooling*. Dabei werden Instanzen der EJB-Komponenten in einem Pool bereitgehalten. Diese Instanzen öffnen Verbindungen zur Datenbank ohne an einen Client gebunden zu sein. Ein Client greift auf die schon bestehenden Instanzen und damit bestehenden Datenbankverbindungen zu (siehe auch Roger Sessions in [21]). Das Verfahren stellt sicher, das immer genügend Instanzen für einen eventuellen Clientzugriff bereitgestellt werden.

Um diese Vorteile der Middleware zu benutzen, ist es notwendig, daß die Java Klassen bestimmte auf die Middleware speziell angepaßte Schnittstellen implementieren. Damit kann jede von der Middleware beherbergte Komponente von diesem Vorzug profitieren.

4.2.6 Umfang und Anzahl der zu generierenden Komponenten

Der Umfang ist bei einer Codegenerierung eine nicht zu unterschätzende Unbekannte. Im Hinblick auf den Einsatz der Strukturen ist die Menge der verschieden Methoden, die Anzahl der Klassen/Dateien und der Umfang innerhalb der Methoden in diesem Kontext relevant.

Entity EJBs bestehen aus folgenden Dateien:

- der eigentlichen Bean
- dem Remote-Interface, es beinhaltet die öffentlichen Methoden der Entity Bean
- dem Home-Interface, es stellt Methoden zum Auffinden der Entity-Bean bereit
- einer Klasse, die den Primärschlüssel beschreibt
- dem sogenannten *Deployment Descriptor*, der z.B. Eigenschaften über die Transaktionsfähigkeit der Bean beinhaltet
- dem Primary-Key, einer Klasse, die den Schlüssel zur eindeutigen Identifikation der Bean definiert

Java Beans besitzen eine solche Aufteilung nicht. Sie benötigen jedoch spezielle Funktionen und Erweiterungen, um ihre Methoden dem Client zu propagieren. Damit hängt die Komplexität sehr von der verwendeten Technologie bzw. dem Umfang ihrer zusätzlichen Implementierung ab.

4.2.7 Entscheidung für Entity Beans als Komponentenmodell

Entity Beans, als Baustein für die Entwicklung verteilter Applikationen, sind hochkomplexe Komponenten. Die J2EE Spezifikation erlaubt es, daß EJBs momentan von vielen Entwicklungsumgebungen und Applikationsservern unterstützt werden. Durch das Container-Komponente Prinzip werden wichtige Funktionen wie Transaktionsverhalten, Caching, Sicherheit, Datenbankzugriff und Kommunikation mit entfernten Clients durch den Hersteller des Containers bereitgestellt. Aufgrund dieser Vorteile eignen sich EJBs als Komponentenmodell für diese Arbeit.

Java Beans sind eher graphisch orientierte Komponenten. Sie können ihre Methoden nicht an entfernte Clients weitergeben. Ebenfalls besitzen sie keine zusätz-

lichen Funktionen, wie Transaktionsmanagement, Caching und Unterstützung für den Datenbankzugriff.

Kapitel 5

Implementierung

Dieses Kapitel ist der Umsetzung der vorgestellten Konzepte gewidmet. Zuerst werden in einem Überblick kurz alle Bausteine der Implementierung und ihre Beziehungen zueinander vorgestellt. Danach werden besondere Ausschnitte und Implementierungsschwerpunkte der einzelnen Bausteine näher erläutert. Der dabei vorgestellte Code dient vorwiegend zur Illustration. Der letzte Abschnitt ist den Besonderheiten des Prototypen bzw. der Templates gewidmet.

5.1 Überblick über die Komponenten der Implementierung

Die Entwicklung eines Systems zur Generierung einer adaptiven Informationsschicht setzt eine komplexe Systemarchitektur voraus. Um diese hohe Komplexität zu beherrschen, wurde eine aus einzelnen Komponenten bestehende mehrstufige Architektur gewählt. Die einzelnen Komponenten können rechnerübergreifend und netzwerkweit miteinander kommunizieren. Sie werden dabei von den Systemen *Ressourcenmanager* und *Codegenerator*, teilweise mehrfach, verwendet. Im Folgenden soll nicht auf die Systeme, sondern vielmehr auf eine Zuordnung der Komponenten zu den einzelnen Systemen eingegangen werden.

Die *clientseitigen Komponenten* des Systems Ressourcenmanager (siehe auch Kapitel 5.5 und Grafik 3.7) stellen eine Schnittstelle zu den die Generierung und den Datenbankzugriff steuernden Serverkomponenten dar. Zu den clientseitigen Komponenten des Systems Ressourcenmanager zählen

- graphische Editoren für Ressourcen, Klassen und Eigenschaften,
- Kontrollmonitor für Transaktions- und Generierungprozesse und
- eine Steuerungskomponente für die Generierung.

Die *serverseitigen Komponenten* dieses Systems umfassen neben der Steuerungskomponente des Ressourcenmanagers Schnittstellen zum System Codegenerator sowie Systemkomponenten zum Zugriff auf die Datenbank. Das System Codegenerator (siehe Grafik 5.1) dient der Erzeugung von Java-Quellcode. Das System besteht aus folgenden Komponenten:

- eine serverseitigen Schnittstelle zum Ressourcenmanager über die die Generierung angestoßen und kontrolliert wird (Grafik 3.12)
- der Komponente *Ressource-Meta-Generator*. Ihre Aufgabe ist es, das Modell der Ressourcen, Klassen und ihrer Eigenschaften aus der Datenbank auszulesen und daraus ein textbasiertes Modell zu erstellen (Grafik 3.11). Dazu erfolgen Zuweisung von Ressourcenklassen zu Ressourcen und Zuweisung von Eigenschaften zu Ressourcenklassen sowie eine Auflistung der Ressourcen mit ihren (abgeleiteten) Eigenschaften. Aus diesen Angaben erfolgt die Generierung eines Metamodells. Das Metamodell liegt als strukturierte Textdatei im XML Syntax vor. Ein Beispiel für ein solches Modell ist in Abschnitt 5.3.1 aufgeführt.
- Eine Entity-EJB besteht aus mehreren einzelnen Dateien, der Entity-Bean-Implementierung, dem Home-Interface und dem Remote-Interface. Diese

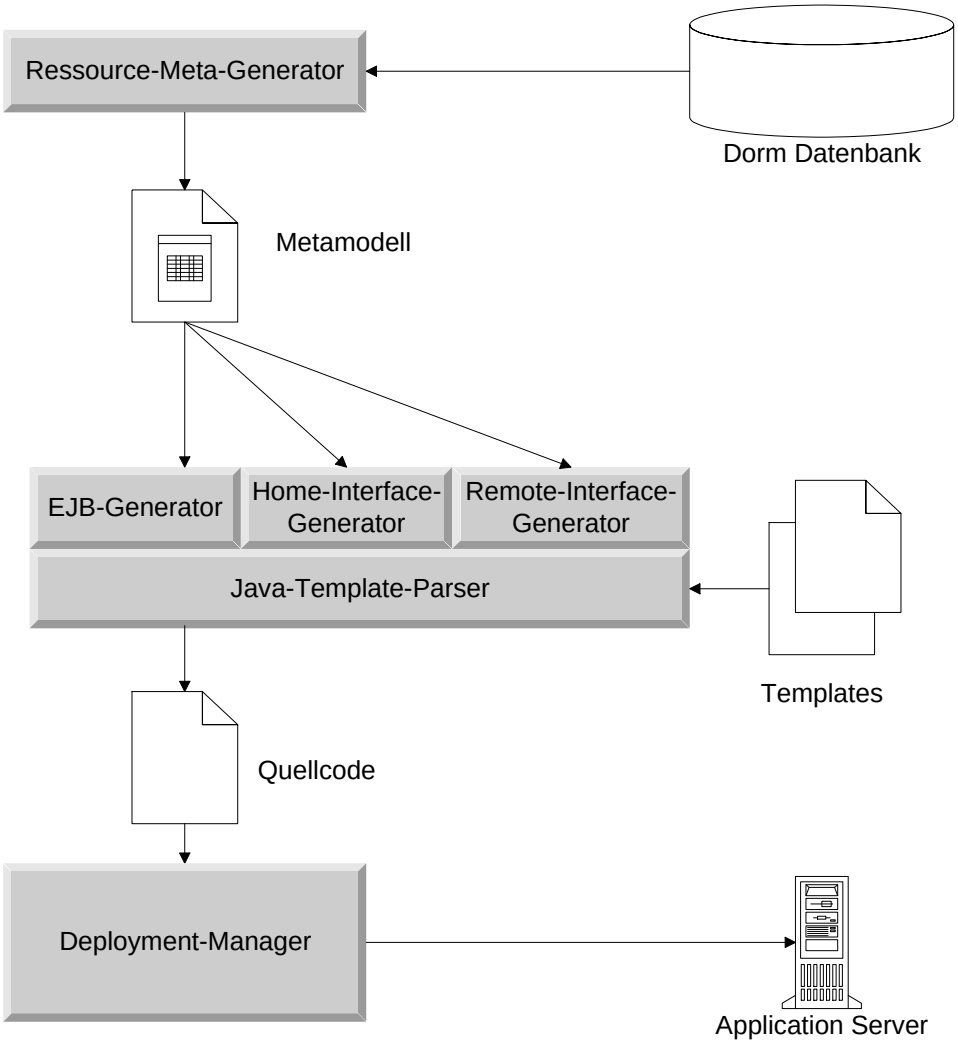

Abbildung 5.1: System Codegenerator

einzelne Teile werden in mehreren Durchläufen durch das System Codegenerator generiert. Dazu besteht das System Codegenerator aus mehreren Teil-Komponenten (`EJB-Generator`,`Home-Interface-Generator` und `Remote-Interface-Generator`), die jeweils spezielle Regeln für die Generierung der einzelnen Teile der EJB implementieren. Diese Regeln sind für jeden Teil der EJB unterschiedlich. Gemeinsame Aufgaben, wie das Analysieren der Java-Templates, das Ersetzen der Schlüsselwörter in den Templates und das Schreiben des fertigen Quellcodes, werden von der Komponente `Java-Template-Parser` übernommen.

- Der Deployment-Manager, ist für ein nach der EJB-Spezifikation konformes Deployment verantwortlich. Dazu erzeugt er einen Deployment-Descriptor gemäß EJB Spezifikation 1.0, kompiliert und dokumentiert den generierten Java Quellcode und packt die kompilierten Klassen sowie den Deployment-Descriptor in ein EJB-Archiv.

Diese beiden Systeme werden von diversen Systemkomponenten unterstützt, die unter anderem den Zugriff auf das Dateisystem regeln, Komprimierungsalgorithmen bereithalten, die Schnittstelle zum Java-Compiler kapseln und eine Schnittstelle zur Datenbank bereitstellen. Im nächsten Abschnitt werden spezielle Java Beans als eine solche Schnittstelle zum Datenbanksystem vorgestellt.

5.2 Java Beans als Bindeglied zwischen Datenbank und den Komponenten

Die DORM-Datenbank bietet als externe Schnittstelle Stored Procedures an, die nach Aufruf mit eventuellen Parametern ihre Ausgabe in Listenform realisieren. Beispielsweise hat ein Aufruf zur Auflistung aller Email-Aliases des Nutzers **aloeser** folgenden Syntax:

```
dorm list_strings aloeser ldap alias
```

wobei **dorm** ein Kommandozeileninterface zum Zugriff auf die Datenbank ist. Eine mögliche Ausgabe dieses Aufrufs könnte dann wie folgt aussehen:

```
aloeser@fhtw-berlin.de aloeser@mail.rz.fhtw-berlin.de
hallo.alex@fhtw-berlin.de aloeser@mail.rz.fhtw-berlin.de
ReturnCode 0: (operation_successful)
```

Die Ausgabe im datenbanktypischen Tabellenformat schließt mit dem Rückgabewert *-Return Code-* ab, der den Verlauf der Transaktion mit einer zweistelligen Ziffer und einer Fehlermeldung belegt.

5.2.1 Kapselung der Stored Procedures in Java Beans

Die Eingabe und das Ergebnis der Stored Procedure soll für kommunizierende Komponenten les - und schreibbar sein. Mit der oben demonstrierten Art der Parameterübergabe und dem Auslesen der Rückgaberesultate kann man in Java schlecht arbeiten. Hier greift das Komponentenkonzept der Java Beans ein [29]. Jeder Stored Procedure ist eine Java Bean gleichen Namens zugeordnet (in diesem Fall heißt die Java Bean `ListStrings`).

Listing 5.1: Die Java Bean ListStrings (Ausschnitt)

```
1 package dorm.server.database;
2 import java.sql.*;
3 public class ListStrings extends java.lang.Object {
4     private String userName=null;
5     private String resourceName=null;
6     private String propertyName=null;
7     final private String spCall = "{?= call  list_strings  ?,?,?}";
8     final private int invalid = -1;
9     private ResultSet rs;
```

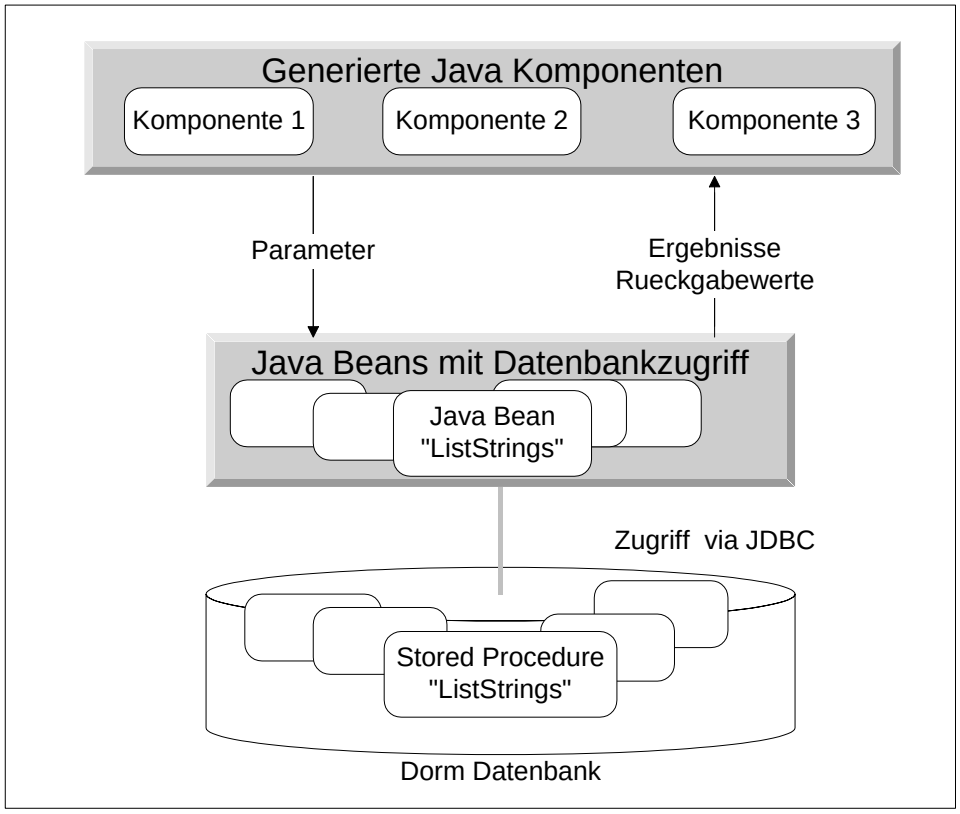

Abbildung 5.2: Schnittstellen der Java Beans

```
10  private Connection connection;
11  private CallableStatement call = null;
12  private int returnCode;
13
14  // Konstruktor
15  public ListStrings (){..}
16
17  // Methoden
18  public void setUserName(String userName) {...}
19  public void setResourceName(String resourceName) {...}
20  public void setPropertyName(String propertyName) {...}
21
22  public int getReturnCode(){
23      return this.returnCode;
24  }
25
26  public dorm.generator.utils.StringList getSetting (){
27      dorm.generator.utils.StringList value = new dorm.generator.utils.StringList ();
28      String valueColumn = " string_value ";
29      Connector dormDb =new Connector();
30      connection= dormDb.getCachedDbConnection();
31      ResultSet rs = executeQuery();
32      try
33      {
34          while(rs.next())
35              value.addElement(rs.getString (valueColumn).trim ());
36      }
37      catch (SQLException sqlE){..}
38      finally {
39          try{
40              this.returnCode = call.getInt (1);
41              call.close ();
42              dormDb.close();
43          }
44          catch (SQLException e){..}
45      }
```

```
46          return value;
47      }
48
49      private ResultSet executeQuery(){
50          rs = null;
51          try{
52              call = connection.prepareCall(spCall);
53              call.registerOutParameter(1, java.sql.Types.INTEGER);
54              call.setString(2, this.userName);
55              call.setString(3, this.resourceName);
56              call.setString(4, this.propertyName);
57              rs=call.executeQuery();
58          }
59          catch(SQLException sqlE){...}
60          return rs;
61      }
62  }
```

Diese Java Beans sind in dem Package `dorm.server.database` zusammengefaßt (Zeile 1). Der Name der Stored Procedure ist der Bean bekannt (Zeile 7). Über Methoden der Bean können sowohl Parameter der Stored Procedure gesetzt, als auch Ergebnisse ausgelesen werden (Zeilen 17 -20). Die Abfrage der Datenbank erfolgt über eine JDBC-Schnittstelle (Zeilen 49 -62 und Zeilen 31 - 45). Der Rückgabewert kann über die Methode `(int).getReturnCode()` abgefragt werden (Zeilen 22- 24). Eventuelle Rückgaberesultate der Stored Procedure werden in spezifische Java Datentypen konvertiert (Zeile 27) und können über öffentliche Methoden der Bean abgefragt werden (Zeilen 26-47).

5.2.2 Die Verbindung zur Datenbank

Die Verbindung zur Datenbank wird über die Bean `dorm.sever.database.Connector` realisiert. Wahlweise kann hier eine neue Verbindung vom Typ `java.sql.Connection` zur Datenbank mit der Methode `dorm.sever.database.`

`Connector.getDbConnection()` oder eine bereits von der Middleware im Verbindungspool gehaltene Verbindung über die Methode `dorm.sever.database.Connector.getCachedDbConnection()` angefordert werden.

Listing 5.2: Anforderung einer bestehenden Verbindung von der Middleware

```
1   public java.sql.Connection getCachedDbConnection()
2   {
3       String _server_url =
4           dbDriver + ":" +
5           dbVendor + ":" +
6           dbProtocol + ":" +
7           dbServer + ":" +
8           dbPort + "/" +
9           dbDatabase;
10      // JCM ist der Jaguar Cache Pool
11      try {
12          dormDbCache = JCM.getCache(dbUser, dbPassword, _server_url);
13      } catch (Exception e){
14          writeMsg(true,"Connector_getCache()_exception"+ e.getMessage());
15          dormDbCache = null;
16      }
17      // Wenn wir keine Verbindung erhalten können, Fehler ausgeben.
18      // Die Exception wird Jaguar veranlassen eine Fehlermeldung an den Stub weiterzugeben.
19      if (dormDbCache == null){
20          writeMsg(true,"Connector:_Could_not_access_connection_cache.");
21          writeMsg(false, "_getCachedConnection:_Cache_may_not_be_configured
22          properly in Jaguar Manager.");
23      }
24      // Wir rufen getConnection (int) auf um eine Verbindung vom Cache der Middleware zu
25      // erhalten. Dabei benutzen wir das JCM_WAIT Flag, da getConnection (JCM_WAIT) "null"
26      // zurückgeben kann, falls eine Unterbrechung erfolgt.
27      // Die Schleife sorgt dafür das in diesem Fall ein weiterer Versuch unternommen wird.
28      Connection conn = null;
29      int i = 0;
30      while (conn == null){
```

```
31              try{
32                  conn = dormDbCache.getConnection(JCMCache.JCM_WAIT);
33              }
34              catch (Exception e){
35                  writeMsg(true," Actually no connection avaible , will try again (Try No.  "+i+"):"
36                  +e.getMessage ()); }
37              i++;
38          }
39          globalConnObj=conn;
40          cachedCon = true;
41          return conn;
42      }
```

Ein bestehender Pool von Datenbankverbindungen in der Middleware, der über das Interface `JCM.getCache(dbUser,dbPassword,_server_url)` (Zeile 12) für ein schnelles Öffnen neuer Verbindungen zur Datenbank sorgt. Der Aufbau der Verbindung erfolgt solange, bis eine Verbindung tatsächlich erfolgt ist (Zeilen 30 bis 38). Dies wird durch das Setzen des Flags `JCMCache.JCM_WAIT` realisiert (Zeile 32).

5.2.3 Bestimmung des Transaktionsverhaltens

Transaktionen im DORM-Modell werden komplett von den Stored Procedures gekapselt. Deswegen werden spezielle von der Midelleware bereitgestellte Transaktionsmodelle nicht benötigt. Das Transaktionsverhalten wird somit auf der Ebene der Datenbank definiert. Für die EJB-Komponenten wurde zunächst ein Transaktionsverhalten nach Isolation Level 1 gewählt (Auf die gemeinsamen Isolation Level von EJBs und der verwendeten Datenbank *Adaptive Server Enterprise* wurde bereits in Kapitel 4.2.4 eingegangen.). Isolation Level 1 bedeutet, daß die EJB den exklusiven Zugriff auf die Daten erhält. Keine andere EJB kann während dieser Zeit auf Daten lesend oder schreibend zugreifen. Die Transaktionen sind serialisierbar, das heißt, die Anordnung der Transaktionsanweisungen hat keine Auswirkung auf

das Ergebnis der Transaktion. Dieses Transaktionsverhalten erlaubt eine maximale Datenintegrität. Diese Einstellung stellt die langsamste aller möglichen Einstellung dar. Sie wurde gewählt, da zu erwarten ist, daß der Zugriff auf gemeinsam genutzte Daten zum selben Zeitpunkt sehr selten vorkommen wird. Leider fehlen zum Zeitpunkt der Arbeit Angaben über den zeitlichen Aufwand für eine Transaktion mit einem solchem Transaktionsmodell. Deshalb wurde die Möglichkeit offengehalten, über den Deployment-Descriptor bei Bedarf ein anderes Transaktionsverhalten festzulegen.

5.3 Die Generierung der Metainformationen

Im Abschnitt 3.4.3 wurde bereits ein Konzept für die Erstellung eines Codegenerators vorgestellt. Ein solcher Generator muß einerseits Schablonen aufweisen, die konstante Teile der Zielsprache abbilden, als auch eine spezielle Generatorsprache, die die Steuerung des Generators übernimmt. Dabei werden an entsprechenden Stellen Informationen aus einem Metamodell benutzt, um Ersetzungen vorzunehmen. Das textbasierte Metamodell wird dabei durch einen Metamodellgenerator aus der Datenbank erzeugt. Im folgenden Abschnitt wird die Syntax dieses Metamodells näher vorgestellt.

5.3.1 Die Syntax des Metamodells

Der generelle Syntax ist an eine XML konforme Schreibweise <Schlüsselwort> Wert </Schlüsselwort> angelehnt.

XML - Extensible Markup Language ist eine Teilmenge von SGML, welche eine äußerst umfangreiche Dokumentenbeschreibungssprache darstellt, die zum standardisierten Dokumentenaustausch benutzt wird. Ein gültiges XML Dokument besteht aus mehreren unterschiedlichen Elementen die nach der XML Syntax angeordnet sind. Desweiteren besteht die Möglichkeit, mit einer sogenann-

ten DocumentTypeDefinition grammatikalische Regeln festzulegen, nach denen die XML Tags angeordnet sein müssen... [5]

Weiterhin wurden Blockanweisungen für Ressourcen und Eigenschaften definiert. Ist ein Nullwert in der Datenbank vorhanden, wird statt des Wertes die Zeichenkette „null “eingesetzt. Die Tabellen 5.1 und 5.2 zeigen alle implementierten Schlüsselwörter (ohne vorstehende <,> und </) und Blockanweisungen auf.

Das folgende Beispiel dient zur Illustration eines Ausschnitts aus dem Metamodell und definiert eine Ressource `UnixServer1` der Superklasse `NIS` mit einer Eigenschaft `DiscQuota` und ihren Parametern.

```
<RESOURCE>
    <RESOURCENAME>UnixServer1</RESOURCENAME>
    <SUPERCLASSNAME>NIS</SUPERCLASSNAME>
    <PROPERTY>
        <PROPERTYNAME>DiscQuota</PROPERTYNAME>
        <PROPERTYLABEL>Plattenspeicher für einen Nutzer</PROPERTYLABEL>
        <PROPERTYTYPE>int</PROPERTYTYPE>
        <PROPERTYMANDATORY>0</PROPERTYMANDATORY>
        <PROPERTYUNIQUE>0</PROPERTYUNIQUE>
        <PROPERTYSIZE>0</PROPERTYSIZE>
        <PROPERTYREFERENZ>null</PROPERTYREFERENZ>
    </PROPERTY>
</RESOURCE>
```

5.3.2 Die Komponente Resource-Meta-Generator

Die Erstellung des Metamodells wird über diese Komponente realisiert. Über eine Verbindung zur Datenbank (siehe Kapitel 5.2.2) werden zuerst Ressourcen und ihre

Schlüsselwort	Typ	Bedeutung
RESOURCENAME	String	Name einer Ressource
SUPERCLASSNAME	String	Name der übergeordneten Klasse
PROPERTYNAME	String	Name einer Eigenschaft
PROPERTYLABEL	String	Informationen zu einer Eigenschaft
PROPERTYTYPE	siehe Tabelle XXX	Datenbank-Datentyp einer Eigenschaft
PROPERTYMANDATORY	int(0..3)	wie eine Eigenschaft gesetzt wird 0 = ohne Beschränkung, Null-Werte sind erlaubt 1 = es sind keine Null- Werte erlaubt 2 = mit Referenzwert 3 = mit einem existierendem Referenzwert
PROPERTYUNIQUE	int(0..3)	Constraints der Eigenschaft 0 = unbestimmt 1 = auf Account eindeutig 2 = auf Ressource eindeutig 3 = auf Klasse eindeutig
PROPERTYSIZE	int	maximale Größe der Eigenschaft 0 = unbestimmt
PROPERTYREFERNEZ	String	Die Referenzeigenschaft, eine Eigenschaft der selben Klasse. In Verbindung mit PROPERTYMANDATORY bei einem Wert von zwei oder drei können für diese Eigenschaft nur Werte gesetzt werden, die Teil der Wertemenge der Referenzeigenschaft sind. Die Anwendung erfolgt meistens bei den DORM-Datentypen *List* oder *Vector*.

Tabelle 5.1: Schlüsselwörter der Metamodell-Transformation

Schlüsselwort	Bedeutung
<RESOURCE>	Blockbefehl markiert den Anfang einer Ressource
</RESOURCE>	Blockbefehl markiert das Ende einer Ressource
<PROPERTY>	Blockbefehl markiert den Anfang einer Eigenschaft
</PROPERTY>	Blockbefehl markiert das Ende einer Eigenschaft

Tabelle 5.2: Blockanweisungen der Metamodell-Transformation

Klassen, unter Berücksichtigung der Metaklassen (siehe Kapitel 5.3.3) ausgelesen. Dann werden die Eigenschaften jeder Klasse erhoben. Die interne Speicherung erfolgt in den Datentypen `dorm.generator.utils.DormProperty` und `dorm.generator.utils.DormReference`. Schließlich wird eine Textdatei in dem in der Konfigurationsdatei definierten Verzeichnis erzeugt. Diese Datei hat das schon in Kapitel 5.3.1 beschriebene Format.

5.3.3 Berücksichtigung der Metaklassen im Resource-Meta-Generator

Durch die Besonderheit der Metaklassen im DORM-Datenmodell (siehe auch 3.3.1) ist es zwingend notwendig, zuerst alle Klassen und ihre Ressourcen auf vorhandene Metaklassen zu analysieren. Ist bei einer Ressource der Parameter *resource_meta* mit einem Ressourcennamen gesetzt, so ist diese Metaklasse zu finden und dieser Ressource zuzuweisen. Die eigentlich kapselnde Klasse der Ressource wird nicht berücksichtigt. Weiterhin sind die Ressourcen der gefundenen Metaklasse nicht weiter zu berücksichtigen und werden aus dem Modell entfernt. Die Methode `java.utils.VectorResourceMetaGenerator.processResource2SuperClassRelation(java.utils.Vector)` implementiert diese Funktionalität.

5.4 Die Generierung der Entity Bean

Die Generierung besteht aus folgenden Teilschritten:

- Parsen des Metamodells
- Auswählen der entsprechenden Java Schablone
- Ausfüllen der Java Schablone mit den aus dem Metamodell gelesenen Parametern
- Schreiben der Java Schablone in die entsprechende Datei

Das aus der Datenbank transformierte Modell dient zur Erstellung der Entity Bean. Dazu werden die im Modell definierten Eigenschaften der Ressourcen von den Codegeneratoren ausgelesen und in die Schablonen an den definierten Stellen eingesetzt, so daß vollständiger Java Code entsteht. Für eine vollständige Entity Bean sind drei Java Objekte notwendig (siehe auch Kapitel 4.2.1):

- Entity Home Interface
- Entity Remote Interface
- Entity Bean Implementation

Für jedes dieser Objekte wird eine eigene Quelldatei durch einen Generator erzeugt.

5.4.1 Automatisierte Programmierung der Generatoren mit JFLex

Allen Generatoren ist es gemein, daß sie die Informationen aus dem Metamodell analysieren und dann je nach Schlüsselwort bestimmte Ereignisse auslösen. Ebenfalls die Komponente Java-Template-Parser die Java-Templates verstehen und analysieren können. Dieser Schritt ist mit dem aus dem Compilerbau bekannten Be-

griff der *lexikalischen Analyse* vergleichbar. *Lexikalische Analyser* sind Programme, deren Ablauf durch *Regular Expressions* in einem Eingabefluß (beispielsweise einer Datei) kontrolliert werden (siehe [17]). Das Schreiben von lexikalischen Analysern - kurz *Lexern*- von Hand stellt einen enormen Aufwand dar. Deshalb wurden Werkzeuge entwickelt, die diese Aufgabe erledigen. Das wohl bekannteste Hilfsmittel dieser Art ist *Lex* [17]. Es generiert Programme zur lexikalischen Analyse für UNIX Systeme für die Programmiersprache C. *Lex* benutzt hierbei eine spezielle Textdatei, die Angaben über das Aussehen des zu generierenden lexikalischen Analysers enthält. Das Hilfsmittel erzeugt daraus eine C-Quellcode Datei, die den eigentlichen Lexer beinhaltet. Das Werkzeug *JFlex* [10] ist ein solcher Generator für die Erzeugung von Lexern in Java. Außerdem ist es selbst komplett in Java geschrieben. Ähnlich dem Programm *Lex* benötigt es eine Datei mit Angaben über den zu generierenden Lexer und erstellt daraus eine Java Quellcode Datei, die den generierten Lexer enthält[1]. Um die Programmierung der EJB-Codegeneratoren zu vereinfachen, wurde JFLex zum automatischen Erzeugen von mehreren Teilen des Systems Codegenerator benutzt (siehe Grafik 5.3).

5.4.2 Definition von Regeln und Aktionen mit JFlex

Die besondere Eigenschaft von *JFlex*, Java Variablen mit Werten zu belegen oder Aktionen in der Programmiersprache Java innerhalb der Spezifikation des Lexer - Generators zu definieren, war ein wesentliches Kriterium bei der Auswahl dieses Lexers. Die Implementation des Home-Interface-Generators ist ein gutes Beispiel für die Funktionalität von *JFlex* und dessen Nutzen bei der Erstellung eines Lexers.

Listing 5.3: Definition des Home-Interface-Generators im JFlex Format (Ausschnitt)

```
1 %state RESOURCENAME_State // Definiere "Unterprogramm"
2 Alpha = [a-zA-Z]
```

[1] Einen Überblick über verschiedene Lexer- und Parser- Generatoren erhält man unter [9].

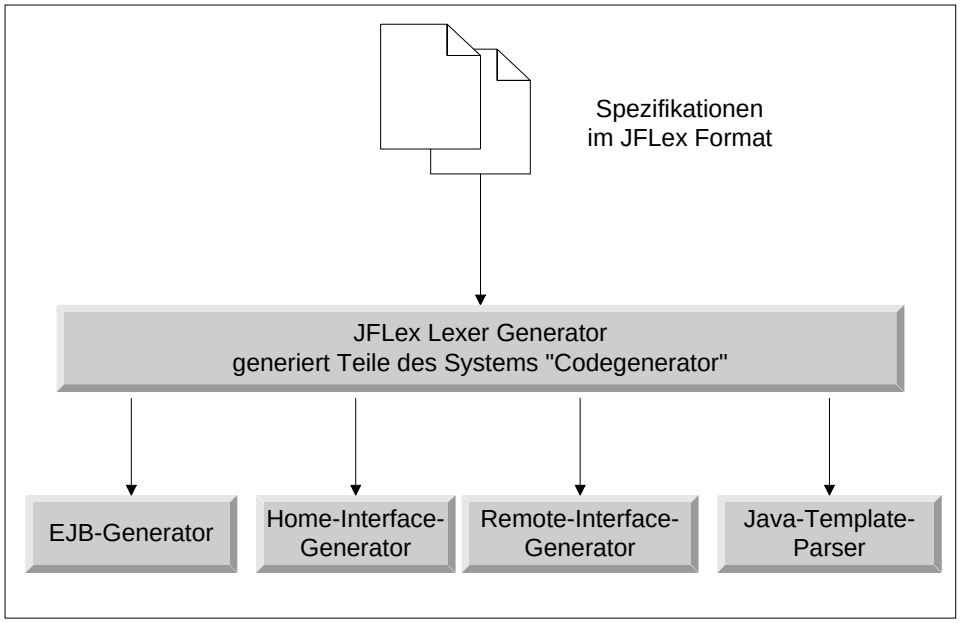

Abbildung 5.3: Automatische Programmierung von Teilen des Systems Codegenerator mit JFlex

```
3  Digit = [0−9]
4  Ident = {Alpha}|{Digit}|_
5  %%
6  <YYINITIAL> // Anfang
7  {
8          /* keywords */
9          "<RESOURCENAME>" { // wenn <RESOURCENAME> im Metamodell gefunden wird ...
10             yybegin(RESOURCENAME_State); // ... gehe zu State <RESOURCENAME_State>.
11         }
12           // ... weitere Schlüsselwörter
13 }
14 /* States */
15     <RESOURCENAME_State> // State <RESOURCENAME>
16     {
17         // Aufgabe:
18         // 1. Parsen des Ressourcenamen aus der Struktur
19         // des Metamodells <RESOURCENAME> eine_Ressource </RESOURCENAME>
20         // 2. Auswahl des Templates
```

```
21
22          {Ident}+                    // Teste ob Identifier OK
23          {
24              this.resourceName=yytext(); // hole Ressourcenname
25              // Festlegen des Dateinamens
26              String ejbTargetFileName = this.resourceName+"HomeInterface.java";
27              try
28              {
29                  // Oeffnen der Java Dateien zum Schreiben
30                  File myFile = new File(pathToEJBSource,ejbTargetFileName);
31                  ejbJavaFile = new PrintWriter(new BufferedWriter(new FileWriter(myFile )));
32                  generatedFiles.put( this.resourceName,myFile.getAbsolutePath ());
33              }
34              catch (java.io.IOException e){
35                  System.out. println ("IOError: "+e.getMessage());
36              }
37          }
38          "</RESOURCENAME>" { yybegin(YYINITIAL); } // gehe zurück zum Anfang
39      }
```

Interessant ist hierbei der Abschnitt von Zeile 6 bis 11. Er definiert die lexikalischen Regeln und darauf folgende Aktionen. Der Lexer beginnt mit der Aktion `<YYINITIAL>` (Zeile 6) seinen Analysevorgang. Beispielsweise soll der später generierte Lexer beim Parsen einer Datei den Namen der Ressource „UnixServer1“erkennen. Dieser ist im Metamodell mit der Zeichenkette `<RESOURCENAME> UnixServer1</RESOURCENAME>` definiert. Mit `<RESOURCENAME>` (Zeile 9) wird der Lexer veranlaßt, die Aktion <RESOURCENAME> abzuarbeiten (Zeilen 14 -34). Mit der Java Anweisung `this.resourceName=yytext();` (Zeile 18) wird der Text, der unmittelbar nach `"<RESOURCENAME>"` gefunden wird, der Java Variable `this.resourceName` zugewiesen, dies entspricht `this.resourceName= "UnixServer1"`. Jetzt folgen weitere Java Befehle. Aus dem ermittelten Ressourcennamen wird der Name der Java Quellcode Datei für das EJB Home-Interface gebildet (Zeile 20). Es folgen typische Java I/O Befehle zum Öffnen

einer Datei (Zeilen 21-32). Mit der Anweisung `generatedFiles.put(this.resourceName,myFile.getAbsolutePath())` (Zeile 26) ist eine Zuweisung eines Ressourcennamen zu einem Dateinamen implementiert. Die Anweisung `"</RESOURCENAME>"{yybegin(YYINITIAL)}` (Zeile 33) fordert den Lexer beim Antreffen der Zeichenkette `"</RESOURCENAME>"` auf, mit der Aktion `<YYINITIAL>` (Zeile 6) weiterzumachen, um nachfolgende Zeichenketten zu parsen.

5.4.3 Parsen des Metamodells am Beispiel des Home-Interface-Generators

Aus Gründen der Übersichtlichkeit wurde im obigen Listing nur auf eine spezielle lexikalische Regel einer *Regular Expression* des Home-Interfaces eingegangen. Der Home-Interface-Generator benötigt jedoch aus dem Modell Informationen über den Anfang und das Ende einer Ressource und über den Ressourcennamen. Einen Überblick über alle im Home-Interface-Generator zusammengefaßten *Regular Expressions* und ihrer Aktionen gibt folgendes Listing.

Listing 5.4: Definition der Aktion: Auswahl des Templates

```
1  <YYINITIAL> {
2          /* keywords */
3          "<RESOURCE>"
4          {
5                  if (! templateNamesLoaded)
6                          homeInterfaceTemplateNames = getTemplateNames();
7                  resourceName=null;
8                  yybegin(YYINITIAL); // Beginn
9          }
10         "</RESOURCE>" // Aufruf der Aktion für den Befehl "</RESOURCE>" des Metamodell
11         {
12                 // Auswahl des Templates " HomeInterface " und Schreiben
13                 doParseAndWriteToStream(pathToTemplates,
```

```
14                homeInterfaceTemplateNames.getProperty(”HomeInterface”));
15                 // Closing  JavaFileStream
16                 ejbJavaFile . close ();
17                yybegin(YYINITIAL); // Zurück zum Anfang
18            }
19 }
```

Von besonderem Interesse sind hierbei die Aktionen, die *Regular Expressions* für den Anfang (Zeile 3) und das Ende (Zeile 10) einer Ressource einleiten. So veranlaßt die Aktion beim einem Auftreten des Blockbefehls `"<RESOURCE>"` den fertigen Home-Interface-Generator, eine neue Ressource mit dem Löschen der Variablen einzuleiten (Zeile 7) und falls noch nicht erfolgt, die Konfigurationsdatei für den Generator auszulesen (Zeilen 5-7). Die Aktion für den ebenfalls im Metamodell vorkommenden Blockbefehl `"</RESOURCE>"` ruft die Methode `doParseAndWriteToStream((String)PathToTemplates, (String)TemplateName)` auf, die Text aus einem Template und den Namen der Ressource in die Java Quellcode-Datei für das Home-Interface schreibt. Dazu wird die Komponente `JavaTemplateParser` benötigt, auf die im Kapitel 5.4.5 näher eingegangen wird. Schließlich werden die I/O-Schnittstellen zu offenen Dateien geschlossen. Der Punkt . (Zeile 24) definiert die Aktion für Text, der mit keiner *Regular Expression* übereinstimmt. Auch die Generatoren für das EJB Remote-Interface und die EJB Bean-Implementierung wurde mit *JFlex* erstellt. Im Unterschied zum Home-Interface-Generator sind hier jedoch Aktionen für alle Schlüsselwörter und Blockbefehle des Metamodells implementiert.

5.4.4 Bestimmung der Java Schablone durch den Code Generator

Die Auswahl der Java Schablone ist in den Generatoren unterschiedlich implementiert. Die Auswahl ist an die Komplexität der Java Quelle gebunden. So existiert im Home-Interface-Generator nur eine Schablone für das einfach strukturierte Home-Interface. Im Remote-Interface-Generator und Entity-Bean-Generator werden die

Schablonen in Verbindung mit den Aktionen des Metamodells ausgewählt. So wählen Remote-Interface-Generator und Entity-Bean-Generator mit dem Blockbefehl für eine neue Ressource `<RESOURCE>` auch die Schablone für den Kopfbereich des Java Dokuments aus. Für den Blockbefehl `</RESOURCE>` wird der Fußbereich des Dokumentes ausgewählt. Der Blockbefehl `</PROPERTY>` wählt, nachdem der Datentyp durch `<PROPERTYTYPE>Datentyp</PROPERTYTYPE>` bekannt ist, das entsprechende Template aus.

5.4.5 Parsen der Java Schablone durch den Java-Template-Parser

Nachdem die Parameter aus dem Metamodell extrahiert (siehe auch Grafik 5.4) wurden und die entsprechende Java Schablone gewählt wurde, werden diese Angaben dem Java-Template-Parser übergeben. Diese Komponente hat die Aufgabe, die Java Schablone zu parsen und Schlüsselwörter in Quellcodepassagen mit den Parametern aus dem Metamodell zu überschreiben.

Listing 5.5: Template für das Entity-Home Interface

```
1 package dorm.server. resources ;
2 public  interface  <RESOURCENAME>HomeInterface extends javax.ejb.EJBHome
3 {
4     public <RESOURCENAME>RemoteInterface
5         findByPrimaryKey(dorm.generator. utils .DormPk primkey)
6         throws java.rmi.RemoteException, javax.ejb.FinderException;
7 }
```

Anstatt des Schlüsselwortes `RESOURCENAME` wird der Name der Ressource (hier im Beispiel UnixServer1) eingesetzt. Das fertige Home-Interface hat dann folgendes Aussehen:

Listing 5.6: Generiertes Entity-Home-Interface für die Ressource „UnixServer1“

```
1 package dorm.server. resources ;
2
```

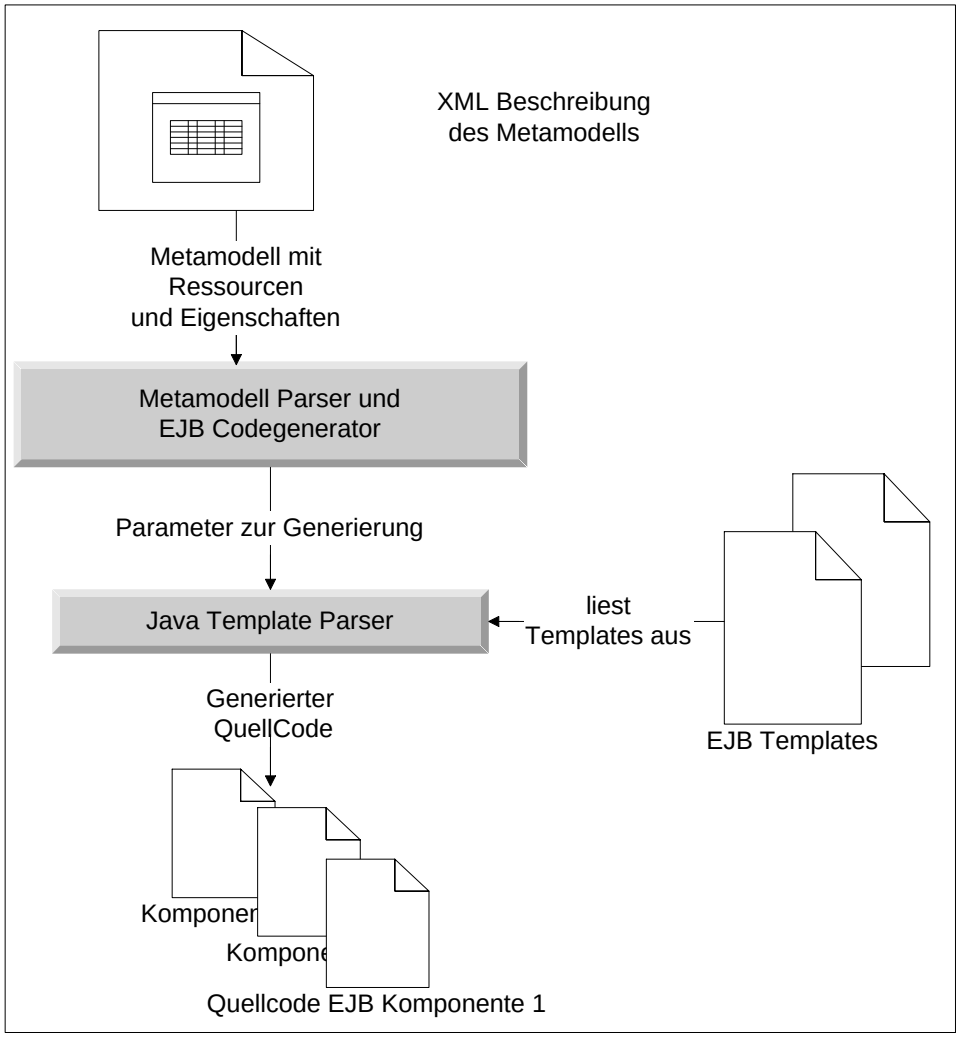

Abbildung 5.4: Die Codegeneratoren und ihre Schnittstellen

```
3 public interface UnixServer1HomeInterface extends javax.ejb.EJBHome
4 {
5     public UnixServer1RemoteInterface findByPrimaryKey(dorm.generator.utils.DormPk primkey)
6         throws java.rmi.RemoteException, javax.ejb.FinderException;
7 }
```

5.4.6 Schreiben der Java Schablone

Der Java-Template-Parser ist ebenfalls mit JFlex erstellt. Er wird in den Generatoren in der Methode `doParseAndWriteToStream((String) PathToTemplates,(String)TemplateName)` aufgerufen. Dabei dienen die Parameter zum Öffnen des Templates innerhalb des Dateisystems (Zeilen 3 -14). In einer neuen Instanz des `dorm.server.generator.logic.JavaTemplateParser` (Zeile 16) werden die zu ersetzenden Parameter des Metamodells gesetzt (Zeile 18). Nun werden solange Token vom Typ `dorm.server.generator.logic.Yytoken` gelesen, bis das Ende des Templates erreicht ist. Die gelesenen Token werden sofort in die vorher geöffnete Quellcodedatei geschrieben (Zeile 23). Innerhalb des `dorm.server.generator.logic.JavaTemplateParser` findet dabei die in Kapitel 5.4.5 geschilderte Ersetzung statt. Zum Abschluß werden die Datenströme zur Instanz des `dorm.server.generator.logic.JavaTemplateParser` geschlossen (Zeilen 28-34).

Listing 5.7: Datenstrom innerhalb des Generators

```
1 private void doParseAndWriteToStream(String pathToTemplates, String templateFileName)
2 {
3         try{
4             // öffnen der Templatedateien zum Lesen
5             ejbTemplateReader =
6                 new BufferedReader(
7                     new FileReader(
8                         new File(pathToTemplates,templateFileName)
```

```
9                       )
10                  );
11              }
12              catch ( java.io.IOException e){
13                  System.out. println ("IOError: "+e.getMessage());
14              }
15              // Neuer EJB Template Parser
16              ejbTemplateParser = new JavaTemplateParser(ejbTemplateReader);
17              // Setzen der gelesenen Variablen aus dem Metamodell
18              ejbTemplateParser.setResourceName(this.resourceName);
19              // Schreiben des Programmteils
20              try{
21                  Yytoken t;
22                  while (( t = ejbTemplateParser.yylex ()) != null)
23                      ejbJavaFile . print (t);
24              }
25              catch ( java.io.IOException e){
26                  System.out. println ("IOError: "+e.getMessage());
27              }
28              // Closing TemplateFileStream
29              try{
30                  ejbTemplateReader.close ();
31              }
32              catch ( java.io.IOException e){
33                  System.out. println ("IOError: "+e.getMessage());
34              }
35  }
```

5.5 Der Ressourcenmanager

Das im Kapitel 5.3 vorgestellte Metamodell dient als Grundlage für den Generierungsprozess. Das System Ressourcenmanager hat die Aufgabe, eine benutzerfreundliche Schnittstelle für die Definition dieses Modells bereitzustellen und den

Generierungsvorgang interaktiv anzustoßen und zu überwachen. Diese Aufgaben setzen eine komplexe Multitier-Architektur voraus (vergleiche Kapitel 5.1).

5.5.1 Session Beans als serverseitige Komponente

Die als zustandsorientierte -*statefull*- EJB-Session Bean implementierte Komponente stellt ein Repository an Methoden für die Definition der Elemente des Metamodells zur Verfügung. Konkret sind Methoden für das Einfügen (insert), Verändern (update), Löschen (delete) und Auflisten (list) von Klassen, Ressourcen, Eigenschaften und Optionen von einem entfernten Client aufrufbar. Darüberhinaus können Nutzer und Nutzergruppen aufgelistet werden, die Generierung des Metamodells und der Teile der Entity Beans sowie der Deploymentvorgag angestoßen werden.

5.5.2 Zugriff des Ressourcenmanagers auf die Datenbank

Über die im Kapitel 5.2 beschriebenen, die Datenbank kapselnden Java Beans erfolgt auch in der serverseitigen Komponente der Zugriff auf die Datenbank. Dabei konnte von der Wiederverwendung einiger bereits definierter Komponenten Gebrauch gemacht werden. Folgendes Listing beschreibt den Aufruf einer Java Bean für das Hinzufügen einer neuen Klasse.

Listing 5.8: Einfügen einer neuen Klasse im Ressourcenmanager

```
1 public void insertClass ( String className, String classLabel )
2     throws java.rmi.RemoteException, dorm.generator. utils .NestingException
3 {
4      InsertClass myInsertClass = new InsertClass ();
5     myInsertClass.setClassName(className);
6     myInsertClass. setClassLabel ( classLabel );
7     myInsertClass.executeUpdate ();
8
9      // ErrorHandling Database
```

```
10      if ( myInsertClass.getReturnCode()!=0)
11          throw new dorm.generator. utils .NestingException(
12          new dorm.server. database .DatabaseAccessException(myInsertClass.getReturnCode ()));
13  }
```

Nachdem eine neue Instanz der Java Bean `dorm.server.database.InsertClass` (Zeile 4) mit ihren Parameter erzeugt wurde (Zeilen 5-6), kann die unterliegende *Stored Procedure* ausgeführt werden (Zeile 7). Falls ein Fehler auftritt, wird dieser mit der *Exception* vom Typ `dorm.server.database.DatabaseAccessException` an den Client weitergereicht (Zeile 10-12).

5.5.3 Die Schnittstelle zu den Generatoren

Da vom Client die Generierung der Entity EJB Objekte angestoßen wird, muß die Serverkomponente des Ressourcenmanagers Zugriff auf die Generatoren haben. Das folgende Listing zeigt dies für den Home-Interface-Generator.

Listing 5.9: Schnittstelle für den Aufruf den Home-Interface-Generators im Ressourcen Manager (Ausschnitt)

```
1  public void generateEntityBean ()
2  throws java.rmi.RemoteException,dorm.generator. utils .NestingException {
3           dorm.server. generator . logic .EJBBeanGenerator beanGen = null;
4           String metaFile= null;
5           try {
6             java. util . Properties config = new java. util . Properties ();
7             config.load(new FileInputStream(new File( generatorConfFile )));
8             metaFile = config. getProperty ("MetaFile");
9             beanGen = new dorm.server. generator . logic .EJBBeanGenerator(new java.io. FileReader( metaFile ));
10            // Parameter
11            beanGen.setOutDirectory( config. getProperty ("SourceOutPutDir"));
12            beanGen.setPath2Templates( config. getProperty ("PathToTemplatesBean"));
13            beanGen. setTemplateFileList ( config. getProperty ("FileListName"));
```

```
14          }
15          catch (...);
16          try{
17              while (! beanGen. finished ) beanGen.yylex ();
18          }
19          catch (...)
20           ...
21  }
```

Zuerst werden die Parameter aus der Konfigurationsdatei ausgelesen (eine Auflistung möglicher Einstellungen findet sich in Kapitel 6.2). Da die Generatoren in Java programmiert sind, kann durch einfache Instantiierung ein neuer Home-Interface-Generator erzeugt werden (Zeile 9). Nun werden dem Generator die Verzeichnisparameter übergeben (Zeilen 11 -13). Der eigentliche Generierungsprozeß findet in der Lexermethode `.yylex()` (Zeile 17) statt (siehe auch [10] - Scanning method) . Eventuelle I/O Fehler werden hierbei abgefangen und an den Client weitergegeben.

5.5.4 Der Deploymentvorgang

Das *Deployment* umfaßt die Vorgänge

- Kompilieren der Quellen
- Dokumentation mit JavaDoc
- Erstellen des Deployment-Descriptors
- Packen der kompilierten Dateien und des Deployment-Descriptors in ein Jar Archive

Die Quellen werden mit dem *javac* Werkzeug [30] von *Sun Microsystems* [32] kompiliert. Der Aufruf ist im folgenden Listing beschrieben.

Listing 5.10: Aufruf des Java Compilers innerhalb eines Java Programms

```
1 sun. tools . javac . Main javacComp = new sun.tools. javac . Main(System.out,"Dorm␣JavaCompiler");
2  String [] cmd2CompilerHomeInterface = {
3                                  "-verbose",
4                                  "-d",
5                                  resourceDeployDir. getAbsolutePath (),
6                                   homeInterfaceFiles . getProperty (resourceName)
7  };
8  javacComp.compile(cmd2CompilerHomeInterface);
```

Es wurde der *JDK 1.1* verwendet, da die verwendete Middleware *Sybase Enterprise Application Server* in der Version 3.5 [35] auch nur Entity EJB Komponenten der Version 1.0 mit dieser Version des JDK akzeptiert.

5.5.5 Der Deployment-Manager als Verbindung zur Middleware

Die Komponente *Deployment Manager* `dorm.server.generator.logic.DeploymentManager` erstellt einen Descriptor nach der EJB Spezifikation 1.0 [26] von Sun.[2]Dazu liest er aus der Konfigurationsdatei die Parameter für das *Deployment* aus. Nun wird eine neue Instanz des Objektes `javax.ejb.deployment.EntityDescriptor` erstellt und die Parameter über die in [26] beschriebenen Methoden gesetzt. Dieses Objekt wird mit einem *Manifest*, das einen Verweis auf das eben erstellte Objekt enthält, in das Verzeichnis der Ressource geschrieben. Für das Deployment werden also pro Ressource folgende Dateien benötigt:

- Entity Home Interface
- Entity Remote Interface
- Entity Bean

[2]Der Autor erwartet, daß Sybase noch innerhalb der Erstellung dieser Arbeit seine Software auf den Stand der EJB Spezifikation 1.1 hebt. Deshalb kann die hier beschriebene Implementierung des Deployments von der tatsächlichen geringfügig abweichen.

- Manifest
- Instanz des *Deployment Descriptors*

Diese Dateien werden jetzt zu einem *Jar Archive* [28] gepackt. Dazu wird die Klasse `dorm.generator.utils.MyCompressor` benutzt, die mit den Bibliotheken [22] und [15] erstellt wurde. Die so entstandene Datei kann über die *Sybase Software Jaguar Manager* eingelesen werden (siehe auch Kapitel 6.3).

Zu einem vollständigem *Deployment* gehört sowohl das Einlesen der *Jar-Archive*, das Generieren und Kompilieren der *Stubs* und *Skeletons* und der notwendigen IDL Interfaces auf dem Server sowie ein Auffrischen des Server-Repository. Die verwendete Middleware *Sybase Enterprise Applikation Server* unterstützt dies jedoch nicht. Sybase antwortete auf eine Anfrage des Autors, daß diese Fähigkeit erst in einer späteren Version unterstützt wird. [3]

5.5.6 Der Client als Java Applet

Um den Client in kurzer Zeit implementieren zu können, wurde stark auf den *Graphical User Interface (GUI)* Builder der Entwicklungsumgebung *PowerJ* von Sybase [36] zurückgegriffen. Die hatte zu Folge, daß der Quelltext des Client von Menschen nur bedingt lesbar ist. Das *GUI* des Clients wurde fast ausschließlich mit Standard *Java - AWT* Komponenten implementiert. Aufgrund der komponentenorientierten Architektur kann auch ein anderes Interface mit beispielsweise *Java Swing* oder mit *Java Server Pages* Technologie leicht erstellt werden. Der Client implementiert folgende Funktionen:

- Verbinden zur EJB Session Bean `dorm.server.generator.Resourcemanager`

[3] Zitat der Mail vom 29. August 2000:

Command line deployment is not supported in this release. It is slated for a future release.

Dave Wolf Internet Applications Division

- Bearbeiten der Elemente des Metamodells unter Beibehaltung der Modellintegrität
- Darstellung von möglichen Transaktionsfehlern und des Verlaufs der Codegenerierung
- Beschränkung bestimmter Eingaben auf Java Identifier

5.5.7 Clientseitige Verbindung zur serverseitigen EJB-Komponente

Da die Firma Sybase, wie viele Hersteller von Middlewaresystemen, die Java Klassen zum Aufruf einer Verbindung und der Erzeugung eines Remote-Interfaces proprietär implementiert, wird an dieser Stelle kurz in einem Beispiel der Zugriff auf eine *Session Bean* Komponente beschrieben.

Listing 5.11: Verbindung zu einer Session Bean mit der Sybase Implementation der JNDI Methoden (Ausschnitt)

```
1  protected powersoft.powerj.jaguar. InitialContext Dorm
2      = new powersoft.powerj.jaguar. InitialContext ();
3  Dorm.create( "ResourceManagerForm.Dorm" );
4  Dorm.setUseJavaxNaming(true);
5  Dorm.setInitialCtxName("com.sybase.ejb. InitialContextFactory ");
6  Dorm.setUser (..);
7  Dorm.setPassword (...);
8  Dorm.setURL( "iiop://amor.rz.fhtw-berlin.de:8000" );
9  Dorm.connect();
10  ...
11 powersoft.powerj.jaguar. InitialContext jctxt_resourceManagerHome =
12     powersoft.powerj.jaguar. InitialContext .findByName("ResourceManagerForm.Dorm");
13  ...
14 protected dorm.server.generator.ResourceManagerHome resourceManagerHome =
15     (dorm.server.generator.ResourceManagerHome)
16         jctxt_resourceManagerHome.lookup("ResourceManager");
17  ...
```

```
18 dorm.server . generator . ResourceManagerHome _EJBHOME_TMP =
19     (dorm.server . generator . ResourceManagerHome)
20         jctxt_resourceManagerRemoteInterface . lookup("ResourceManager");
21 protected dorm.server . generator . ResourceManagerRemoteInterface
22     resourceManagerRemoteInterface = _EJBHOME_TMP.create();
23  ...
```

Die Klasse `powersoft.powerj.jaguar.InitialContext` stellt die Sybase eigene Variante der von Sun empfohlenen Klasse `javax.naming.InitialContext` des *Java Naming Interfaces* dar (Zeilen 1-2). Die selbe Klasse wird noch einmal bei der Erzeugung des Home-Interface der *Session Bean* benutzt (Zeilen 11-12). Das Remote-Interface kann dann mittels eines „lookups“ aus dem Home-Interface zugewiesen und (Zeilen 18-20) mit der Methode `.create()`. schließlich erzeugt werden (Zeilen 21-22). Weitere Beispiele für den Zugriff auf entfernte EJB Objekte findet der Leser unter [31].

5.5.8 Besonderheiten bei der Implementierung des Clients

Sämtliche unter 5.5 beschriebenen Funktionen sind über eine Schnittstelle vom Client aufrufbar. In einer `java.awt.TextArea` werden die Ergebnisse der Transaktionen und der Codegenerierung dargestellt. Bei der Eingabe der Klassen-, Ressourcen- und Eigenschaftsnamen werden nur Zeichen zugelassen, die *Java Identifier* für den Beginn eines Bezeichners sind, also Buchstaben, das Zeichen ’$’ und der Unterstrich ’_’. Folgendes Listing zeigt die Implementation dieser Funktionalität.

Listing 5.12: Filter für die Eingabe von Zeichen von der Tastatur

```
1 void checkIfValidKey( java . awt. event . KeyEvent e)
2 {
3     // erlaubt sind Buchstaben , ’ $ ’, ’ _’
4     if ( Character . isJavaIdentifierStart (e . getKeyChar()))
5         return;
```

```
6       else
7           // Den Event verwerfen
8           e.consume();
9   }
```

5.6 Entwicklung eines Prototypes für die Erstellung der Templates

Um konstante Teile einer Klasse zu generieren, kann der Generator diese mittels `print`-Befehlen erzeugen. Die Entwicklung des Generators wird dann allerdings sehr zeitaufwendig und der resultierende Code ist unübersichtlich und wartungsunfreundlich. Vorteilhafter ist es, die konstanten Teile der Zielsprache wie gewohnt in eine Datei zu schreiben und zusätzlich eine Generatorsprache in spezielle Markierungen einzubetten. Die eingebetteten Befehle steuern den Generator, konstanter Zielcode wird unverändert ausgegeben. Eine solche Datei wird Stanzform, Schablone oder auch Template genannt (siehe auch [18]). Für die Umsetzung der *Entity Bean* wurden folgende Templates erstellt:

- ein Home-Interface-Template
- Remote-Interface: Kopf- und Fußzeilen, sowie für jeden Datentyp des DORM-Modells ein Template mit einer Get- und Set- sowie eventuell List-Methode
- Entity -Bean-Implementation: Kopf- und Fußzeilen, sowie für jeden Datentyp des Dorm Modells ein Template mit einer Get- und Set- sowie eventuell List-Methode

Die Vorgehensweise zum Erstellen der Templates bestand in der Erstellung eines Prototypes einer Entity Bean, die mögliche Funktionen und dazu benötigten Methoden implementiert. Die folgenden Absätze beziehen sich vorwiegend auf die

Implementierung des Entity-Bean Rumpfes. Eine ausführliche Darstellung der Interfaces würde den Rahmen dieser Arbeit deutlich überschreiten. Ebenso wurde auf eine komplette Beschreibung aller Methoden verzichtet

5.6.1 Die Aktivierung einer Entity Bean im Prototyp

Eine wesentlicher Abschnitt im Lebenszyklus einer Entity Bean (siehe 14) ist die Aktivierung. Zur Aktivierung einer Entity Bean benötigt man einen eindeutigen Schlüssel um eine eindeutige Zuordnung zu einer Relation in der Datenbank zu gewährleisten.

Listing 5.13: Aktivierung der Entity Bean

```
1   /**
2    * Entity Context of this EJB.
3    * Set in ' setEntityContext ()' before any ' ejbCreate ()' or ejbFindXXX() is executed.
4    */
5   private javax.ejb.EntityContext _entityContext;
6   /**
7   * UserName
8   */
9   private String uid = null;
10   // method for interface javax.ejb.EntityBean
11  public void ejbActivate()
12      throws java.rmi.RemoteException{
13      uid = ((dorm.generator.utils.DormPk) this._entityContext.getPrimaryKey()).uid;
14      writeMsg("Ejb Activated with PK: "+uid+" !");
15  }
16   // findermethod
17  public dorm.generator.utils.DormPk ejbFindByPrimaryKey(dorm.generator.utils.DormPk primkey)
18      throws java.rmi.RemoteException, javax.ejb.FinderException{
19      int returnCode= invalid;
20      ListAccounts myListAccounts = new ListAccounts();
21      try{
22          if (myListAccounts.findByResourceAndUid(primkey.uid,resourceName))
```

```
23                  return primkey;
24              else
25                  throw new javax.ejb.ObjectNotFoundException("Object for PK: " +primkey.uid+" not found");
26          }
27          catch (Exception ex){
28              throw new java.rmi.RemoteException("ejbFindByPrimaryKey: "+ex.getMessage());
29          }
30          finally {
31                  primkey=null;
32          }
33      }
```

Dazu wird zuerst über das Home-Interface eine *Finder Methode* in der Entity Bean aufgerufen (Zeilen 17 -33). Mit dem Aufruf der Java Bean `dorm.server.database.ListAccounts` wird festgestellt, ob ein Nutzer zu der aktuellen Ressource gehört. Ist dies nicht der Fall, wird eine `javax.ejb.FinderException` an den Entity-Bean-Container der Middleware weitergeleitet, die darauf den Aktivierungsvorgang der Entity Bean abbricht. Wird die *Finder Methode* mit positivem Resultat beendet, ruft der Container der Entity Bean die Methode `ejbActivate()` auf. Die im Kontext der Entity Bean gespeicherte Nutzeridentifikation-*UserID*-wird einer lokalen Variable in der Entity Bean zugewiesen (Zeile 13) und steht damit für weitere Datenbankabfragen innerhalb der Entity Bean zur Verfügung.

5.6.2 Transformation der Datentypen des Modells auf Java Datentypen

Eine Hauptaufgabe des Prototypen der Entity Bean ist die Implementierung generischer Methoden für jeden im Modell definierten Datentyp. Das Metamodell muß sich in eine Java Instanz vollständig transferieren lassen. Dazu ist eine vollständige Beziehung zwischen einem Datentyp des Modells und einem Java Datentyp herzustellen und geeignete Methoden des Zugriffs auf die Daten zu implementieren. Tabelle 5.3 zeigt diese Beziehung für alle implementierten Datentypen. Eine Be-

Dorm Modell Datentyp	Java Datentyp
String	java.lang.String
boolean	boolean
date	java.util.Date
int	int
float	double
list	dorm.generator.utils.StringList
option	java.lang.String
vector	dorm.generator.utils.StringList

Tabelle 5.3: Implementierte Java-Datentypen des Metamodells

sonderheit stellt der Datentyp `date` dar. Der Java Datentyp `java.util.Date` muß hierbei das in der Datenbank gespeicherte Datumsformat verstehen können. Im folgenden Listing übernimmt hierzu eine Instanz von `java.text.DateFormat` diese Aufgabe (Zeile 3).

Listing 5.14: Parsen des Datumsformates der Datenbank

```
1 java.util.Date returnValue = new java.util.Date();
2 String tmpValue=getDatabaseValue();
3 java.text.DateFormat df =
4     java.text.DateFormat.getDateInstance(java.text.DateFormat.SHORT,java.util.Locale.GERMAN);
5 returnValue = df.parse(tmpValue);
```

Die Parameter der Instanz (`java.text.DateFormat.SHORT` und `java.util.Locale.GERMAN`) beschreiben hierbei das in der Datenbank definierte Format (Zeile 4). Der in der Datenbank als `float` bekannte Typ wird aufgrund des identischen Wertebereiches in Java als `double` definiert. `dorm.generator.utils.StringList` repräsentiert eine verkettete Liste von Elementen des Types `java.lang.String` und ist mit dem Datentypen `java.util.Vector` vergleichbar. Sinnvoll wurde die Einführung dieses Datentypes, da die Zielplattform JDK 1.1 nicht den Datentyp `java.util.Collection` anbietet. Weiterhin ist das Ergebnis der Abfragen oft eine Liste von Zeichenketten. Der Datentyp `java.util.Vector`

bietet hier zwar eine Lösung, `dorm.generator.utils.StringList` vermeidet jedoch die lästige Konvertierung von `java.lang.Object` zu `java.lang.String`.

5.6.3 Template einer Methode für den Datentyp String

Der folgende Abschnitt zeigt die Implementation einer *Get- Methode* für den Datentyp String. Die Methode ist Teil des entwickelten Prototypes (siehe Kapitel 3.4.7) aus dem die Templates für die Generatoren extrahiert worden.

Listing 5.15: Template einer Get-Methode innerhalb der Entity Bean für den Datentyp String

```
1  public String get<PROPERTYNAME>()                                    // Dynamisch
2      throws java.rmi.RemoteException,dorm.generator.utils.NestingException {
3       //   Initalise
4      int returnCode= invalid ;
5      String propertyName = "<PROPERTYNAME>";                          // Dynamisch
6      String returnValue =null;
7       // Execute Stored Proc
8       ListSetting myListSetting = new ListSetting ();
9      myListSetting.setUserName(uid);
10     myListSetting.setResourceName(this.resourceName);
11     myListSetting.setPropertyName(propertyName);
12     returnValue = ( String ) myListSetting.getSetting ();
13      //  ErrorHandling Database
14     if ( myListSetting.getReturnCode()!=0)
15          throw new dorm.generator.utils.NestingException(
16          new dorm.server.database.DatabaseAccessException(myListSetting.getReturnCode ()));
17     return returnValue ;
18 }
```

Die Implementation der Methode umfaßt

- einen Initialisierungsteil (Zeilen 3-6), hier werden Parameter zum Aufruf der die Datenbank kapselnden Java Beans (siehe Abschnitt 5.2) gesetzt und Ergebnisvariablen definiert, bzw. initialisiert
- einen Ausführungsteil, dabei wird eine neue Instanz der die entsprechende *Stored Procedure* abbildenden Java Bean erzeugt. Die Java Bean wird dann mit ihren Parametern aufgerufen und ausgeführt (Zeilen 7-12).
- den Fehlerbehandlungsteil (Zeilen 15 -16). Tritt ein Fehler während des Datenbankzugriffs auf, wird die zugehörige Fehlernummer in der Klasse `dorm.server.database.DatabaseAccessException` zu einer eine menschlich lesbaren Fehlermeldung zugeordnet und über die Klasse `dorm.generator.utils.NestingException` zum Client übertragen.

Circa 50 weitere Templates für Methoden zum Löschen, Anlegen, Listen, Setzen und Holen von Attributen eines Nutzers wurden im Rahmen der Arbeit nach diesem Schema implementiert. Die entstandenen Implementierungen variieren dabei in den Datentypen, den verwendeten Java Beans sowie in der Behandlung möglicher Ergebnisse.

5.6.4 Beispieldurchlauf der Generierung einer Methode für den Datentyp String

Das Listing des letzten Abschnittes soll als Beispiel für die Generierung einer Methode zur Ermittlung des Nachnamen eines Nutzers an einer bestimmten Ressource dienen. Die Eigenschaft Nutzernamen wird in der Datenbank für die Ressource `UnixSever1` mit `FullName` bezeichnet. (siehe Grafik 5.5). Aus diese Werten wird durch die Komponente `Meta-Modell-Generator` folgendes Metamodell erstellt:

```
<RESOURCE>
    <RESOURCENAME>UnixServer1</RESOURCENAME>
```

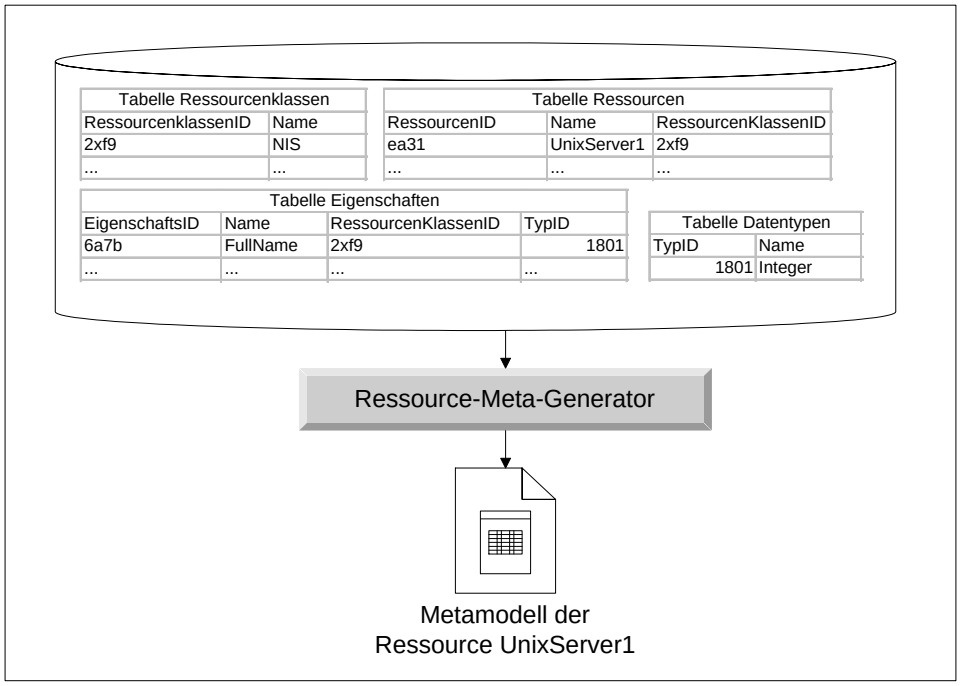

Abbildung 5.5: Beispieldurchlauf für die Ressource UnixServer1

```
    <SUPERCLASSNAME>NIS</SUPERCLASSNAME>
    <PROPERTY>
        <PROPERTYNAME>FullName</PROPERTYNAME>
        <PROPERTYLABEL>Nachname eines  Nutzers</PROPERTYLABEL>
        <PROPERTYTYPE>String</PROPERTYTYPE>
    </PROPERTY>
</RESOURCE>
```

Dieses Metamodell wird von den Codegeneratoren `Home-Interface-Generator`, `Remote-Interface-Generator` und `EJB-Generator` eingelesen. Diese werten das Metamodell aus und bestimmen das im letzten Abschnitt vorgestellte Template für die Generierung aus (Grafik 5.6). Weiterhin übergeben sie der Komponente `Java-Template-Parser` die Parameter der aus dem Metamodell entnommenen Eigenschaft. Diese Kom-

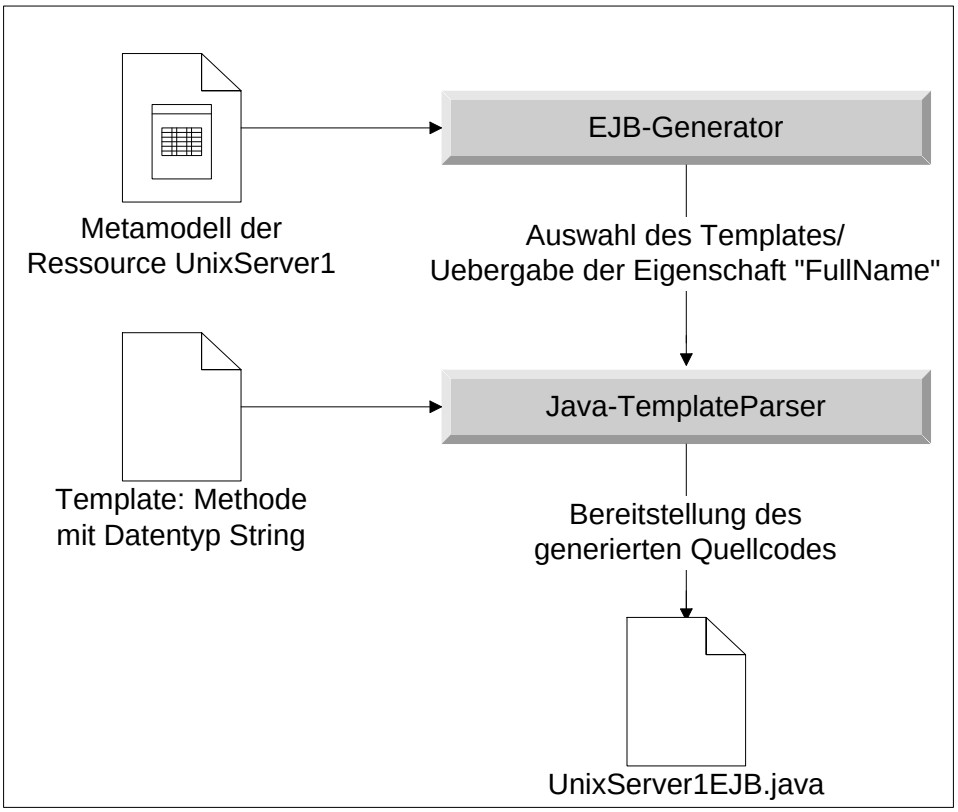

Abbildung 5.6: Auswahl des Templates durch den Generator

ponente analysiert das Template nach Anweisungen wie `<PROPERTYNAME>` und ergänzt es durch die Angaben aus dem Metamodell. In der Grafik 5.6 wird die Auswahl eines Templates für den EJB-Rumpf durch die Komponente `EJB-Generator` gezeigt. Aus Gründen der Übersichtlichkeit wurde auf die Darstellung der Auswahl von Templates durch den Remote-Interface-Generator und den Home-Interface-Generator verzichtet.

Nach dem die Komponente `Java-Template-Parser` den fertigen Quellcode generiert hat, wird dieser an den Deployment-Manager weitergegeben. Für das Deployment wird neben dem Quellcode für den EJB-Rumpf (in der Grafik mit `UnixServer1EJB.java` bezeichnet), auch Quellcode für das EJB-HomeInterface (`UnixServer1HomeInterface.java`) und

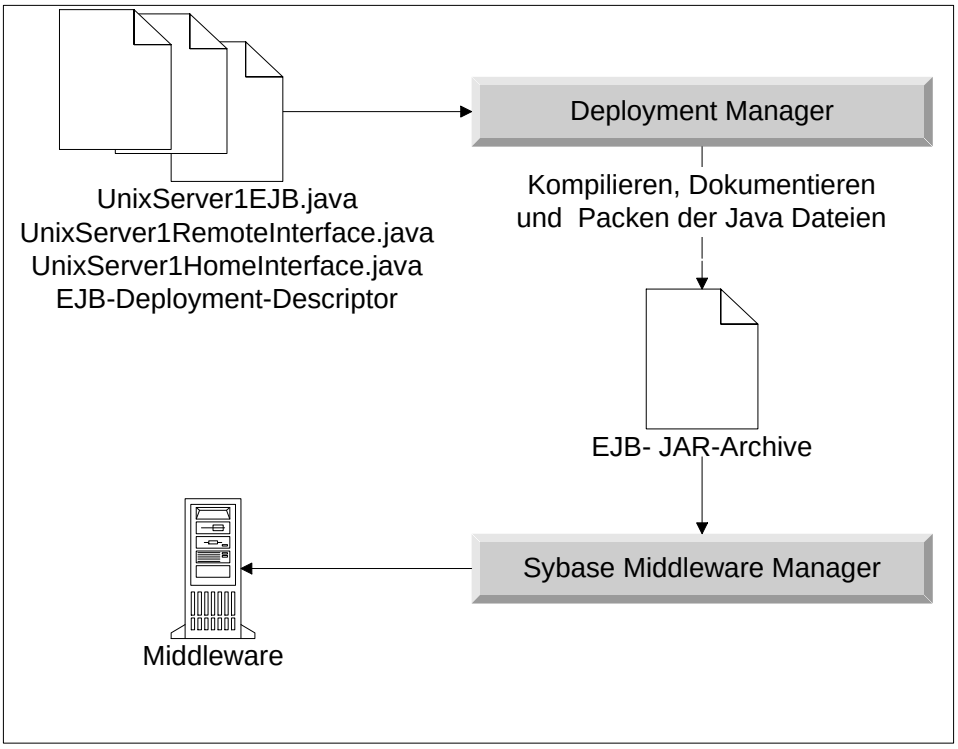

Abbildung 5.7: Deployment der Ressource UnixServer1 an die Middleware

ein EJB-RemoteInterface (`UnixServer1RemoteInterface.java`) sowie ein Deployment-Descriptor benötigt. Die Java Dateien werden durch das Werkzeug *JavaDoc* dokumentiert und mit dem Java-Compiler zu Bytecode compiliert. Aus den compilierten Java -Klassen wird zusammen mit dem Deployment-Descriptor ein komprimiertes Archiv erstellt. Dieses Archive entspricht der EJB Spezifikation 1.0. Es kann von jeder Middleware die diese Sepzifikation unterstützt eingelesen werden. Für die in dieser Arbeit verwendete Middleware *Sybase Enterprise Application Server 3.5* erfolgt dieser Schritt durch das von Sybase gelieferte Werkzeug *Sybase Jaguar Manager*.

Kapitel 6

Beispielanwendung

Die in den vorangegangenen Abschnitten vorgestellten Leistungsmerkmale des Systems zur Generierung einer adaptiven Informationsschicht erlauben somit die Erzeugung von Java Objekten zur Laufzeit aus einem dynamischen in einer Datenbank gehaltenen Modell. Um die Leistungsfähigkeit des Systems zu verdeutlichen, soll dies im Folgenden am Beispiel der Definition einer neuen Klasse mit ihren Eigenschaften und einer Ressource demonstriert werden. Aus dem dann definierten Modell wird der Generierungsprozeß angestoßen. Schließlich werden die Komponenten in einem Applikationsserver verteilt.

6.1 Modellierung des Modells

Um das Modell zu definieren, wird der Client des Systems *Ressourcenmanager* als Java Applet in einem WWW-Browser [1] geladen. Nun kann der Verbindungsaufbau zur serverseitigen Komponente des *Ressourcenmanager* erfolgen. Nach erfolgtem Verbindungsaufbau und dem Ladevorgang des bisherigen Modells in den Client

[1] Getestet wurden unter anderem Netscape Browser ab Version 4.5, bzw. 4.05 unter IRIX, Solaris und WINDOWS NT, sowie Internet Explorer ab Version 5 unter Windows NT und ab Version 4.5 unter MACINTOSH

kann jetzt die Klasse `Test` definiert werden. Dazu werden der Name der Klasse und eine Beschreibung in die entsprechenden Felder eingetragen (siehe Abbildung6.1). Nach dem Bestätigen der Eingabe über den `InsertButton` können der neuen

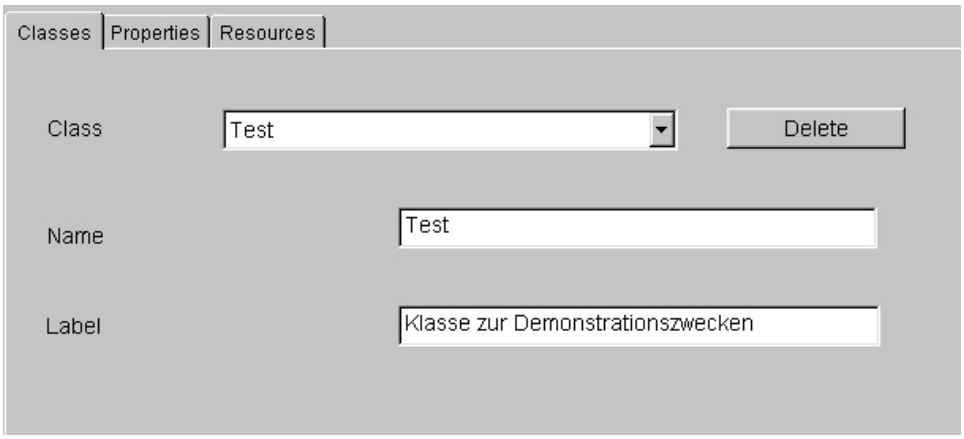

Abbildung 6.1: Modellierung der Ressourcenklasse *Test*

Klasse Eigenschaften zugewiesen werden. Dazu wird in die Ansicht `Properties` gewechselt und die eben erstellte Klasse `Test` in der Auswahl `Class` selektiert. Das System hat zu der eben erstellten Klasse eine Reihe von Standardeigenschaften erzeugt. Wir fügen die Eigenschaft `FullName` hinzu (siehe Abbildung 6.2). Sie soll den vollen Nutzernamen beinhalten. Über die Auswahl `Property` können diese verändert bzw. neue Eigenschaften hinzugefügt werden. Dabei werden die Attribute einer bereits vorhandenen Eigenschaft automatisch in die dafür vorgesehenen Datenfelder geschrieben. Dabei muß ein Datentyp ausgewählt werden. Für die Datentypen `Option` und `List` können über den Button `configure` Standardwerte zugewiesen werden. Angaben über das verbindliche Setzen eines Attributes erfolgen durch die Auswahl `Mandatory`. Bei Auswahl `with_Reference` oder `with_Existing_Reference` muß eine (existierende) Referenzeigenschaft angegeben werden. Die Eindeutigkeit wird mit der Auswahl `Constraints` bestimmt. Weiterhin kann zu jeder Eigenschaft eine Größe angegeben werden. Die Übernahme der Attributsangaben erfolgt mit `Insert`.

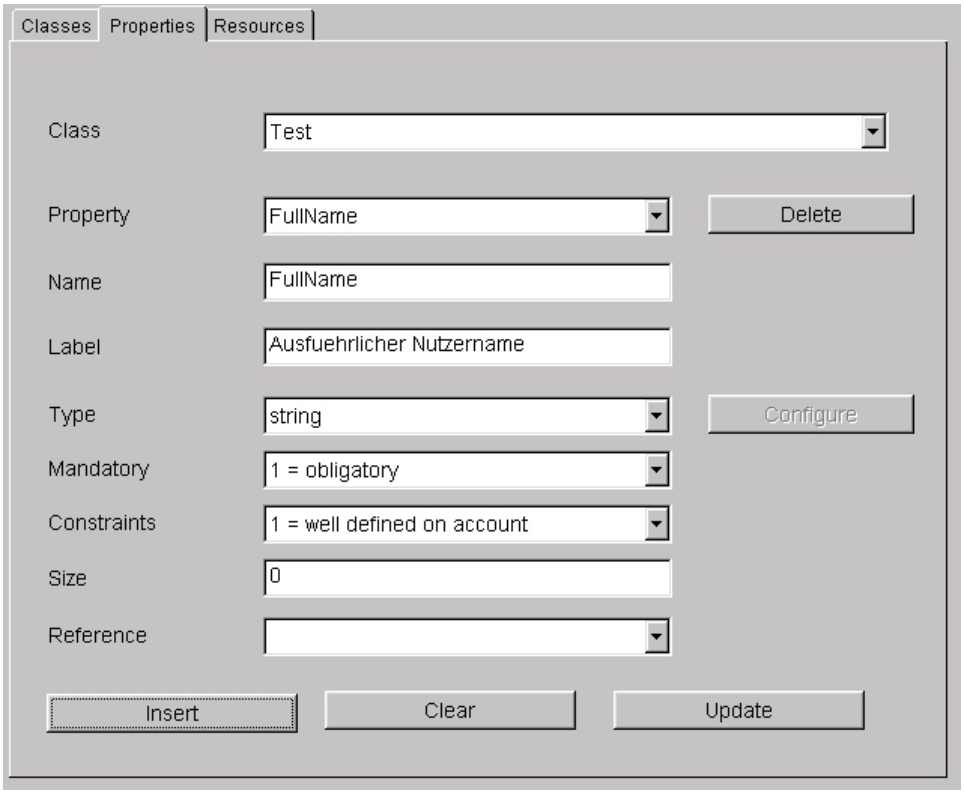

Abbildung 6.2: Modellierung der Eigenschaft *FullName*

Die so modellierte Klasse kann nun für die Instanziierung einer neuen Ressource dienen. In der Ressourcenansicht (siehe Abbildung 6.3) werden die Attribute der neuen Ressource bestimmt. Dazu wird ein neuer Name der Ressource sowie eine erklärende Bezeichnung eingegeben. Der Standardbenutzer wird über das Feld `DefaultUser` definiert. Der Eigentümer der Ressource wird durch `RessourceUser` festgelegt. Weiterhin kann die Gruppe der Ressource `GroupName` und die selten benötigte Elternressource `ParentName` angegeben werden. Wird die Ressource von einer Metaklasse abgeleitet (siehe 5.3.3), so muß diese unter `MetaName` definiert werden. Schließlich kann die neue Ressource durch `Insert` eingefügt werden.

Der Manager bietet darüber hinaus analoge Funktionen zum Ändern und

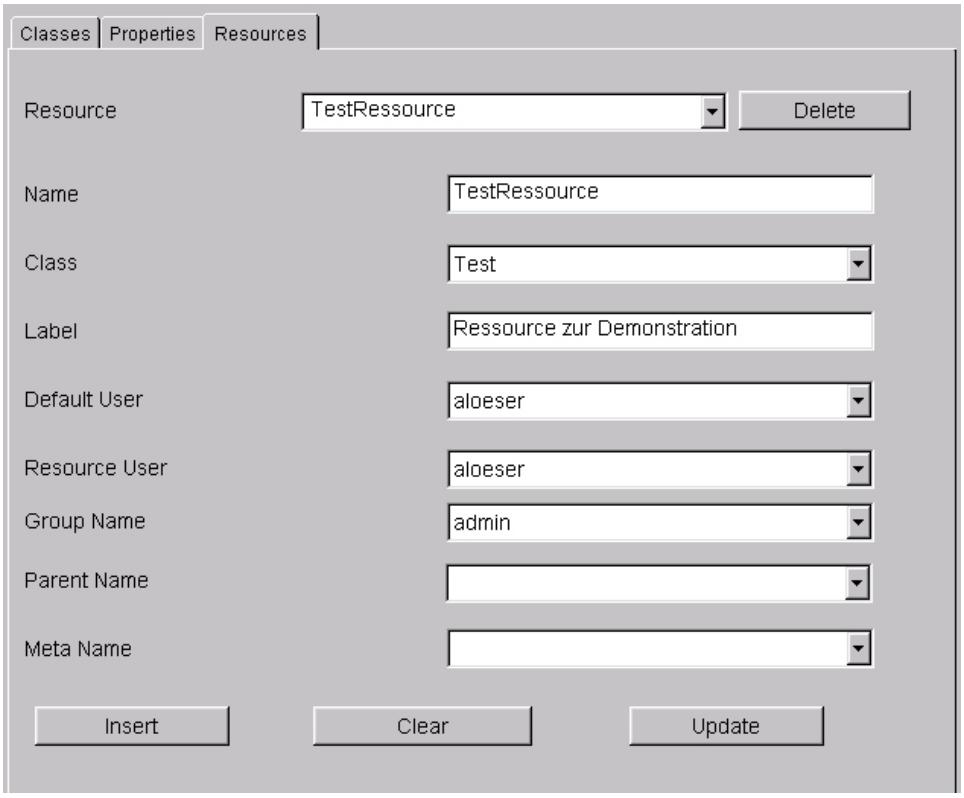

Abbildung 6.3: Modellierung der Ressource *TestRessource*

Löschen von Ressourcen, Attributen und Klassen. Dabei sind eventuell bestehende Abhängigkeiten zwischen Klassen und Ressourcen zu beachten.

6.2 Konfiguration des Generators und Generierung

Die Generierung wird zum größten Teil nicht sichtbar auf dem Server abgewickelt. Sie wird durch den Client mit dem Button `Generate` ausgelöst. Der Generierungsvorgang kann auf dem Client verfolgt werden. Er nimmt einige Minuten in Anspruch.

Zuvor muß jedoch die Generierung konfiguriert werden. Die Konfiguration wird über die Datei `$InstallPath$/DormGenerator/conf/DormGenerator.`

`conf` im Installationsverzeichnis der Installation des Generators auf dem Server definiert. Das folgende Listing zeigt eine Beispielkonfiguration.

Listing 6.1: Beispielkonfiguration Dorm-Generator

```
1 MetaFile    = DormGenerator/MetaModell/modell.txt
2 SourceOutPutDir = DormGenerator/ejbsource/dorm/server/ resources /
3 PathToTemplatesHomeInterface = DormGenerator/Templates/Java/EJBHomeInterface/
4 PathToTemplatesRemoteInterface  =  DormGenerator/Templates/Java/EJBRemoteInterface/
5 PathToTemplatesBean = DormGenerator/Templates/Java/EJBBean/
6 FileListName =      filelist   . dir
7 DeploymentDir   = / DormGenerator/deployment/
8 PathToTemplateDescriptor  = / DormGenerator/Templates/Descriptor /
9 TempClassPathDir = / net / pub/apps/ eas / JaguarCTS35/java/ classes /
```

Generell müssen die als Parameter definierten Verzeichnisse und Dateien Lese- und Schreibrechte für den Applikationsserver aufweisen. Nach einer gelungenen

Konfigurationsparameter	Bedeutung
MetaFile	Pfadangabe der Textdatei des Metamodells
SourceOutPutDir	Pfadangabe der generierten Java Quellcode Dateien
PathToTemplatesHomeInterface	Pfadangabe der Schablonen des Home-Interfaces
PathToTemplatesRemoteInterface	Pfadangabe der Schablonen des Remote-Interfaces
PathToTemplatesBean	Pfadangabe der Schablonen der Entity Bean Implementation
FileListName	Name der Datei, die eine Abbildung der Datentypen auf Dateinamen enthält
DeploymentDir	Verzeichnis, in dem die fertigen *Jar* Archive abgelegt werden
PathToTemplateDescriptor	Pfadangabe des *Standard Descriptors*
TempClassPathDir	Temporäres im *Java Classpath* liegendes Verzeichnis

Tabelle 6.1: Konfigurationsparameter des Generators und ihre Bedeutung

Generierung stehen im *Deployment*-Verzeichnis die generierten, kompilierten und mit einem *Deploymentdescriptor* versehenen *Jar* - Archive. Im folgenden Beispiel sind die Ressourcen `CustomerUser,DMA` und `LoginPage` erzeugt worden.

```
...
-rw-r--r--   1    7576 Sep 25 12:02 CustomerUserEntityBeanAll.jar
-rw-r--r--   1    7460 Sep 25 12:02 DMAEntityBeanAll.jar
-rw-r--r--   1    7545 Sep 25 12:02 LoginPageEntityBeanAll.jar
...
```

6.3 Das Deployment der Ressourcen zum Applikationsserver

Um die generierten Ressourcen in den Applikationsserver zu integrieren, ist ein Deploymentvorgang notwendig. Dieser Vorgang erfolgt momentan noch manuell und umfaßt das Einlesen der *Jar-Archive*, das Generieren und Kompilieren der *Stubs* und *Skeletons* und der notwendigen IDL Interfaces auf dem Server sowie ein Auffrischen des Server-Repository. Die verwendete Middleware, *Sybase Enterprise Application Server*, unterstützt eine Automatisierung dieses Vorgangs jedoch noch nicht. Sybase antwortete auf eine Anfrage des Autors, daß diese Fähigkeit erst in einer späteren Version unterstützt wird. [2]

Der hier verwendete Applikationsserver bietet das Werkzeug *Jaguar Manager* zum manuellen Deployment an. Innerhalb des Werkzeuges wird zu dem gewünschten Server eine Verbindung aufgebaut. Nun kann ein neues Package importiert werden (siehe Abbildung 6.4). Dazu wird die komplette Pfadangabe des *EJB Jar Archives* sowie der interne Packagename des Applikation Server benötigt. Die Entity Bean Informationen werden vom Server ausgelesen und können dann noch manuell angepaßt werden. Für die so importierte Entity Bean müssen nun die *Stubs* und *Skeletons* generiert werden. Nachdem dieser Schritt abgeschlossen ist, werden die generierten Java Dateien der *Stubs* und *Skeletons* mit dem *javac* Compiler

[2] Zitat der Mail vom 29 August 2000:

Command line deployment is not supported in this release. It is slated for a future release.

Dave Wolf Internet Applications Division

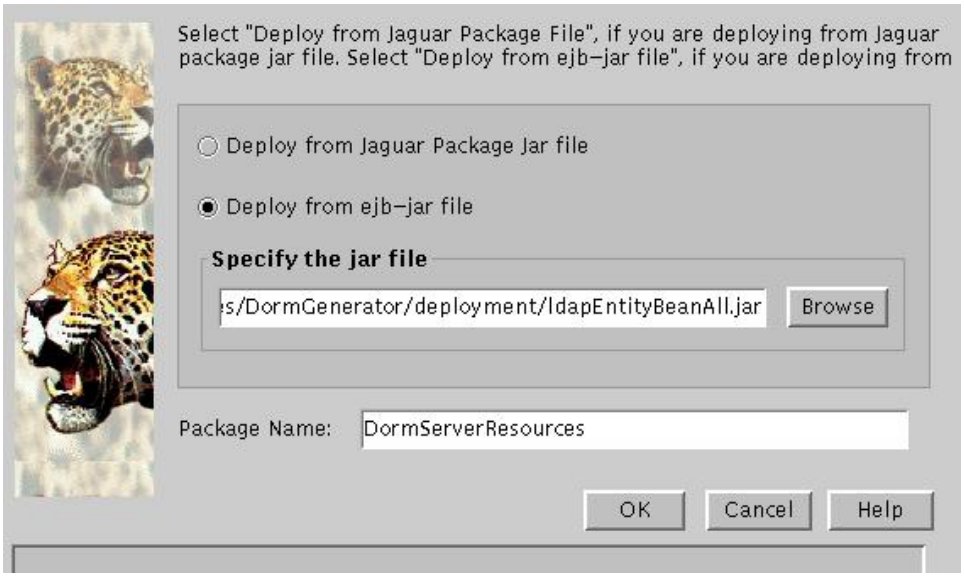

Abbildung 6.4: Einfügen eines neuen Packages

kompiliert. Nun kann der Server aufgefordert werden, sein *Repository* neu einzulesen. Die generierten Ressourcen stehen jetzt einem geeigneten Client zur Abfrage bereit.

6.4 Testen der Ressource

Nachdem eine Ressource modelliert und eine Entity Bean aus diesem Modell generiert wurde, kann auf die Daten der Ressource von einem entfernten Client zugegriffen werden. Dazu simulierten Clients Zugriffe auf die vorhandene Ressource für die Datentypen `String`,`Date`,`Float`,`Integer`,`List`,`Vector`,`Option` und `Boolean` für `Get`,`Set` und `List`- Zugriffe. Das folgende Listing zeigt eine Testimplementation.

Listing 6.2: Beispiel eines Get-Zugriffs für den Datentyp String

```
1 public class TestClient {
2     public static void main(String [] args){
3         // Context erstellen
```

```
4            ...
5            myJaguarContext.connect();
6            ...
7            //  Ressourcennutzer  festlegen
8            dorm.generator.utils.DormPk pk = new dorm.generator.utils.DormPk();
9            pk.uid = args[0];
10           // Remote Objekt der Ressource "Test" instanziieren
11           try {
12               myObjectEntityHome = (TestHomeInterface)myJaguarContext.lookup("TestEntity");
13               myObjectEntityRemote = (TestRemoteInterface)myObjectEntityHome.findByPrimaryKey(pk);
14           } catch( java.lang.Exception _e ) {..}
15
16           // Ausgabe des Ergebnisses
17           System.out.println(myObjectEntityRemote.getFullName());
18       }
19   }
```

Die Testimplementation instanziiert eine EJB Ressource vom `Test` (Zeilen 10 14). Für einen zur Laufzeit festzulegenden Nutzer der Ressource wird das Attribut `Fullname` aufgerufen. Keiner der erfolgten Tests führte zu einem fehlerhaften Ergebnis.

Kapitel 7

Zusammenfassung und Ausblick

In Zukunft werden Softwarewerkzeuge benötigt, die es einem Fachmann - auch ohne Programmierkenntnissse - erlauben, nur durch die Modellierung seiner Idee mit einem geeignetem Werkzeug, eine fehlerfreie und unmittelbar benutzbare Anwendung ohne Entwicklungs- und Codieraufwand zu erzeugen.

In dieser Arbeit wurde ein System vorgestellt, das bereits jetzt Teile von Anwendungen für das Ressourcenmanagement automatisch generieren kann. Dazu werden die Strukturen der Ressourcen über ein WWW-basiertes Werkzeug modelliert und in einem RDBMS gespeichert. Ein Codegenerator erzeugt aus diesen Strukturen Enterprise Java Beans. Diese stehen unmittelbar WWW-basierten Anwendungen für das Ressourcenmanagement zur Verfügung. Dabei werden das Transaktionsverhalten, der Datenbankzugriff und der Zugriff auf die Ressourcen und ihre Eigenschaften pro Nutzeridentität von den EJB-Komponenten gekapselt. Der Quellcode dieser Java-Komponenten ist dabei in frei editierbaren Templates definiert. Eine XML-konforme Schnittstelle bietet weiterhin auch externen Systemen die Möglichkeit, die Strukturen der Ressourcen und ihrer Eigenschaften auszulesen.

Momentan wird nur ein Teil der Applikationen für das Ressourcenmanagement automatisch generiert. Interessant wäre eine vollständig automatisierte Er-

stellung dieser Applikationen. Dazu könnte man die in dieser Arbeit entwickelten Technologien ebenfalls bei der Erstellung von adaptiven Clients bzw. Fachlogikkomponenten nutzen. Ein erster Schritt wäre der Entwurf von neuen Templates. Aus diesen Templates könnten dann durch den Codegenerator für die bestehende Softwarearchitektur weitere Applikationsbausteine generiert werden. Ebenfalls ist vorstellbar, daß der Generator ein Modell einer externen Software, beispielsweise *Rational Rose*, als Grundlage für die Erstellung von Komponenten benutzen könnte.

Anhang A

Abkürzungsverzeichnis

AWT	Abstract Window Toolkit
CORBA	Common Object Request Broker Architecture
DBMS	Database Management System
DTD	Document Type Definition
EJB	Enterprise Java Bean
ERM	Entity Relationship Modell
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity Package
JSP	Java Server Page
OOA	object-oriented Analysis
OOD	object-oriented Design
ORB	Object Request Broker
RDBMS	Relational Database Management System
WWW	World Wide Web
XML	Extensible Markup Language

Anhang B

Entwicklungsumgebung

Bei der Entwicklung der Plattform wurde folgende Hardware verwendet:

- Sun Ultra Enterprise 450
  2 CPU 250Mhz, 256 Mb RAM
- Standard PC
  1 CPU AMD Athlon 500 Mhz, 128 Mb RAM

Folgende Software wurde benutzt:

- Betriebssysteme:
  Sun Solaris 7
  Windows NT 4.0
- Java Software Development Kit:
  JavaTM 2 Plattform, Enterprise Edition Edition (J2EE)
  JavaTM 1.1.7 Plattform
- Verwendete Middleware:
  Sybase Enterprise Application Server 3.5
  Jaguar CTS 3.5

- Entwicklungsumgebung:

  Sybase Powerj 7

  Sybase Power Designer 6.1

- Datenbanksystem:

  Sybase Adaptive Server Enterprise 11.9.2

- WWW-Browser:

  Netscape Communikator 4.6

  Internet Explorer 5.0

- Textsatzsystem für diese Arbeit:

  LaTeX 2ε

Anhang C

Anlagen

Die beiliegende CD-Rom hat den folgenden Inhalt:

Verzeichnis java In diesem Verzeichnis liegen die Java Quellen der implementierten Programme.

Verzeichnis PowerJ 3.6.1 Workspace Eine Projektdatei für den Einsatz in der Entwicklungsumgebung Sybase PowerJ

Verzeichnis templates benötige Schablonen für die Generierung

Verzeichnis JFlex Der JFlex Generator und die in der Arbeit beschriebenen „Flex“ Dateien

Verzeichnis Beispielconfig eine Beispielkonfiguration

Verzeichnis Literatur Interessante Beiträge zum Thema

Anhang D

Selbständigkeitserklärung

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

Literaturverzeichnis

[1] Arentzen Ute ,Lörcher Ulrike. *Gabler Wirtschaftslexikon*. ISBN 3-4093-0387-1. Th. Gabler Verlag, Wiesbaden, 1997.

[2] Avantis GmbH. Avantis Homepage. `http://www.avantis.de`, 2000.

[3] Bannert Gabriele, Weitzel Martin. *Objektorientierter Softwareentwurf mit UML*. ISBN 3-8273-1487-9. Addison Wesley Longman Verlag Gmbh, München, 1999.

[4] Barocca Leonor,Hall John, Hall Patrick. *Software Architectures*. Springer Verlag, London, first edition, 2000.

[5] Feustel Björn. Ein multmediales zeitbasiertes Lehr - und Lernsystem. Diplomarbeit, Fachhochschule für Wirtschaft und Technik, 2000.

[6] Ciesla, Gräßler, Ordenewitz . *Integriertes Ressourcenmanagement*. 1995.

[7] Courd, Yourdan. *Objectoriented Analysis*. Yourdon Press, 1990.

[8] Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John. *Entwurfsmuster*. ISBN 3-89319-950-0. Addison Wesley, Bonn, first edition, 1996.

[9] Institute for Computer Architecture German National Research Center for Information Technology and Software Technology (GMD FIRST). Compiler Construction with Java. `http://www.first.gmd.de/cogent/catalog/java.html`, 1999.

[10] Klein Gerwin. JFlex - The Fast Scanner Generator for Java - Version 1.2.2. http://www.jflex.de/, 1999.

[11] Hollemann . *Rechnerunterstütztes Ressourcenmanagement*. 1995.

[12] Valerie Illingworth. *Dictionary of Computing*. ISBN 0-19-853855-3. Oxford University Press, Walton Street, Oxford OX2 6DP, forth edition, 1998.

[13] Sybase Inc. *Sybase Adaptive Server Performance and Tuning Guide*. Seite 5-17.

[14] Java User Group Stuttgart - JUGS e.V. Java Forum Stuttgart. http://www.jugs.de/jfs2000/index.html, 2000.

[15] JGuru - John Zukowski. How do I get a listing of the files in a directory? . http://www.jguru.com/jguru/faq/view.jsp?EID=89870.

[16] Lampkemeyer, Dittmer, Petersen . *Ressourcenmanagement*. 1989.

[17] Lesk M. E. and Schmidt E. The Lex and Yacc Page. http://www.combo.org/lex_yacc_page/, 1999.

[18] Metzen, Szallies, Pfaffenholz, Töpfler. Architekturgetriebene Generierung für objektorientierte Softwaresysteme. *Objekt Spectrum*, 02/2000.

[19] Microsoft Corporation. Comparing Microsoft Transaction Server to Enterprise JavaBeans. http://www.microsoft.com/Com/wpaper/mts-ejb.asp.

[20] Sun MicroSystems. The java tutorial - a practical guide for programmers. http://java.sun.com/docs/books/tutorial/reflect/class/index.html, December 2000.

[21] ObjectWatch, Inc. SCALABILITY IN EJB AND MTS. http://www.objectwatch.com/issue18.htm.

[22] Rana Bhattacharyya. JAR compressor libary. http://www.mycgiserver.com/~ranab/jar/index.html.

[23] Rosch M. Generierungstechnik für die Implementierung von Business-Objekten. *Objekt Spectrum*, 06/1999.

[24] Rösch Consulting. Rösch Consulting, fehlerfreie Fachkonzepte u. Software, objektorientierte Software-Komponenten CORBA, DCOM. `http://www.roesch.com/`, 1999.

[25] Sun Microsystems. AUTHORIZED JAVA LICENSEES OF J2EE TM. `http://java.sun.com/j2ee/licensees.html`.

[26] Sun Microsystems. Enterprise JavaBeans(TM) Specification Version 1.0 . `http://java.sun.com/products/ejb/docs10.html`.

[27] Sun Microsystems. Industry Opinions On Enterprise JavaBeansTM (EJBTM) vs. COM+/MTS. `http://java.sun.com/products/ejb/ejbvscom.html`.

[28] Sun Microsystems. JAR Guide. `http://java.sun.com/products/jdk/1.1/docs/guide/jar/jarGuide.html`.

[29] Sun Microsystems. Java Beans - the only component architecture for java technologies . `http://java.sun.com/products/javabeans/`.

[30] Sun Microsystems. javac - The Java Compiler. `http://java.sun.com/products/jdk/1.1/docs/tooldocs/win32/javac.html`.

[31] Sun Microsystems. JavaTM 2 SDK, Enterprise Edition Documentation Bundle . `http://java.sun.com/j2ee/j2sdkee/techdocs/`.

[32] Sun Microsystems. Sun Microsystems. `http://www.sun.com/`.

[33] Sun Microsystems. The J2EE Application Model. `http://java.sun.com/j2ee/`.

[34] Sun Microsystems. Writing Advanced Applications. http://developer.java.sun.com/developer/onlineTraining/Programming/JDCB\%ook/bmp4.html\#attr.

[35] Sybase Inc. Enterprise Applikation Server 3.5. http://www.sybase.com/eas/.

[36] Sybase Inc. PowerJ - The Web development environment for Sybase EA-Server. http://www.sybase.com/products/internetappdevttools/powerj/.

[37] Bernd Thalheim. *Entity-Realtionship Modeling, Foundations of Database Technology*. ISBN 3-540-65470-4. Springer Verlag, Berlin, Heidelberg, New York, 1998.

[38] Warnecke, Romberg, Hornscheid . *Toolmanagement*. 1991.

[39] What is? The IT specific encyclopedia. http://www.whatis.com/, 1999.