Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Hochverfügbare Middlewareplattform für ein mobiles Patientenbetreuungssystem

Diplomarbeit

Leipzig, Juli 2001

vorgelegt von:
Matthias Löbe

Inhaltsverzeichnis

1. Einleitung

In praktisch allen Bereichen des täglichen Lebens hat heute die Informationstechnologie Einzug gehalten. Rechnerhardware wird seit vielen Jahren deutlich leistungsfähiger und preiswerter. Gleichzeitig werden computerbasierte Lösungskonzepte für einen immer größeren Aufgabenkreis entwickelt. Im Bereich der Medizin sind viele Fortschritte ohne Hilfe von Computern kaum noch denkbar. Mit ihrer Hilfe können Aufgaben zuverlässig, schnell und automatisiert ausgeführt und ausgewertet werden.

Diese Diplomarbeit ist Teil eines Forschungsprojekts, in welchem ein mobiles Betreuungssystem für Patienten mit Gedächtnisstörungen in Folge von Hirnschädigungen entwickelt worden ist. Für dieses mußte eine *Verteilungsplattform* für die zu entwickelnden Softwarekomponenten gefunden werden, die den speziellen Ansprüchen des Einsatzes in verschiedenen, räumlich verteilten Umgebungen genügte. Im Rahmen der Arbeit war daher zu untersuchen, welche Middleware-Umgebungen generell verfügbar sind und inwiefern diese den Kriterien der Verteilbarkeit, Componentware-Unterstützung und Hochverfügbarkeit entsprechen.

Eine weitere Zielstellung bestand in der Entscheidung für eine einfache und effiziente Art der *Informationsspeicherung.* Für die gewählte Plattform sollte ein geeigneter Persistenzmechanismus gefunden und implementiert werden, der sowohl die Konsistenz der Daten wahrt, als auch einen leistungsfähigen und standardisierten Zugriff ermöglicht. Der Schwerpunkt der Aufgabe lag dabei auf der Untersuchung von Problemen bei der Abbildung der Datenstrukturen der verwendeten Programmiersprache auf das konzeptionelle Datenmodell des Persistenzmediums. Besondere Berücksichtigung wird hier der Objektdatenbank POET zuteil.

Auf die grundlegenden Eigenschaften mobiler Betreuungssysteme allgemeiner Art soll nicht eingegangen werden. Es werden nur die Ansprüche untersucht, die sich speziell an *Mobtel als mobiles verteiltes System* richten. Zum einen hat der Bereich Telemedizin/ Telerehabilitation erst in den letzten Jahren mit der Vernetzung der Computersysteme und der Verfügbarkeit preiswerter drahtloser Kommunikationshilfen an Bedeutung gewonnen, so daß nur eine sehr geringe Zahl von Projekten auf vergleichbarer Ebene existieren. Die Forschung auf diesem Gebiet befindet sich erst am Anfang, eine Standardisierung von Leistungen ist noch nicht gegeben. Zum anderen war Mobtel zum Zeitpunkt des Beginns der Arbeit schon so weit spezifiziert, daß die Betrachtung einer generellen Kategorisierbarkeit nicht Gegenstand dieser Arbeit werden würde. Vielmehr sollte Mobtel vordringlich basale Funktionen für eine spezielle Personengruppe von hirngeschädigten Patienten unterstützen und zu einem späteren Zeitpunkt um fortgeschrittene Funktionen erweitert werden können.

Eine letzte Zielstellung war die Untersuchung von *Skalierbarkeit und Hochverfügbarkeit*. Obwohl beide Aspekte prinzipiell getrennt analysiert werden können, ist es hier von Vorteil, die Betrachtung zusammenzufassen. Wird zum Zweck einer höheren Programmgeschwindigkeit die Zahl der ausführenden Instanzen erhöht und arbeiten diese Instanzen autark, wirkt sich dies auch auf die Verfügbarkeit aus. Es kommen hierfür Möglichkeiten, die auf Hardware oder auf Software basieren in Betracht.

Im ersten Kapitel wird ein Überblick über die Anforderungen und Strukturen des Mobtel-Projekts gegeben, die dieser Arbeit zugrunde lagen. Für eine erfolgreiche Bearbeitung ist eine genaue Analyse der geplanten Anwendungssituationen und der Erwartungen an das System unerläßlich. Aus ihr leitet sich die prinzipielle Vorgehensweise und der Aufbau der Architektur ab. Von Beginn an wurde ein verteiltes Modell verwendet, bei welchem jede Komponente eine klar abgrenzbare Funktion erfüllt. Jeder Teil kommuniziert über standardisierte Schnittstellen, so daß andere Komponenten integriert oder ausgetauscht werden können. Es existieren keine prinzipiellen Abhängigkeiten von bestimmter Hard- oder Software. An dieser Stelle wird die derzeitige Implementation vorgestellt. Dabei werden die tragbare Gedächtnishilfe, die zentralen Server und die Anwendungsumgebung für den Betreuer erläutert. Letztlich wird auch auf rechtliche Aspekte z.B. des Datenschutzes und der Schweigepflicht eingegangen, soweit sie für die Arbeit von Bedeutung waren.

Schon zu Beginn war die Wahl der Programmiersprache auf Java gefallen. Das zweite Kapitel beschreibt die grundsätzlichen Möglichkeiten der Speicherung von Java-Objekten in Datenbanken. Es wird versucht, einen vollständigen Überblick zu geben und die Vor- und Nachteile der einzelnen Verfahren herauszustellen. Dabei wird im besonderen auf die objektorientierte Datenbank POET eingegangen, da deren Evaluation Teil der Gesamtaufgabe war. Die Gründe, die schließlich zur Nutzung einer Relationalen Datenbank führten, sind hier ebenfalls dargelegt. Im letzten Abschnitt des Kapitels werden die grundlegenden Probleme und Lösungsansätze bei einer objektrelationalen Abbildung besprochen, die auf den unterschiedlichen Mächtigkeiten des Objektkonzepts in Java und des Relationentupels in SQL-Datenbanken beruhen.

Da Mobtel als räumlich verteilbares System konzipiert wurde, kam der Auswahl eines geeigneten Kommunikationsprotokolls große Bedeutung zu. Wichtige Kriterien waren dabei die Plattformunabhängigkeit, die Einfachheit der Programmierung, eine hohe Abarbeitungsgeschwindigkeit von Aufgaben sowie die zeitgenaue und garantierte Zustellung an die Patienten. Im dritten Kapitel werden zuerst die Grundlagen für die Kommunikation über Prozeßgrenzen erläutert. Danach werden die darauf aufbauenden, zur Zeit verfügbaren Kommunikationsprotokolle für verteilte Komponenten vorgestellt und die Vorteile der verschiedenen Alternativen aufgezeigt. Da sich Enterprise JavaBeans als am besten geeignet erwiesen, wird der sie umfassende Standard, die Java 2 Enterprise Edition, noch genauer betrachtet. Dabei wird vor allem auf die zur Verfügung stehenden Dienste und Programmierschnittstellen eingegangen.

Ein weiterer Abschnitt beschäftigt sich mit dem zentralen Problem der Persistenz. Es existieren verschiedene Typen von EJB, die auf unterschiedliche Art und Weise zur permanenten Sicherung von Objektinhalten eingebunden werden können. Weiterhin werden Einschränkungen durch die Nutzung von EJB und die damit verbundenen Folgen erwähnt. Letztlich wird auf die softwareseitige Unterstützung von EJB durch Application Server, deren Entstehung und Zukunftsaussichten eingegangen.

Skalierbarkeit und Ausfallsicherheit sind zentrale Anforderungen an Projekte im medizinischen Sektor, gerade wenn es sich bei den Zielpersonen um eingeschränkt selbstständige Personen handelt. Die denkbaren Konzepte und Lösungen werden im vierten Kapitel erörtert. Dabei werden sowohl hardwareseitige als auch softwareseitige Varianten vorgestellt. Konkrete Möglichkeiten zur Erhöhung der Verfügbarkeit von J2EE Application Servern und daraus entstehenden Schwierigkeiten für die einzelnen EJB-Typen werden ebenfalls behandelt.

Da Enterprise JavaBeans ein relativ junger Industriestandard sind, gibt es noch wenig Erfahrungen auf praktischem Gebiet. Oft wird daher die Behauptung erhoben, daß EJB für echte Anwendungen ungeeignet, da unausgereift sind. Im letzten Kapitel wird eine J2EE-Applikation vorgestellt, die die prinzipielle Implementierbarkeit einer performanten Anwendung unter Beweis stellt und mit der zusätzlich Leistungstests des verwendeten Application Servers erfolgten. Daraus geht hervor, daß die Einsetzbarkeit sehr wohl gegeben ist.

2. Grundlagen und Anforderungen des Verbundprojekts Mobtel

In diesem Kapitel soll nach einer Einleitung ein Überblick über zu von Mobtel [*mob*ile Einsatzszenarien in der *Tel*emedizin] zu leistenden Aufgaben gegeben werden. Dabei wird sowohl auf die Ausgangssituation bei Beginn des Mobtel-Projekts eingegangen als auch ein Beispiel für einen typischen Anwendungsfall gegeben. Danach wird die Gesamtarchitektur vorgestellt sowie die einzelnen Schichten und deren Funktion und Integration beschrieben. Es wird ferner dargelegt, warum sich Java optimal zum Programmieren Verteilter Anwendungen eignet. Zum Schluß wird auf Probleme eingegangen, die sich durch die Verwaltung besonders sensibler, persönlicher Daten ergeben und welche durchaus Einfluß auf die Verteilung der Aufgaben am Einsatzort haben können.

2.1. Einleitung

Das Verbundprojekt Mobtel wurde 1998 vom Sächsischen Staatsministerium für Wissenschaft und Kunst sowie dem Sächsischen Staatsministerium für Wirtschaft und Arbeit initiiert. In einer Partnerschaft aus universitären Institutionen, öffentlichen Forschungseinrichtungen und privatwirtschaftlichen Firmen sollte ein Beitrag zu fachübergreifender und praxisrelevanter Projektarbeit in Sachsen geleistet werden.

Ziel des Projekts war die Entwicklung eines mobilen Betreuungssystems für Menschen, die in Folge von Hirnschädigungen unter Gedächtnisstörungen leiden. Die Auswirkungen solcher Schäden beeinflussen in ganz erheblichen Maße deren Fähigkeit zu einer selbstständigen Lebensweise. Da eine vollständige Rehabilitierung nicht in jedem Fall möglich ist, sind die Patienten lange Zeit auf sie unterstützende und betreuende Personen angewiesen. Dabei wäre eine durchgehende Betreuung in vielen Fällen nicht notwendig, da die Störungen nur in unregelmäßigen Abständen auftreten. Mobtel bietet hier eine Hilfsmöglichkeit, die den Patienten weiter autark handeln läßt und nur bei Anforderung durch ihn selbst oder im Falle eines objektiv nötigen Eingriffs aktiv wird. Die gewonnene Privatsphäre und die geringere Abhängigkeit von anderen Personen kann die Lebensqualität deutlich erhöhen.

Der Einsatz eines verteilten Betreuungssystems bietet auch für den Betreuer eine Vielzahl von Vorteilen. So ist eine Kommunikation mit dem Patienten über das Gerät zu jeder Zeit möglich. Das Zuweisen von Aufgaben, die ein Patient wahrnehmen soll, läßt sich mit Hilfe von Vorlagen bis zu einem gewissen Grad automatisieren. Außerdem läßt sich auf elektronischem Weg ein größerer Personenkreis in die Koordinierung der Betreuung einbinden. Insgesamt sollten sich der nötige Aufwand und auch die Kosten verringern. Durch die hier entwickelten Modelle werden neue Erkenntnisse für ähnlich geartete Fälle gewonnen.

Die an das Mobtel-Projekt gestellten Aufgaben konnten erfolgreich realisiert werden. Das System wird als Hilfsmittel in der Grundlagenforschung Verwendung finden, auf deren Basis sich später weitergehende Resultate erzielen lassen. Es wird aber ebenfalls in praktischen Einsatz gelangen und so die tatsächliche Lebensqualität einer Reihe von Patienten verbessern. Der hier verwendete Ansatz der Projektführung offenbart die Chancen und die Schwierigkeiten einer interdisziplinären Zusammenarbeit: die personellen Kompetenzen zu nutzen und durch enge Abstimmung in die Arbeit einfließen zu lassen. Nur durch die intensive Kooperation der am Projekt beteiligten Partner und deren Spezialisierung auf die verschiedenen Teilgebiete wurden die vorliegenden Resultate erreicht.

2.2. Problembereich

Das Hauptziel des Verbundprojekts Mobtel besteht in der Unterstützung der hirngeschädigten Patienten bei der Bewältigung ihres gewohnten Tagesprogramms. Dieses besteht neben einer Reihe privater Dinge aus objektiv wichtigen Aktionen für die Therapie und die Gesundheit des Patienten. Durch die erlittene Gehirnschädigung können folgende Leistungen gestört sein:

- das Ermessen zwischen belanglosen und zu speichernden Informationen
- der Informationserhalt über den gesamten Zeitraum
- die Entscheidung, bei Erreichen des Ausführungszeitpunkts mit der Aktion zu beginnen
- die Erkenntnis, einen laufenden Vorgang nach einer bestimmten Zeit bzw. nach Ausführung wieder zu beenden
- die Erinnerung, ob die gewünschte Tätigkeit bereits ausgeführt wurde und ob dies erfolgreich geschah

Bisher wurde dazu vom Betreuer für den Patienten ein Tagesplan zusammengestellt, der ihm in Form eines Kalenders oder Terminplaners zur Verfügung stand. Leider kann diese Vorgehensweise nur einen Teil der oben genannten Punkte abdecken. Speziell eine unmittelbare Erfolgskontrolle ist nicht möglich, was bei wichtigen Aufgaben der Anwesenheit einer helfenden Person bedarf. Weiterhin ist nicht gewährleistet, daß der Patient im richtigen Moment den Terminplaner nutzt.

Andererseits können elektronische Hilfen aufgrund ihres eingeschränkten Funktionsumfangs nicht für die Erfüllung aller nötigen Aspekte modifiziert werden. Es war die Entwicklung eines neuen, ganzheitlichen Konzepts erforderlich.

Ziel der Arbeit war die Entwicklung eines mobilen Organizers als Gedächtnishilfe, die für den Patienten leicht zu bedienen und zu warten ist [IRM 99]. Der Patient sollte dabei im besonderen bei der Wahrnehmung von Terminen unterstützt werden. Dazu zeigt und er-

klärt ihm das Gerät die auszuführenden Aufgaben und wirkt mittels eskalierender Meldungen auf deren Ausführung hin. Wird eine maximale Zeitspanne überschritten oder gibt der Patient an, daß die Ausführung fehlgeschlagen ist, so wird ein Alarm an den zuständigen Betreuer gesendet, der daraufhin geeignet reagieren kann. Zusätzlich kann der Patient über eine Sprachverbindung dessen Hilfe in Anspruch nehmen. Der Betreuer kann jederzeit neue Termine hinzufügen oder bestehende ändern, ohne mit dem Patienten in Kontakt treten zu müssen. Die Anzeige eines Termins zu gegebener Zeit wird von dem Gerät sichergestellt, so daß der Betreuer davon ausgehen kann, daß der Patient bei normaler Benutzung des Organizers über das Anstehen eines Termins informiert wurde.

Folgende Akteure können in den gesamten Prozeß involviert werden:

- *Patienten*, die Termine zugesendet bekommen und ausführen müssen. Dies geschieht mittels einer Reihe von Dialogen. Patienten sollen sich auch selbst Termine zuordnen können.
- *Betreuer*, die den Terminplan für ihre Patienten bearbeiten. Sie müssen über den Ablauf der Termine informiert werden und reagieren.
- *Verwandte* sollen eine eingeschränkte Betreuertätigkeit ausüben können. Sie können die Termine ihres Patienten einsehen und selbst neue Termine hinzufügen, wichtige Termine aber nicht löschen.
- weitere Personen wie Ärzte oder Administratoren, die Nebenaufgaben übernehmen

Die kontrollierte Abarbeitung von Terminen ist zentrales Ziel des Projekts. Ein Termin kann einmalig am Ausführungstag stattfinden oder mit einer zu definierenden Periodizität wiederholt werden. Es existieren laut Mobtel-Anforderungsprofil [Mob 98] 4 Klassen von Terminen:

Orientierungsfunktionen

Diese Aufgaben bedingen keine eigene Handlung des Nutzers. Sie dienen lediglich seiner Information. Eine Reaktion ist nicht unbedingt erforderlich.

Tägliche Routinen

Darunter werden häufig erteilte Aufgaben verstanden. Der Patient muß selbst handeln und auf Gerätemeldungen reagieren. Diese dienen jedoch primär der Erinnerung an den Termin, der Ablauf ist dem Patienten im allgemeinen vertraut. Auf Fehler kann der Betreuer meist mit einer gewissen Verzögerung antworten.

Erledigungen

Hierunter können auch einmalige Aufgaben fallen, mit denen der Patient nicht vertraut ist. Erledigungen müssen meist in einem gewissen Zeitraum abgearbeitet werden.

Termine

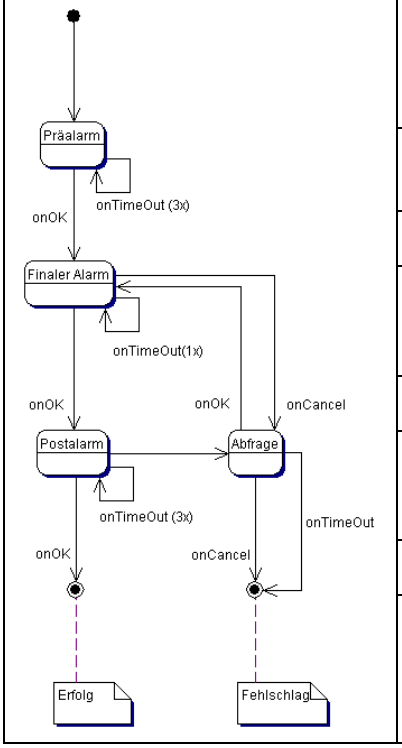
Termine sind die komplexesten Aufgaben. Sie müssen zu einem vorgegebenen Zeitpunkt erledigt werden. Oft muß der Patient hierfür vorher den Ort wechseln, weiterhin müssen eine Reihe von Präkonditionen erfüllt sein. Termine haben meist die höchste Ausführungspriorität. Häufig sind in ihre Abarbeitung noch andere Personen bzw. Organisationen involviert.

Die Abgrenzung zwischen diesen Gruppen ist sehr unscharf, in Wirklichkeit wird sich es oftmals um Mischformen handeln. Der Erfolg einer Aufgabe kann durch Postkonditionen festgestellt werden. Allerdings ist zu beachten, daß der Patient subjektiv über Erfolg/ Mißerfolg entscheidet, d.h. ist gewissen Situationen der Überlastung oder Unwissenheit kann er dem System eine falsche Erfolgsmeldung geben.

Ein Termin besteht für den Patienten aus einer Reihe von Meldungen (*Alarmen*), die ihn zeitlich und sachlich bei der Wahrnehmung von Aufgaben unterstützen sollen. Der Zustand, in dem sich der Pager befindet, hängt in erster Linie von der Patientenreaktion ab. Unter bestimmten Umständen müssen mehrere Alarme simultan angezeigt werden. Da jedoch nur eine Meldung zu einem Zeitpunkt auf dem Gerät erscheinen kann, besitzen alle Alarme eine Priorität. Von allen konkurrierenden Alarmen wird der mit der höchsten Priorität angezeigt. Damit hat auch der Pager selbst Einfluß auf den Zustand eines Alarms.

Das Alarmmodell verleiht einem Termin eine gewisse Transaktionalität. Der Ablauf eines Termins, also die Reihenfolge und Anzahl der Meldungen, hängt von der Art der Abarbeitung der Alarme ab. Auf die verschieden Möglichkeiten muß geeignet reagiert werden. Tritt z.B. ein kritischer Fehler auf, wird der Betreuer vom Scheitern des Termins informiert.

Der logische Aufbau soll an einem Beispiel demonstriert werden:



	Typ	Anzeigezeit	Inhalt	Priorität	Dauer
	Präalarm (Information)	ab 12.45 Uhr	In (15) min müssen Sie ihr Medikament XY einnehmen.	hoch	5 min
		OK: Start Finaler Alarm nicht angezeigt: Priorität erhöhen (1 min) nicht beantwortet: noch 3 Mal starten (5 min), dann Start Finaler Alarm			
	Finaler Alarm (Handlung)	ab 13.00 Uhr	Nehmen Sie jetzt Ihr Medikament XY ein!	sehr hoch	10 min
		OK: Start Postalarm nicht angezeigt: Priorität erhöhen (1 min) nicht beantwortet: noch 1 Mal starten (5 min) nicht möglich: Start Abfrage			
	Postalarm (Bestätigung)	ab 13.10 Uhr	Haben Sie Medikament XY um 13.00 Uhr eingenommen?	hoch	5 min
		OK: Erfolg nicht angezeigt: Priorität erhöhen (1 min) nicht beantwortet: noch 3 Mal starten (5 min) nicht möglich: Start Abfrage			
	Abfrage (Bestätigung)	auf Anforderung	Können Sie das Medikament jetzt einnehmen ?	höchste	1 min
		OK: Start Finaler Alarm nicht beantwortet: Alarm an Betreuer nicht möglich: Alarm an Betreuer			

Tabelle 1: Termin Medikamenteneinnahme

2.3. Architektur

Eine der wichtigsten Anforderungen war es, ein fehlertolerantes und skalierbares System zu entwerfen [SCH 00]. Deshalb ist Mobtel auf einer modernen 3-Schicht-Architektur aufgebaut. Im einzelnen besteht es aus einer Clientschicht, der Serverschicht und der Datenbank. Der Betreuer und andere Personen interagieren mittels eines Thin-Clients (Schicht 1) mit serverseitigen Objekte in einem Webcontainer (Schicht 2). Diese stellen das grafische Nutzer-Frontend dar. Weiterhin sind sie eine Hülle für die eigentliche Geschäftslogik, die sich auf dem Application Server (ebenfalls Schicht 2) befindet. Dieser ist auf einem Komponentenmodell aufgebaut und lädt und speichert die relevanten Daten in ein relationales Datenbankmanagementsystem (Schicht 3). Der Patient bekommt seine Termine auf seinen Pager überspielt. Dies geschieht, indem der Application Server die Objekte in ein geeignetes textbasiertes Format konvertiert. Dieses wurde eigens für das Mobtel-Projekt entwickelt und enthält neben den Termindaten auch Ausführungsanweisungen. Dieses Format ermöglicht es, daß der Pager im Normalbetrieb eigenständig ohne Verbindung arbeiten kann. Die erzeugten Objekte werden mittels eines ebenfalls entwickelten Transportprotokolls zum Pa-

ger übertragen. In regelmäßigen Abständen veranlaßt der Pager eine Übertragung der während dieser Zeit angefallenen Protokolldaten.

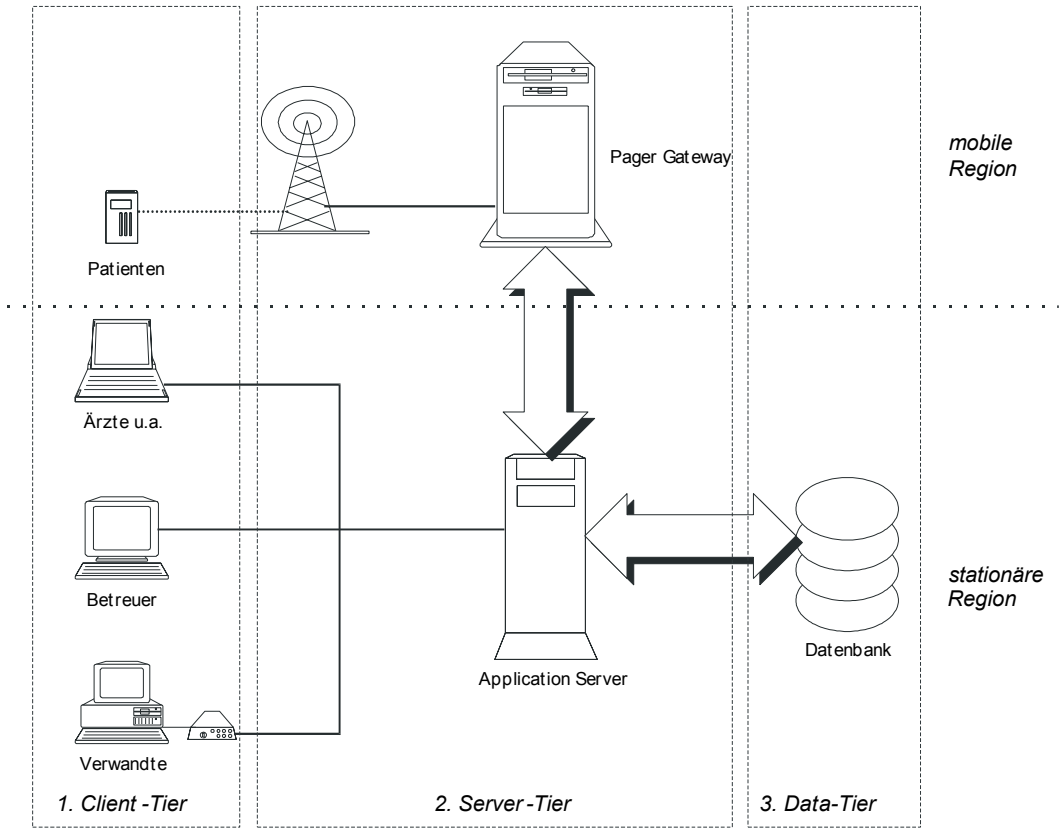


Abbildung 1: Architektur des Mobtel-Projekts

2.3.1. Der mobile PMA

Das wichtigste Teil auf der Clientseite in der mobilen Region ist der PMA (*Personal Memory Assistant*). Das Gerät besitzt einen StrongARM1100-Prozessor mit 66 MHz und 2 MB SRAM. Als Anzeige nutzt er ein 4"-TFT-Display mit Touchscreen-Funktionalität, dessen Auflösung 240x320 Pixel beträgt. Daten können auf einer bis zu 32 MB großen Flashspeicherkarte permanent gesichert werden. Als Betriebssystem wurde Windows CE 2.11 verwendet. Im Gerät sind ein Mikrofon und Lautsprecher integriert, die Sprachkommunikation ermöglichen. Ferner verfügt es über ein GSM-Modul zur mobilen Kommunikation mit dem Server. Es besitzt nur 2 feste Tastaturknöpfe. Weitere Bedienelemente werden über den Touchscreen angezeigt. Die Energieversorgung wird über einen Akkumulator sichergestellt, der eine Betriebszeit von 4 bis 8 Stunden garantieren soll. Der Pager war Teil der Arbeit des Projektpartners RBM Elektronik Automation GmbH Leipzig.

Aus technischen und finanziellen Gründen ist der Patient nicht ständig mit dem Hauptsystem verbunden. Während dieser Zeitabschnitte obliegt dem Pager die Verantwortung, die Termine dem Patienten darzubieten und zu verwalten. Da die hard- und softwareseitige Leistungsfähigkeit des Gerätes um einiges geringer ist als die eines normalen Personal Computers, erfolgt die interne Realisierung der Termine mittels einer an HTML und WML angelehnten Markierungssprache, die durch den Pager interpretiert wird. Diese Sprache wird MML (*Mobtel Markup Language*) bezeichnet. Sie baut wie WML auf einer hierarchischen Struktur auf, die aus Decks und Cards besteht. Decks besitzen mehrere Cards und dienen der logische Kapselung von Programmfunktionen. Cards repräsentieren Bildschirmseiten, die auf dem Display angezeigt werden können. Es kann immer nur eine Card zu einem Zeitpunkt zu sehen sein. Zur Steuerung werden Prioritätsattribute verwendet. Weiterhin besitzt die MML umfangreiche Datums- und Zeitfunktionen, die für die Ausführung der Terminsteuerung unerläßlich sind, ebenso wie die Variablenverwaltung, die Skriptfähigkeiten und die Möglichkeit zur Anzeige mehrfarbiger Elemente. Im Gegensatz zu einem Standard wie WML können MML-Anwendungen flexibel auf Zeit- und Nutzerereignisse reagieren und benötigen für eine zuverlässige Abarbeitung einer Workflowaktion keine durchgängig bestehende Verbindung zum Gateway.

Für bidirektionale Übertragung zum Pagergateway wurde ebenfalls ein eigenes Protokoll verwendet, das MTP (*Mobtel Transport Protokoll*), welches sich an HTTP anlehnt und auf TCP/IP, PPP und GSM basiert.

2.3.2. Java als Programmiersprache

Als Programmiersprache wurde auf Serverseite Java verwendet. Java konnte in das Projekt eine Vielzahl seiner Vorteile einbringen. Zuerst handelt es sich bei Java um eine der jüngsten objektorientierten Sprachen, in deren Entwicklung deshalb viel Erfahrung eingeflossen ist. Des weiteren besitzt Java eine mächtige Klassenbibliothek, die einen standardisierten Zugriff auf wichtige Programmierfunktionen für elementare und fortgeschrittene Anwendungszwecke und ein hohes technisches Abstraktionslevel bietet. Java-Quellcodedateien werden bei der Kompilation zu Byte-Code verarbeitet, der von einer Virtuellen Maschine ausgeführt wird. Der Byte-Code besitzt für alle Betriebssysteme, zu denen es eine Virtuelle Maschine gibt, denselben Aufbau und Befehlssatz. Deshalb können Java-Programme ohne Neukompilierung auf allen bedeutenden Plattformen ausgeführt werden. Java verlangt explizite Typkonvertierung (Casts), erlaubt keine Buffer-Overruns (z.B. bei Feldbereichsgrenzen) und keine unzulässigen Speicherzugriffe, wie sie z.B. durch Pointer entstehen können. Dies alles hat dazu beigetragen, daß sich Java in den letzten Jahren zur beherrschenden Programmiersprache für die serverseitige Programmierung entwickelt hat. Die Evolution der Sprache ist noch nicht abgeschlossen und wird bis zum heutigen Tage fortgeführt, vor allem auf dem Gebiet der Programmierschnittstellen. Dabei ist jedoch anzumerken, daß die schnelle Adaption neuer Trends und die Pflege der bestehenden Klassen auch eine Entscheidung gegen Java begründen könnte. Es besteht die Gefahr, daß in Java geschriebene

Programme mit zukünftigen Versionen der Laufzeitumgebung nicht mehr vollständig kompatibel sein werden, entweder durch Änderungen an den Basisfunktionen oder durch konkurrierende Entwicklungen, die in eine ähnliche Richtung weisen, aber durch Herstellermacht zum Standard werden. Ein Beispiel für Aufwärtsinkompatibilität besteht bei älteren Java-Applets, die aufgrund erweiterter Sicherheitsrestriktionen bestimmte Methoden nicht mehr aufrufen sollen. Solche Funktionen sind als veraltet („*deprecated*“) gekennzeichnet. Sie sind auch in neueren Java-Implementationen weiter vorhanden, geben aber im allgemeinen nur das NULL-Objekt zurück. Dadurch sind die Applets zwar prinzipiell weiter aufrufbar, können aber keine sinnvolle Arbeit mehr leisten.

2.3.3. Serverschicht

Zentrales Element ist der Application Server. Es handelt sich dabei um den Inprise Application Server 4, der zur J2EE-Spezifikation 1.2 kompatibel ist. Er ist für eine Reihe von Plattformen erhältlich und ermöglicht so den Aufbau heterogener Architekturen. Weiterhin ist mit ihm eine einfache Adaption an zu erwartende Entwicklungen im Hard- und Softwarebereich möglich. Als Datenbank wird zur Zeit Oracle verwendet. Da aber keine Oracle-spezifischen Funktionen verwendet wurden, läßt sie sich gegen jede andere SQL-Datenbank austauschen, für die es einen JDBC-Datenbanktreiber gibt.

Ein leistungsfähiger Application Server bietet prinzipiell eine höhere Stabilität, als das durch die Programmierung eigener Serverapplikationen mit den gegebenen limitierten Möglichkeiten erreichbar ist. Beispielsweise wird ein Java-Programm im Fall einer nicht behandelten Ausnahme beendet. Dies ist besonders kritisch bei Laufzeitausnahmen oder gesperrten Ressourcen. Der Zustand, in dem sich eine solche Instanz oder ihre Daten befinden, ist dann häufig undefiniert. Des weiteren sind solche handgemachten Server oft selber der Grund für eine Ressourcenverknappung, da sie aus Gründen der hohen Komplexität weder Transaktionsverwaltung, Zwischenpufferung von Daten noch redundante Verbindungen anbieten. Das Bereitstellen von Methoden zur Skalierbarkeit und Verfügbarkeit erfordert viel Erfahrung und Wissen über die Abläufe in Verteilten Systemen. Durch Nutzung vorhandener Frameworks kann das Hauptaugenmerk auf die Umsetzung des gewählten Projektmodells gelegt werden.

Die Gründe für die Auswahl der Serverarchitektur werden in den folgenden Kapiteln näher erläutert.

2.3.4. Clientschicht der Betreuerregion

Der Zugriff auf die vom Application Server angebotenen Dienste erfolgt in HTML-Browsern. Diese sind praktisch auf jedem Client-PC installiert. Eine spezielle Zugangssoftware ist nicht notwendig. Die Benutzung von HTML eliminiert häufige Fehlerquellen, die mit der lokalen Installation von Programmen oder der fehlerhaften Wartung durch den

Anwender zusammenhängen. Die HTML-Seiten werden dynamisch durch JSP *(Java Server Pages)* erzeugt, einer serverseitigen Skriptsprache, die über eingebetteten Java-Code auf die volle Funktionalität verteilter Anwendungen zugreifen kann.

Eine weitere Aufgabe ist die Bereitstellung einer benutzerfreundlichen Oberfläche, mit der die Betreuer Patienten und Termine verwalten. Dazu gehört neben einer Reihe von Eingabemasken auch die Möglichkeit, gegenwärtige und abgearbeitete Termine detailliert zu verfolgen. Weiterhin müssen die Termine eines Patienten dem Betreuer in einer attraktiven und übersichtlichen Weise präsentiert werden, sowohl für den aktuellen Tag, als auch in anderen Zeitintervallen. Wichtig sind ebenfalls Such- und Sortierfunktionen, die es dem Betreuer erlauben, sich einen genauen Überblick über bestimmte Attribute und ihren Status zu verschaffen. Adaptive Oberflächen, die dem Anwender nur die Möglichkeiten anbieten, die er auch ausführen können soll, lassen sich aufgrund der Erzeugung des HTML-Codes auf Serverseite sehr einfach verwirklichen. Natürlich spielt auch die Sicherheit der oft vertraulichen Daten eine gewichtige Rolle. Während die Serverkomponenten und deren Kommunikation untereinander aufgrund ihrer räumlichen Geschlossenheit ein unwahrscheinlicher Abhörpunkt sind, muß der Transport über Internetverbindungen auf jeden Fall gesichert werden, beispielsweise mit Paßwörtern, Zertifikaten und SSL.

2.4. Rechtliche Aspekte

Bei dem Entwurf eines Patientenbetreuungssystems muß in besonderem Maße an rechtliche Konsequenzen bei der Abarbeitung personenbezogener Abläufe durch digitale Speicher- und Verarbeitungssysteme gedacht werden [REI 00]. Dabei ist es vorteilhaft, wesentliche Fakten bereits vor Einsatzbeginn zu erkennen und zu analysieren.

Datenschutz

Zu allererst betrifft dies die Krankendaten der Patienten, die aufgrund ihrer weitreichenden Beschreibung des Zustands einer natürlichen Person den Regeln des Bundesdatenschutzgesetzes unterliegen. Dabei ist die digitale Speicherung von Patientenakten und –befunden grundsätzlich erlaubt. Andererseits unterliegt die Nutzung/ Verarbeitung strengen Vorschriften, um den Mißbrauch oder den Zugriff für Unbefugte zu verhindern. Wichtig ist dabei im besonderen, daß der Patient der Verwendung seiner Daten schriftlich zugestimmt haben muß. Doch auch dann behält er die Entscheidungsgewalt über seine Daten. Sie bleiben praktisch sein Eigentum; er kann die Erlaubnis auch zurückziehen. Das ist speziell für die Speicherung in Datenbanken und das Anlegen von Sicherungskopien von Bedeutung. Prinzipiell dürfte sich eine Datenbank nicht ohne weiteres außerhalb der behandelnden Einrichtung befinden. Eine räumliche Verteilung von Anwendungsserver und Datenbankserver kann aber erhebliche Auswirkungen auf die Leistungsfähigkeit und Verfügbarkeit des Gesamtsystems haben.

Schweigepflicht

Noch weitergehende Folgen können sich durch die allgemeine Schweigepflicht für Angehörige von Heilberufen ergeben. Die unbefugte Weitergabe einer während (oder im Zusammenhang mit) einer Behandlung anvertrauten Mitteilung ist ein Straftatbestand. Darunter fällt schon die Tatsachenoffenbarung des Arztbesuches. Der Abgleich von Patiententerminen zu Koordinationszwecken durch externe Personen ist folglich nicht zulässig. Daraus ergeben sich Auswirkungen auf das Design und den Umfang der Objektklassen, speziell der Patientendatensätze, sowie auf die Anwendungsfälle des Projektes. Ferner ist die Übermittlung der Patientendaten an Dritte nur unter begrenzten Voraussetzungen, beispielsweise an nachbehandelnde Ärzte oder Krankenkassen, zulässig. Daraus folgt, daß andere Einrichtungen dazu der ausdrücklichen Erlaubnis des Patienten bedürfen.

Weiterhin besitzen Personen mit Schweigepflicht ein Zeugnisverweigerungsrecht gegenüber offiziellen Stellen, z.B. bei Ermittlungen gegen Patienten. Daraus folgt ein Beschlagnahmeverbot für Patientenakten, um ein Unterlaufen der Schweigepflicht zu verhindern. Externe Bearbeiter müssen also auch für den Schutz der Daten am Aufbewahrungsort und während der Übertragung sorgen. Ein weiteres Problem ergibt sich aus der Tatsache, daß sich die externen Bearbeiter im allgemeinen nicht auf das Zeugnisverweigerungsrecht berufen können, da sie nicht der Schweigepflicht unterliegen. Dem Patienten muß klar sein, daß er entweder auf elementare Rechte verzichten muß oder die Daten auch für das IT-Fachpersonal nicht zu lesen sind, dementsprechend auch nicht für den Datenbankadministrator oder den Anwendungsprogrammierer, z.B. durch den Einsatz von Verschlüsselung.

Die Verantwortung für die korrekte Abhandlung liegt hier bei der den Patienten betreuenden Institution, die Mitarbeiter und Patienten vollständig im Hinblick auf Datenschutz, Schweigepflicht und Haftungsfragen beraten muß.

Zusammenfassung Kapitel 2

Das Mobtel-Projekt baut auf einer 3-Schicht-Architektur auf. Auf Clientseite befinden sich der PMA und die Anwendermasken des Betreuungssystems. Beide Clients interpretieren Hypertext-Dokumente in einer Browserumgebung und können deshalb als Thin-Clients bezeichnet werden. Auf der Serverseite befinden sich das GSM-Gateway und ein Web-Server zur Entgegennahme der Anfragen des jeweiligen Clients. Weiterhin gibt es als zentrale Instanz den Application Server, der die eigentliche Abarbeitung der Programmteile übernimmt. Die Daten werden dagegen auf einer eigenen Schicht in einer Relationalen Datenbank gesichert.

3. Persistenzabbildung von Java-Objekten

Informationen bilden das Kernstück jedes Computerprogramms. Informationen müssen gelesen, verändert, kopiert und gespeichert werden. Informationen abzurufen und dem Nutzer anzuzeigen ist letztendlich auch der Zweck der Mobtel-Projekts. In praktisch allen wichtigen Situationen gibt es den Bedarf, Daten permanent zu sichern. Datenverlust stellt über den Informationsverlust hinaus wegen der starken Abhängigkeit der einzelnen Teilaufgaben voneinander eine der größten Gefahren dar. Inkonsistente Datenstrukturen können gravierende Auswirkungen haben und vom Programm nur schwer erkannt oder gar korrigiert werden. Weiterhin ist es bedeutsam, Daten wieder vollständig auslesen zu können, vor Unbefugten zu sichern und die Integrität sicherzustellen.

Im vorigen Kapitel wurden die grundlegenden Anforderungen an das Mobtel-Projekt und die daraus entstandene Architektur erläutert. Im Rahmen der Arbeit lag der Fokus auf der Untersuchung von Möglichkeiten der Persistenzabbildung. Diese muß offensichtlich auf Seite der Serverschicht geschehen, welche die Daten zentralisiert verwaltet. Der Application Server schreibt dabei kein Datenzugriffsmodell vor, dies obliegt der verwendeten Programmiersprache. Programme, die auf Java basieren, besitzen eine ganze Reihe von Möglichkeiten, Daten in verschiedene Datenquellen zu schreiben und aus ihnen zu lesen. Diese sollen vorgestellt und auf ihre Tauglichkeit hin untersucht werden.

Intensiver wird dabei die objektorientierte Datenbank POET behandelt. Diese war ursprünglich als Persistenzmedium ausgewählt wurden, da sie zur Speicherung stark strukturierter Daten am geeignetsten schien.

3.1. Serialisation

Bei der persistenten Speicherung von Daten wird fast ausschließlich an das Schreiben in Datenbanken gedacht, obwohl auch für andere Zwecke Daten gesichert werden müssen. Eine interessante Eigenschaft der Programmiersprache Java ist die leichte Verwendbarkeit der Serialisierung.

Während der Ausführung eines Programms befinden sich sowohl Code als auch Daten im Hauptspeicher des Rechners. Unter *Serialisierung* versteht man die Fähigkeit, ein Java-Objekt von dem eventuell hardwarespezifischen Format, in dem es im RAM liegt, in ein einheitliches Format zu konvertieren, das man beispielsweise in eine Datei schreiben kann.

Serialisierung ist nicht gleichzusetzen mit Persistenz. Persistenz dient zum dauerhaften Schreiben von Daten über das Ende des Programms hinaus auf einen externen Datenträ-

ger. Serialisierung kann auch das leisten, die Hauptanwendung aber ist das Transferieren von Objekten über Netzwerkverbindungen. Im Gegensatz zu anderen Sprachen können Java-Programme Objekte untereinander *per value* austauschen, d.h. eine echte Kopie des Objektes übergeben statt einer Referenz.

Damit ein Objekt serialisiert werden kann, muß dessen Klasse das leere Interface `java.io.Serializable` implementieren. Der eigentliche Schreibvorgang wird von einem speziellen Ausgabestrom übernommen. Unter einem Ein- oder Ausgabestrom (*Stream*) versteht man geordnete Bytefolge unbestimmter Länge. Das Stromkonzept erlaubt eine Abstraktion beliebiger Quellen und Ziele (z.B. Dateien, Filter, Datenbanken und URLs) vor der Anwendung.

Mit der Klasse `ObjectOutputStream` lassen sich Objektströme transparent serialisieren, indem die Methode `writeObject(Object o)` aufgerufen wird. Diese Methode schreibt alle nicht-statischen, nicht-transienten Instanzvariablen inklusive der aus Superklassen geerbten in den Ausgabestrom. Dabei werden die einzelnen Instanzvariablen ebenfalls serialisiert. Referenzbeziehungen zwischen ihnen bleiben bestehen. Weiterhin wird der Klassenname des übergebenen Objekts und die Klassensignatur gesichert. Beim Deserialisieren wird zuerst eine Instanz der Klasse erzeugt ihr dann die Attributwerte zugewiesen. Das erzeugte Objekt hat anschließend dieselbe Struktur und denselben Zustand, den das serialisierte Objekt hatte.

Die Serialisation wird komplett von der Virtuellen Maschine übernommen. Diese muß zur Laufzeit ermitteln, welche Attribute das zu serialisierende Objekt besitzt. Dazu wird das Reflection-API verwendet, welches dynamisches Instantiieren von Klassen und das Auslesen von Methoden und Parametern ermöglicht.

Das Serialisieren zum permanenten Speichern von Informationen in Dateien hat in der Praxis keine Bedeutung.

3.2. JDBC

JDBC (Java Database Connectivity) ist die Standardschnittstelle von Java für den Zugriff auf relationale Datenbanken. JDBC verbirgt alle Datenbank-abhängigen Details durch eine abstrakte Zugriffsschnittstelle. Basis für JDBC ist die SQL/CLI der X/Open. Dabei handelt es sich um eine Variante der Einbettung von SQL-Befehlen in Java-Quellcode. Der Programmierer kann dabei auf eine prozedurale Schnittstelle zur Datenbank zurückgreifen, die *Einbettung* der SQL-Befehle erfolgt *dynamisch* zur Laufzeit [SAA 00]. JDBC ist konzeptionell mit ODBC, einer ähnlichen Lösung von Microsoft für Windows-Plattformen, verwandt.

JDBC ist eine Low-Level-API. Java-Objekte können nicht einfach auf Datenbanktabellen abgebildet werden, sondern müssen SQL-Befehle als Zeichenketten in den Code integrieren. Dadurch erfordert das Programmieren mit JDBC genaues Wissen über das Datenbankschema sowie die Zuordnung von Attributen auf Tabellen.

Für den Zugriff auf eine Datenbank ist ein JDBC-Treiber nötig. Dieser implementiert Schnittstellen der Klasse `DriverManager`. Das Treiberkonzept von JDBC sieht vor, daß die eigentliche Kommunikation mit dem DBMS nur über den JDBC-Treiber erfolgt. Da dieser jederzeit gegen einen anderen austauschbar ist, können Java-Programme ohne Quellcodeänderungen mit verschiedenen Datenbanken kommunizieren. Die Treiberklassen werden zur Laufzeit geladen; entweder durch explizites Angeben des Namens oder das Auslesen der Systemeigenschaft `sql.drivers`.

Es gibt 4 verschiedene Typen von JDBC-Treibern:

- *Typ 1* bezeichnet die JDBC-ODBC-Bridge. Hier werden JDBC-Aufrufe in ODBC-Aufrufe umgewandelt. Dadurch ist nur eine ODBC-Datenquelle und kein spezieller JDBC-Treiber nötig. Allerdings ist ODBC vorwiegend auf Windows-Rechnern zu finden und die Bridgelösung arbeitet nicht sonderlich stabil. Diese Möglichkeit ist nur als Notlösung zu empfehlen.

- *Typ 2* bezeichnet Treiber, die auf proprietären clientseitigen Bibliotheken für ein bestimmtes DBMS basieren. Sie erfordern Binärcode und sind deshalb nicht plattformunabhängig. Allerdings können sie zusätzliche Fähigkeiten eines DBMS ausnutzen.

- *Typ 3* bezeichnet eine Client-Server-Lösungen. Dabei existiert ein Datenbanktreiberclient, der vollständig in Java programmiert ist. Er greift auf einen Treiberserver zu, der die Aufrufe in ein datenbankspezifisches Protokoll überträgt. Der Client ist portabel und kann deshalb auch in Applets Verwendung finden. Bei diesem Lösungsansatz kann der Server oft mit mehreren verschiedenen DBMS kommunizieren oder wird auf demselben Rechner wie die Datenbank installiert.

- *Typ 4* bezeichnet einen reinen Java-Treiber, der JDBC-Aufrufe in das Protokoll der Datenbank übersetzt. Der Client kommuniziert direkt mit der Datenbank, was sich positiv auf die Geschwindigkeit auswirkt. Dieser Typ ist am häufigsten verbreitet.

Der Verbindungsaufbau zur Datenbank geschieht über die Klasse `java.sql.Connection`, die eine logische Verbindung repräsentiert. Zum Aufbau dient die Methode `getConnection` von `java.sql.DriverManager`. Sie erwartet als Parameter eine URL, die das verwendete Protokoll, den Datenbankserver und die Datenbank enthält, sowie ferner einen Benutzernamen und ein Paßwort. War der Verbindungsaufbau erfolgreich, kann über die

Verbindung mittels `createStatement` ein Anweisungsobjekt (`java.sql.Statement`) erzeugt werden. Dieses wird für unparametrisierte SQL-Abfragen verwendet, indem der SQL-String als Parameter der Methode `executeQuery` übergeben wird. Die Ergebnisse der Abfrage werden in ein Objekt vom Typ `java.sql.ResultSet` gekapselt. Die Navigation über die Ergebnismenge erfolgt zeilen- und spaltenweise. Mit `next` kann die nächste Zeile angesprochen werden. Für den Zugriff auf die Spalten stehen diverse getXXX-Methoden zur Verfügung, wobei XXX für einen Java-Datentyp steht. Letztendlich werden die benutzten Ressourcen durch `close` freigegeben. Auch wenn JDBC heute die meistgenutzte Methode für den Datenbankzugriff ist, die Fehleranfälligkeit und schlechte Wartbarkeit der Persistenzkodierung bei Änderungen am Datenbankschema werden zukünftig eine stärkere Verbreitung randere Mechanismen begünstigen.

3.3. SQL-J

Wie im vorigen Kapitel ausgeführt, basiert JDBC auf einer dynamischen Einbettung von SQL in Java-Programme. Dadurch können fehlerhafte SQL-Anweisungen oder Typkonflikte bei der Abbildung der Ergebnismengen erst zur Laufzeit erkannt werden. Das Resultat ist eine Fehlermeldung des DBMS.

Um solche Probleme schon zur Compilezeit erkennen zu können, wurde SQL-J entwickelt. Es basiert auf einer *statischen Einbettung* von SQL, d.h. alle Anweisungen sind zur Übersetzungszeit festgelegt und können außer in ihren Parametern nicht mehr geändert werden. Der Java-Quellcode enthält also spezielle SQL-Anweisungen, die mit `#sql` beginnen. Diese Zeilen werden von einem Präprozessor geparst und durch JDBC-Anweisungen ersetzt. Der Präprozessor führt dabei aber nicht nur eine Sustitution der SQL-Anfragen aus, er prüft sie auf korrekte Syntax, vergleicht die Übereinstimmung der Tabellen- bzw. Spaltennamen mit denen des Datenbankschemas, überprüft die Typkompatibilität zwischen den SQL-Datentypen und den Java-Variablen und kontrolliert Sicherheitsrestriktionen.

Auch in SQL-J wird zuerst eine Verbindung zur Datenbank angelegt. Dies kann mittels JDBC oder mit `#sql context dbContextName` geschehen. Bei letzterer Variante befinden sich die Verbindungsoptionen in einer Konfigurationsdatei. Der Datenaustausch zwischen den SQL-Anweisungen und dem Java-Programm erfolgt über Host-Variablen. Dies sind Java-Variablen, die im lokalen Programmblock sichtbar sein müssen und den Regeln für die Typabbildung zwischen Java und SQL entsprechen. Sie werden mit einem führenden Doppelpunkt versehen und können eine beide Richtungen genutzt werden. Für den Austausch von Ergebnismengen nutzt man Iteratoren. Iteratoren sind ähnlich wie Java-Klassen, die nur Attribute besitzen, aufgebaut. Ihre Attribute entsprechen von Anzahl und Typ den Spalten der SQL-Ergebnismenge. Sie werden vor ihrer ersten Verwendung mit `#sql [modifier] iterator IteratorName (Spaltentyp1 varName1, ...)` definiert. Diese Deklaration erlaubt dem Präcompiler, eine Klasse zu generieren. Von dieser

können wie gewohnt Instanzen in dem Javaprogramm verwendet werden, beispielsweise `Iteratorname meinIt;`. Die Abfrage auf der Datenbank erfolgt dann wieder mit einem SQL-J-Kommando: `#sql meinIt = {SELECT * FROM ... WHERE ...}`. Die Navigation über die Ergebnismenge ähnelt der Vorgehensweise von JDBC: Zeilensprünge werden mit `next()` realisiert, die Spalten werden über die in der Deklaration verwendeten Variablennamen angesprochen.

SQL-J konnte sich ungerechtfertigterweise niemals gegen JDBC durchsetzen. Dies kann an der mangelnden Unterstützung durch SUN oder der fehlenden Bereitschaft der Entwickler liegen, sich mit einem weiteren Programmiermodell vertraut zu machen.

3.4. XML

Ein weiterer Weg, Java-Objekte persistent abzubilden, liegt in der Verwendung von XML. XML ist eine vom W3 Consortium entwickelte Dokumentenbeschreibungssprache. Als solche definiert sie kein neues Zielmodell für die Aufbewahrung von Datenstrukturen, sondern eher ein geeignetes Austauschformat. Es gibt allerdings erste Implementierungen von reinen XML-Datenbanken (Software AG: Tamino) bzw. die Fähigkeit neuer Versionen von DBMS, Abfragen über HTTP (X-Query) auszuführen und ein XML-Dokument, das die Antwortdaten enthält, zurückzusenden.

XML enthält Tags, die untereinander verschachtelt sind. Gültige Tags werden in einer separaten Konfigurationsdatei, dem DTD, definiert. Ein XML-Dokument ist:

- syntaktisch wohlgeformt, wenn es zu jedem öffnenden Tag ein schließendes Tag gibt.
- semantisch gültig, wenn der Aufbau nicht gegen die Grammatik (DTD) verstößt. Insbesondere sind hier die Abfolge und die Vielfachheiten von Elementen beschrieben.

XML-Elemente enthalten ferner Attributlisten. Die darin vorhandenen Attribute können fortgeschrittene Integritätsregeln besitzen, z.B.:

- Attributwerte, bei denen eine Angabe verpflichtend ist (#REQUIRED), aus vorgegebenen Optionen gewählt werden muß („option1|option2“) oder konstant vorgegeben ist (#FIXED)
- eindeutige Bezeichner (IDs) und Referenzen darauf (IDREF)
- Referenzen auf andere URLs (über externe Entities)
- Parameter (ENTITY)
- Einsatz selbst definierter Datentypen (NOTATION)

Bei der Verwendung von XML unterscheidet sich der speichernde Abbildungsweg erheblich vom lesenden. Zum Speichern muß ein Framework entwickelt werden, was syntaktisch und semantisch gültige Dokumente erzeugt. Beim Lesevorgang kommt im allgemeinen ein XML-Parser zum Einsatz, der aus der XML-Datei eine Baumstruktur erstellt. Die Abbildung in Java-Objekte muß dabei zusätzlich noch ausgeführt werden.

Die Verwendung von XML hat viele Vorteile. Für XML-Dokumente ist kein Datenbanktreiber notwendig. Da es sich um einen definierten Standard handelt, sind XML-Dateien universell austauschbar. Nachteilig ist vor allem die geringe Erfahrung mit der Technologie. Ferner kann sich die Länge der Tags ungünstig auf das Verhältnis zwischen Nutzlast und Overhead auswirken und letztlich bleibt es Aufgabe der Applikation, den Daten eines XML-Dokuments eine Bedeutung zuzuweisen.

3.5. Orthogonal Persistentes Java

Alle in den vergangenen Abschnitten angesprochenen Persistenzlösungen haben einen wesentlichen Nachteil, der zu einem späteren Zeitpunkt genauer dargelegt werden soll: Die Unterschiede zwischen dem objektbasierten Typsystem von Java und dem Datenmodell der Persistenzlösung, also z.B. den Relationen eines RDBMS. Soll eine Transition zwischen beiden stattfinden, kann einerseits Java um das Typsystem der Datenbank erweitert werden, wie das durch JDBC-Resultsets oder XML-DOM-Bäume erreicht wird.

Eine andere Möglichkeit ist es, das Typmodell der Programmiersprache auf die Datenbank abzubilden. Diese Vorgehensweise wird als *orthogonale Persistenz* bezeichnet. Dabei sind 3 Faktoren von Bedeutung [SUN 01]:

- Persistenz-*Unabhängigkeit*: Der Programmcode für persistente und transiente Objekte ist gleich, im speziellen gibt es keine expliziten Lade- und Speicherbefehle.
- Typ-*Orthogonalität*: Persistenz ist für alle Objekte der Programmiersprache verfügbar.
- Persistenz durch *Erreichbarkeit*: Es werden alle Attribute eines Objekts persistent gemacht, also alle erreichbaren Wege im Objektgraphen durchlaufen.

Für Java gibt es eine solche Implementierung, *PJama*. In PJama sind standardmäßig alle Objekte persistent. Mit Hilfe einer speziellen Virtuellen Maschine (OPJ VM) werden alle Objekte beim Start eines Programms aus einem persistenten Speicher (persistent store) geladen und bei Beendigung in diesen geschrieben. Ein persistenter Speicher muß nie erzeugt, geöffnet oder geschlossen werden, er steht implizit immer zur Verfügung.

Zur Zeit gibt es eine Einschränkung der Orthogonalität: die Persistenz von einzelnen Threads wird nicht unterstützt, da die Realisierung von Threads in Java auf Unterstützung

durch das Betriebssystem aufbaut. Wäre das aber der Fall, könnte Quellcode ohne Änderungen unter der OPJ-VM ausgeführt werden. So muß die aktuelle Threadklasse, die das Wurzelobjekt bildet, ausdrücklich mit `OPRuntime.roots.add(ThreadName.class)` hinzugefügt werden.

Die Anbindung von PJama an herkömmliche Datenbanken ließ sich aber nur schwer durchführen. Deshalb wurde die Arbeit an PJama wurde zugunsten von *JDO* (Java Data Objects) aufgegeben. JDO lehnt sich stärker an objektorientierte Datenbanken an, definiert aber keine Anbindung von Java an OODBMS und kann Java-Objekte auch auf RDBMS abbilden. Ein Java-Objekt, das mit JDO persistent gespeichert werden soll, muß das Interface `PersistenceCapable` erben. Alle Anfragen auf dem Datenspeicher werden über die Methoden des Interfaces `PersistenceManager` abstrahiert. Dieses Vorgehen widerspricht der geforderten Persistenzunabhängigkeit. Für JDO ist zur Zeit noch keine Implementierung verfügbar.

3.6. Objektdatenbanken

OODBMS sind ursprünglich für eine Reihe von Spezialanwendungen entwickelt worden, die bestimmte Anforderungen an die Persistenzverfahren stellen, welche von Relationalen Datenbanken nicht erfüllt werden. Die zu speichernden Objekte sind häufig sehr *komplex strukturiert*, d.h. aus anderen Objekten zusammengesetzt, besitzen Beziehungen zueinander und müssen Integritäts- und Randbedingungen erfüllen. Weiterhin erlaubt ein ResultSet einer SQL-Anfrage nur das *Navigieren* in der Ergebnismenge, aber nicht in davon referenzierten Objekten. Die Transaktionsverwaltung in RDBMS kann bei restriktiven Einstellungen zu Lesesperren auf Objekten führen und dadurch komplexe Vorgänge, bei denen mehrere Instanzen auf dem Objekt arbeiten, unmöglich machen. Weiterhin können fortgeschrittene *Beziehungsaspekte* zwischen Objektinstanzen wie Assoziation, Aggregation und Komposition [FOW 98] nur beschränkt ausgedrückt werden. Letztendlich besteht weiterhin der *impedance mismatch* zwischen dem Java-Objekt und der deklarativen, mengenorientierten SQL-Relation.

Die Object Database Management Group (ODMG) hat einen Standard veröffentlich, der inzwischen in der Version 3.0 vorliegt. Er spezifiziert, wie Objekte in Datenbanken abgelegt werden. Im einzelnen besteht er aus folgenden Komponenten:

- dem *Objektmodell*: Dieses definiert ein neutrales Datenmodell für OODBMS. Hier dient als Modellierungseinheit der Objektbegriff. Wichtige Punkte sind die Unterstützung von:
    - Objektkapselung (Zustand bestimmt durch Menge und Wert der Attribute)
    - Objektmethoden (Verhalten bestimmt durch Operationen)
    - Objektidentität (Schlüsselattributmengen bestimmen ein Objekt eindeutig)

    - Objektmengen (Extensionen (`extents`) als Abstraktion über die Menge aller Instanzen eines Typ)
    - Objektbeziehungen (explizite binäre Relationen zwischen Typen).
- der *Objektdefinitionssprache* ODL, die zur programmiersprachenneutralen Beschreibung von Klassenstrukturen dient und auf der IDL aufbaut
- der *Objektanfragesprache* OQL, einer auf SQL basierenden deklarativen Anfragesprache, die allerdings nur lesende Zugriffe kennt
- der *Objektmanipulationssprache* (OML); sie definiert die Namenskonvention von Methoden zur Arbeit auf Datenbankobjekten und bewirkt so eine Kompatibilität auf Quelltextebene
- dem *Objektaustauschformat* (OIF), welches den Import und Export zwischen verschiedenen Datenbanksystemen ermöglichen soll

Ferner gibt es eine Sprachanbindung für Java und damit eine einheitliche, objektorientierte Schnittstelle für Java-Programme. Dies stellt einen großen Vorteil dar, da Anwendungen dadurch theoretisch auch mit Datenbanken anderer Hersteller zusammenarbeiten können. Das Java-Binding ermöglicht typorthogonale Persistenz durch deklarative Festlegung der persistenten Klassen. Im Gegensatz dazu verlangt das Binding für C++ eine explizite Oberklasse `d_Object` für alle persistenten Objekte.

Der ODMG-Standard trifft keine Festlegung über den Mechanismus, wie die Objektpersistenz implementiert werden soll. Der Aufbau einer solchen Schicht kann auf verschiedene Weise geschehen:

- über einen Präprozessor, der eine Konfigurationsdatei ausliest und die daran beschrieben Klassen persistent macht (POET)
- über einen Postprozessor, der den Bytecode der Java-Klasse abändert (ObjectStore)
- über eine eigene Implementation der Virtuellen Maschine (GemStone)
- über eine objektrelationale Abbildungsschicht, die einen objektorientierten Zugriff auf relationale Datenbanken ermöglicht (SQL Object Factory, Powertier)

Die ersten beiden Verfahren sind statische Implementierungen. Sie ersetzen die normalen Aufrufe der Objektreferenzen mit Aufrufen, die Operationen auf der Datenbank ausführen. Die anderen Verfahren lassen den Code völlig unverändert.

Eines der Kernprinzipien bei der Entwicklung des ODMG-Standards war die transparente Persistenz. Java und die OO-Datenbank teilen dasselbe Typsystem. Aus einer Klasse können sowohl persistente als auch transiente Objekte erzeugt werden, ohne daß dazu spezielle Schlüsselwörter nötig wären. Eine Klasse wird, wie oben gezeigt, außerhalb von Java *persistenzfähig* deklariert. Ausdrücklich als flüchtig definierte Attribute (mit dem Modifier `transient`) und Klassenvariablen (`static`) werden nicht abgespeichert.

Operationen auf persistenten Objekten müssen zwingend innerhalb einer *Transaktion* ausgeführt werden. Transaktionen werden als Einheit angesehen, mit `begin()` gestartet und mit `commit()` beendet oder mit `abort()` abgebrochen wird. Geschachtelte Transaktionen werden nicht unterstützt. Standardmäßig verfolgt auch der ODMG-Standard einen pessimistischen Synchronisationsmechanismus. Beim Lesen eines Objektes wird eine Lesesperre gesetzt, die verhindert, daß es von anderen Transaktionen geändert werden kann. Eine Schreibsperre verhindert den lesenden und den schreibenden Zugriff. Diese Sperren werden implizit bei Lesen bzw. Modifizieren eines Objekts gesetzt., können aber von dem Programm mit eigenen Werten überschrieben werden.

Ein wesentliches Konzept des ODMG-Standards ist die Persistenz durch Erreichbarkeit. Hierbei wird jedes Objekt persistent, das von anderen persistenten Objekten referenziert wird. Das heißt, daß nicht ein nur einzelnes Objekt, sondern ein ganzes Objektnetz gesichert wird. Werden durch Modifikationen Teile des Netzes abgetrennt, werden diese aus der Datenbank entfernt. Jedes Objektnetz besitzt ein Wurzelelement. Diesem Element muß beim Speichern ein eindeutiger Name zugewiesen werden, mit dem es in der Datenbank identifiziert werden kann. Alle abhängigen Objekte bleiben unbenannt.

Abfragen können auf verschiedene Weise geschehen:

- über den Namen eines Wurzelelements
- mittels Referenz über den Objektgraphen
- durch eine OQL-Query.

Die Vorteile objektorientierter Datenbanken liegen eindeutig in der einfacheren Abbildung der Objekte, ohne daß eine Kodierung von Hand nötig wäre. Bei komplexen Objektstrukturen erzielen sie eine höhere Geschwindigkeit für die Erstellung, den Abruf und das Verfolgen abhängiger Objekte, da dies in RDBMS durch teure Verbundoperationen nachgebildet werden muß. Die Benennung von Methoden nach der OML ist häufig der Namenskonvention von Java ähnlich und daher intuitiver für Programmierer, die noch keine Kenntnisse von Datenbankzugriffen haben.

Nachteilig sind vor allem die geringe Verbreitung der Produkte und die dadurch bestehenden Resistements. Für RDBMS gibt es eine Vielzahl von komfortablen Werkzeugen, Benchmarks und Anwendungsbeispielen. Diese Systeme haben ihre Verfügbarkeit, Stabilität und Skalierbarkeit hinreichend bewiesen. Die Verwendung von SQL garantiert einen universellen Datenaustausch. Im Gegensatz dazu sind einige Aspekte im ODMG-Standard unterspezifiziert und daher herstellerabhängig, beispielsweise die Paketnamen der Standardklassen, die Konstruktoren der Kollektionsklassen oder der Aufbau der Konfigurationsdateien zur Persistenzabbildung.

POET – Eine objektorientierte Datenbank

Im Rahmen des Verbundprojekts Mobtel wurde auch die Möglichkeit untersucht, eine objektorientierte Datenbank als Persistenzschicht zu verwenden. Die Wahl fiel damals auf die POET Object Server Suite 5.1 der Firma POET. Diese ist zum ODMG 2.0-Standard kompatibel.

Die Firma POET beschäftigt sich ausschließlich mit objektorientierten Datenbanktechnologien und ist auf diesem Gebiet gemeinsam mit der Firma ObjectStore der bekannteste Anbieter. Die POET-Version 5.1 lag zum Testzeitpunkt für die Betriebssysteme Windows und UNIX vor. POET unterstützt die Anwendungsentwicklung in C++ und seit Version 5.0 auch in Java. Da die Wahl der Programmiersprache bereits auf Java gefallen war, wurden mit C++ keine Tests durchgeführt.

Während die eigentliche Installation der Datenbank unkompliziert ist, müssen danach aufwendige Konfigurationsarbeiten erfolgen. Zuerst wird zwischen Serverinstallation, Clientinstallation sowie dem POET Software Development-Kit unterschieden.

Das POET-SDK ist nur für den Einsatz auf Entwicklungsrechnern gedacht. Es unterstützt keine Netzwerkzugriffe und nur einen lokalen Zugriff. Es benutzt nicht den Object Server, sondern eine SDK-integrierte Version. Zum Test wurde daher eine separate Object Server Version eingesetzt. Bei dieser müssen nach der Installation in der Datei *ptserver.cfg* Netzwerkzugriffe durch Eingabe eines Aktivierungsschlüssels angegeben werden. Soll nun ein Java-Client auf den Object Server zugreifen, muß vor dem ersten Start auf Clientseite ein Registrierungsprogramm gestartet werden. Außerdem ist der POET-Zugriff nicht plattformunabhängig, so daß einige DLLs lokal vorhanden sein müssen.

POET arbeitet am besten mit Programmen zusammen, die auf demselben Rechner wie die Datenbank laufen. Dazu exisitiert ein spezieller Zugriffsmodus (LOCAL mode). Dieser arbeitet (ohne Netzwerk-Overhead) am schnellsten. Netzwerkverbindung werden über ein spezielles Protokoll hergestellt, so daß dann URLs mit `poet://` beginnen.

POET unterscheidet zwischen persistenzfähigen (*persistance capable*) und persistenzbewußten (*persistance aware*) Klassen. Die ersteren wurden im vorigen Abschnitt bereits erläutert. Persistenzbewußte Klassen sind diejenigen, die mit persistenten Objekten arbeiten. Sie müssen mit einem speziellen Präprozessor (`ptjavac`) kompiliert werden. Dieser sucht im lokalen Verzeichnis nach einer Datei `ptjavac.opt`. Sie enthält die folgenden wichtigen Abschnitte:

- Datenbanken (`databases`): Name der Datenbank
- Schemata (`schemata`): Name des Schemas (enthält Metadaten)

- Persistenzfähige Klassen (`classes`): Hier müssen alle Klassen genannt werden.
- Indizes (`indexes`): Durch Angabe eines Attributs lassen sich Suchvorgänge beschleunigen.

Eine minimale Beispieldatei wäre:

```
[schemata\MobtelDict]

[databases\MobtelBase]

[classes\Patient]
persistent = true
hasExtent = true

[indexes\PatientNameIndex]
class = Patient
members = name
unique = true
```

Da es keine Möglichkeit gibt, die Konfigurationsdatei automatisch zu erstellen oder zu ändern, muß sie bei jeder neuen Klasse oder bei Modifikationen von Hand aktualisiert werden. Auch sonst ist die Arbeit mit POET wenig komfortabel. Eine Integration in die verwendete Entwicklungsumgebung (JBuilder) war nicht möglich, da der Präprozessor nicht als gültiger JDK-Compiler erkannt wurde. Übersetzungen mußten mit einem eigenen make-Skript durchgeführt werden. Die Dokumentation zu POET ist zwar recht ausführlich, läßt aber Beispiele, die über triviale Funktionen hinausgehen, vermissen. Es wird weder auf die Verwendung von Java-Packages eingegangen, noch gibt es Beispiele für die Vererbung von persistenten Klassen, der Nutzung von abhängigen Objekten, dem Aufruf von Methoden persistenter Klassen oder von Beziehungen zwischen diesen.

POET benötigt den Quellcode von Klassen, um sie persistent machen zu können. Insofern sollte schon zu Projektbeginn feststehen, ob alle Komponenten selber entwickelt werden oder zumindest Open Source-Bibliotheken eingesetzt werden können, die meist auch im Quelltext vorliegen. Gefahr droht einer auf POET basierenden Applikation auch von anderer Seite. So führt eine Änderung am Datenbankschema oder an den persistenten Klassen dazu, daß eine Datenbank nicht mehr gelesen werden kann. Dies ist besonders während der Entwicklungsphase sehr unerfreulich, da es zusätzliche Populationsprogramme erforderlich macht.

Der Satz an vorhandenen Werkzeugen zum Verwalten der Datenbank kann bei weitem nicht mit dem kommerzieller RDBMS-Produkte mithalten. Ein auf POET aufbauendes Framework ist immer an die POET-Datenbank gebunden, da es keine Möglichkeit für einen Datenexport gibt. Sicherheitsfunktionen sind standardmäßig inaktiv. Es ist auch nicht möglich, sich mit den Administrationstools den Inhalt einer Datenbank anzuschauen.

POET bietet eigene Collection-Klassen für die im ODMG-Standard definierten Interfaces an. Die Kollektionen aus dem JDK können nicht verwendet werden. Dies bedeutet einen gewissen Mehraufwand, zumal sie nur persistente Klassen aufnehmen können, falls sie selber persistent gemacht werden sollen. Dies erscheint zwar natürlich, bedeutet aber in Konsequenz, daß beispielsweise `String` oder `Integer` nicht erlaubt sind. Weiterhin verringert ihre Benutzung die Portabilität des Quellcodes.

POET unterstützt die Verwendung von Extents. Darunter versteht man eine abstrakte Kollektion aller Objekte einer bestimmten Klasse. Extents sind nicht Teil des ODMG-Standards, aber nützlich, um über alle Objekte einer Klasse zu iterieren. Extents zeigen nicht nur Wurzelobjekte, sondern auch unbenannte Objekte dieser Klasse an. Zu der Extentmenge einer Klasse gehören auch alle Instanzen von Subklassen.

Für Abfragen verwendet POET die OQL, wenngleich nur eine Untermenge unterstützt wird. So sind nur SELECT-Anfragen auf Extensionen möglich. Voraussetzung ist, daß überhaupt ein Extent für die Klasse angelegt wurde (siehe obige Konfigurationsdatei). Das Ergebnis einer Abfrage ist immer ein Objekt. Dieses muß dahingehend ausgewertet werden, ob es sich um ein einzelnes Objekt oder eine Menge von Objekten handelt. Im letzteren Fall muß man der Kollektion den Typ der enthalten Elemente ausdrücklich zuweisen (`setItemType`), bevor man sie benutzen kann.

In der verwendeten Version war auch ein sogenannter *Ultra Thin Client* (UTC) vorhanden. Dabei handelt es sich um eine Pure-Java-Lösung, so daß keine betriebssystemspezifische Installation auf Seite des Clients nötig ist. Bei Verwendung des UTC wird eine 3-Schichten-Architektur vorausgesetzt. Die Objekte befinden sich dann nicht im Speicher des Client, sondern in dem eines Servers, über den entweder mit RMI oder IIOP zugegriffen werden kann. Der Client agiert nur auf lokalen Schnittstellen. In den Tests funktionierte der UTC problemlos, auch wenn das Handbuch von einem Betastadium ausging.

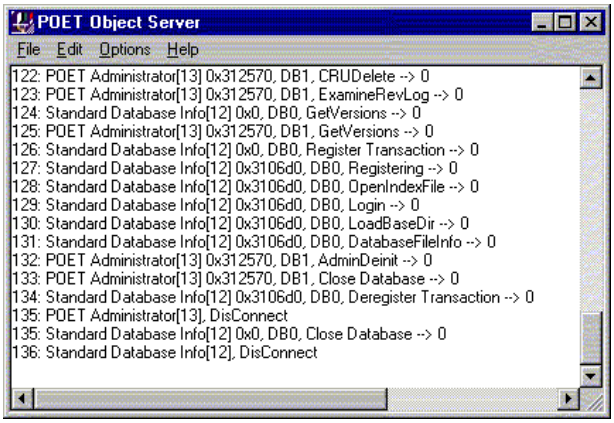


Abbildung 2: Startbildschirm des Object Servers

Eigene Leistungstest wurden mit POET nicht durchgeführt. Es seien daher die Ergebnisse aus [DOM 99] genannt.

	Java-VM	POET 5.1	Oracle 7
Instantiierung von 1000 einfachen Objekten (in ms)	456,5	5075	9545,1
Instantiierung von 1000 komplexen Objekten (in ms)	1056,7	6060,8	18 293,8
Navigationszeit für 100 Objekte mit einer Distanz von 4 (in ms)	1284,5	2344	18150,8
Zeit für eine 100 Abfragen mit 4 beteiligten Objekten (in ms)	-	7044,7	18780,5
durchschnittliche Codegröße für ein Objekt (in LoC)	415,3	482,5	753,2

Tabelle 2: Benchmarkergebnisse für OODBMS vs. RDBMS

Die Tests zeigen eine klare Überlegenheit der POET-Lösung. Allerdings können die Ergebnisse nicht auf andere Aufgabenstellungen als die im Test verwendete verallgemeinert werden. Der Testaufbau bevorzugt klar die Stärken von OODBMS wie z.B. die Objektnavigation, die in der SQL-Variante mit teuren JOINs nachgebildet wurde. Es ist aber nicht sinnvoll, bei Tabellen mit vielen Datensätzen und einer geringen Ergebnismenge einen Verbund zu bilden. Besser ist hier das sukzessive Traversieren des Objektgraphen über die Primärschlüssel der enthaltenen Objekte, was zwar mehrere Zugriffe erfordert, wobei es sich aber nur um kurze SELECT-Lesezugriffe handelt.

Nicht angesprochen wurden bei diesem Test dagegen Operationen, die eine traditionelle Stärke von RDBMS sind, wie das Auffinden von Objektmengen anhand von Parametern oder Einfügeoperationen neuer Objekte.

Während der Evaluation wurde weiterhin untersucht, ob POET als *Persistenzebene für Enterprise JavaBeans* (siehe eigenes Kapitel) verwendet werden kann. Dabei waren folgende Probleme zu lösen:

- Persistenz

Da die vom EJB-Container gesteuerte Persistenzabbildung zur Zeit nur JDBC-fähige Datenbanken unterstützt, wurde die Datenbankaufrufe direkt in die Lebenszyklusmethoden der Entity-Beans eingesetzt.

- Transaktionen

Entity Beans müssen containergesteuerte Transaktionen verwenden. Somit muß in den Lebenszyklusmethoden zusätzlich auch jeweils mit POET-Transaktionen gearbeitet werden, da der ODMG-Standard Transaktionen für die Ausführung von Objektmanipulationen zwingend vorschreibt.

- Primärschlüssel

Die Tests wurden ausschließlich mit benannten Objekten durchgeführt. Da diese einen eindeutigen Namen in der Datenbank besitzen, konnte der Primärschlüssel als Name verwendet werden.

- Statische Felder

Im ODMG-Modell gibt es eine Klasse `Database`, die Methoden zum Öffnen und Schließen von Datenbanken bereitstellt. Jede Datenbank kann nur einmal geöffnet werden, sonst wird eine Ausnahme aufgeworfen. Beim Öffnen erhält man eine statische Referenz auf die Datenbank. Da für Entity Beans keine statischen Attribute erlaubt sind, wurde die Datenbank in einer Singleton-Hilfsklasse gekapselt, die den Entity Beans eine gültige Referenz zurücklieferte.

Letztendlich konnte zwar eine lauffähige Testapplikation erzeugt werden. Leider erwies sie sich im Betrieb als instabil, d.h. es wurden Ausnahmen aufgeworfen, die nicht im Zusammenhang mit der Applikation oder dem Systemzustand der Datenbank standen. Da es auch bei POET keine Erfahrungen mit EJBs gab, wurde der Einsatz für Mobtel verworfen.

Zusammenfassend läßt sich sagen, daß es sich bei POET um ein ausgereiftes Produkt handelt, dessen Stärken allerdings deutlich bei monolithischen Anwendungen oder im Client-Server-Bereich liegen. Ein weiterer Vorteil sind die sehr geringen Hardwarevoraussetzungen und die einfache Abbildung der Java-Objekte.

Die Unterstützung von Internettechnologien oder verteilten Protokollen ist allenfalls rudimentär. Für das Mobtel-Projekt hätte ein eigener Objekt Application Server programmiert werden müssen.

Die dabei auftretenden Probleme in den Bereichen:

- Mehrnutzerzugriff, Verbindungspools
- Transaktionsverwaltung, speziell Lese- und Schreibsperren
- Sicherheitsstufen für Benutzergruppen
- Zugriff für Clients über das Internet (Port 80 Translation)

- Unterstützung für HTML-Clients
- Sitzungsmanagement
- hohe Netzwerklast durch den Datenverkehr
- Interaktion mit anderen Mobtel-Servicen
- zeitgenaue Ausführung
- Lizenzen
- Wartung der Datenbank
- Konfiguration des Application Servers

sind aber nicht zu unterschätzen. Für kleinere Gruppen oder einzelne Personen wäre diese Aufgabe nur sehr schwer zu verwirklichen gewesen. Die Implementierung der obigen Punkte ist zeitaufwendig und spricht noch nicht einmal den Problembereich des eigentlichen Projekts an. Bei einem fertigen und ausgereiften Produkt wie dem auf dem Visigenic-ORB basierenden Borland Application Server stehen diese Dienste implizit zur Verfügung. Die Architektur verspricht ferner eine höhere Stabilität und Skalierbarkeit.

Eine Einschränkung ist die dadurch erzwungene Verwendung einer Relationalen Datenbank. Die Probleme, die beim Abbilden eines Datenmodells auftreten, werden im folgenden Abschnitt erläutert.

3.7. Grundlagen einer objektrelationalen Abbildung

Prinzipiell kann die Abbildung von Objekten in Relationen auf zwei Arten geschehen. Beim *forward engineering* wird ausgehend von einem Objektmodell das Datenbankschema und die SQL-Befehle erzeugt. Im Gegensatz dazu wird beim *backward engineering* das Datenbankschema analysiert und daraus Java-Klassen generiert.

Das Backward Engineering ist aus zwei Gründen weniger vorteilhaft: Erstens wird das Klassenmodell aus dem Datenbankschema erzeugt. Dies widerspricht den Entwurfsregeln für objektorientierte Software und sollte nur Verwendung finden, wenn eine Applikation auf bestehenden Daten aufbauen muß. Zweitens weisen relationale Datenbankschemata eine geringere *semantische Ausdrucksmächtigkeit* auf. Besitzt zum Beispiel eine Tabelle `Rechnung` eine Fremdschlüsselreferenz auf `Verkäufer`, so ist nicht klar, ob es sich um eine unidirektionale Assoziation oder um eine bidirektionale Assoziation handelt. In diesem Beispiel träfe ersteres zu, da eine Rechnung zwar einen Verkäufer hat. Das Verkäuferobjekt besitzt aber im allgemeinen keine Liste aller von ihm erstellten Rechnungen, weil dies bei der Objektinstantiierung sehr aufwendig wäre. Handelt es sich aber um die Tabellen `Patient` und `Betreuer`, so ist es wahrscheinlich, daß ein Betreuer auch eine Liste der von ihm betreuten Patienten verwaltet. Um solche Zusammenhänge darstellen zu können, müssen Metadaten vorhanden sein. Im folgenden wird deshalb vom Forward Engineering ausgegangen.

Die als *impedance mismatch* bezeichnete Kluft zwischen dem Relationalen Paradigma und dem Objektparadigma wurde bereits vorgestellt. Sie äußert sich beispielsweise bei Abfragen über Objektmengen, die eine Eigenschaften besitzen. Im Objektmodell wird dabei Referenzen im Objektgraphen gefolgt, während im Relationalen Paradigma Tabellenverbunde gebildet werden.

3.7.1. Objektidentifikatoren

Objekten müssen Objekt-IDs zugewiesen werden, um sie voneinander unterscheiden zu können. Im Relationalen Modell heißen diese Schlüssel, aber auch im Objektmodell sollte ein eindeutiger Bezeichner für ein Objekt vorhanden sein. Objekt-IDs sind sehr hilfreich bei der Umsetzung des Datenmodells der Anwendung auf ein SQL-basiertes Schema. Sie werden sowohl für Beziehungen zwischen Objekten als auch zum Auffinden verwendet.

Ein Punkt ist von besonderer Bedeutung. Objekt-IDs sollten keine Bedeutung im Kontext der Anwendung besitzen. Ein Beispiel dafür ist die Verwendung des Namens als ID einer Person oder das Kaufdatum als ID für eine Rechnung. Solche IDs sind nur für kleine Applikationen geeignet, da sich Probleme ergeben, wenn die Struktur oder der Typ der ID geändert werden sollen. So könnte sich die Notwendigkeit ergeben, den Namen einer Person in die Bestandteile Vorname, Nachname und Titel aufzuspalten. Diese Aktualisierung muß dann sowohl in der Datenbanktabelle, die `Person` abbildet, geändert werden, als auch in allen Tabellen, die eine Referenz auf `Person` im Sinne von Fremdschlüsseln besitzen.

Da Objekt-IDs eindeutig sein müssen, stellt sich das Problem, diese zu erzeugen. Dazu muß zuerst festgelegt werden, in welchem Umfang die IDs eindeutig sein sollen. Möglich wären eindeutige IDs für jede Klasse, jeden Klassenbaum oder für die gesamte Applikation. Da jedoch mehrere Programme gleichzeitig auf die Datenbank zugreifen können, ist es am besten, die Objekt-IDs mit Hilfe der Datenbank zu generieren. Einige Datenbanken bieten dazu spezielle Funktionen an, die das Programm aber dann auf diesen Datenbanktyp festlegen. Eine andere Möglichkeit besteht darin, die höchste bisher vergebene ID mittels einer SQL-MAX()-Anweisungen auszulesen und zu inkrementieren. Dieses Verfahren liefert allerdings nur eindeutige Bezeichner bezogen auf diese Tabelle und erfordert zudem eine kurzzeitige Lesesperre. Besser wäre es, für diese Aufgabe eine eigene Tabelle anzulegen, die einen globalen Zähler für die Objekt-IDs enthält. Zwar muß diese Tabelle beim Lesen immer noch gesperrt werden, aber die ID ist für die gesamte Datenbank eindeutig. Eine weitere Verbesserung wäre die Zuweisung von Kontingenten an IDs an eine Applikation. Sie kann die IDs dann sukzessive aufbrauchen, und die Tabelle muß erst wieder angesprochen werden, wenn das Kontingent erschöpft ist.

3.7.2. Abbildung von Klassen in Tabellen

Klassen entsprechen von ihrer Struktur den Tabellen einer Datenbank. Im einfachsten Fall kann 1 Klasse auf 1 Tabelle abgebildet werden. Es ist aber ebenfalls möglich, eine Klasse auf mehrere Tabellen abzubilden oder mehrere Klassen auf eine Tabelle, z.B. um Redundanzen zu vermeiden.

Nur in sehr einfachen Modellen besteht die Möglichkeit, eine 1:1-Abbildung vornehmen zu können. Der Grund hierfür liegt in der fehlenden Unterstützung von Vererbung im Relationenmodell. Vererbung ist die Fähigkeit von Klassen, Attribute aus einer Superklasse zu übernehmen, ohne sie selbst deklarieren zu müssen. Die Vererbungen von Methoden ist für die Abbildung nicht von Bedeutung.

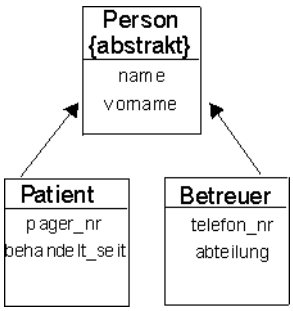


Abbildung 3: Beispiel Klassenmodell (1)

Es gibt drei Verfahren, Vererbungsbeziehungen im Relationenmodell darzustellen:

1. Eine Tabelle für die gesamte Klassenhierarchie (*typisierte Partitionierung*)

   In diesem Fall werden die Attribute aller Klassen in eine einzige Datenbanktabelle übertragen. Die Unterscheidung zwischen einzelnen Objekttypen erfolgt mittels eines zusätzlichen Attributs `objecttype`. Vorteile hierbei sind die Einfachheit der Abbildung und die schnelle Bearbeitung von Anfragen über alle Typen von Klassen. Außerdem kann der Typ eines Objekts durch Änderung von `objecttype` umgewandelt werden. Nachteilig ist, daß für jedes neue Attribut irgendeiner Klasse eine weitere Spalte in der Tabelle hinzugefügt werden muß. Ein Fehler an dieser Stelle beeinflußt das Verhalten aller Objekte, nicht nur das der Klasse, die das Attribut enthält. Außerdem wird bei dieser Lösung viel Datenbankplatz verschwendet, da nicht genutzte Zellen der Tabelle mit `NULL`-Werten aufgefüllt werden.

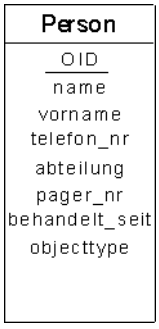
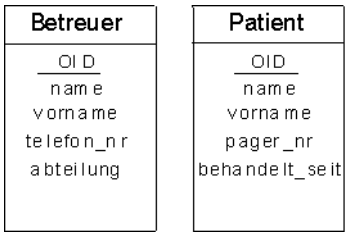


Abbildung 4: typisierte Partitionierung (1)

2. Eine Tabelle für jede nichtabstrakte Klasse (*horizontale Partitionierung*)



Jede Tabelle enthält sowohl die Attribute der assoziierten Klasse als auch alle Attribute seiner Superklassen. Anfragen über eine Klasse können schnell ausgeführt werden, da sich alle Attribute in einer Tabelle befinden. Änderungen an einer Klasse müssen aber sowohl an der entsprechenden Tabelle als auch an den Tabellen aller Unterklassen vorgenommen werden.

Abbildung 5: horizontale Partitionierung (1)

3. Eine Tabelle für jede Klasse (*vertikale Partitionierung*)

Für jede Klasse wird eine eigene Tabelle erzeugt, die nur die in der Klasse enthaltenen Attribute aufnimmt. Dieser Ansatz kommt dem OO-Konzept am nächsten. Es wird Mehrfachvererbung unterstützt und das Hinzufügen und Ändern von Klassen wirkt sich nur auf eine Tabelle aus. Es gibt jedoch auch Nachteile. Erstens gibt es relativ viele Tabellen, zusätzlich zu den Tabellen, die Beziehungen symbolisieren (siehe unten). Zweitens müssen Anfragen in praktisch jedem Fall mehrere Tabellen lesen, was die Geschwindigkeit vermindert.

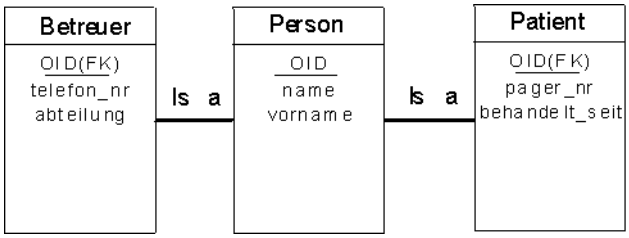


Abbildung 6: vertikale Partitionierung (1)

Zum besseren Verständnis soll verdeutlicht werden, welche Auswirkungen Änderungen am Klassenmodell auf das Datenbankschema haben. Dazu wird eine weitere Klasse hinzugefügt, welche die Hierarchie um eine Stufe verlängert.

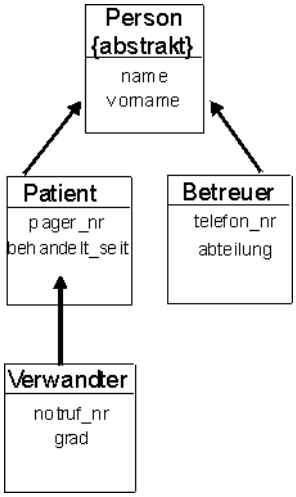


Abbildung 7: Beispiel Klassenmodell (2)

Die geringsten Änderungen mußten bei der typisierten Partitionierung vorgenommen werden. Allerdings gibt die Abbildung nicht wieder, daß die Anzahl von NULL-Werten in der Tabelle beim Eintrag von Tupeln stark ansteigen wird. Bei der horizontalen Partitionierung muß eine neue Tabelle hinzugefügt werden, die Zahl der Spaltennamen mit gleicher Bezeichnung nimmt weiter zu. Bei der vertikalen Partitionierung wird ebenfalls eine neue Tabelle hinzugefügt, die aber nur die Attribute enthält, die neu definiert wurden. Soll von dieser Klasse eine Instanz erzeugt werden, muß aus drei Tabellen gelesen werden. In der Praxis kann auch eine Kombination der Partitionierungsvarianten erfolgen.

Stärken und Schwächen finden sich zusammengefaßt in nachfolgender Tabelle.

	typisierte Part.	**horizontale Part.**	**vertikale Part.**
Einfachheit der Implementation	leicht	mittel	schwierig
Geschwindigkeit des Datenzugriffs	schnell	schnell	mittel
Probleme bei der Änderung von Klassen	sehr groß	groß	gering

Tabelle 3: Vergleich der Vererbungsansätze

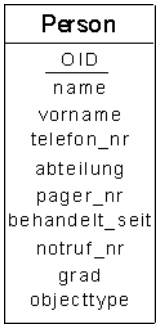


Abbildung 8: typisierte Partitionierung (2)

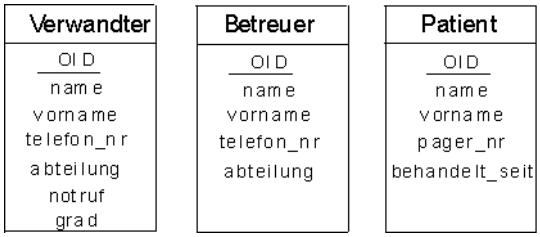


Abbildung 9: horizontale Partitionierung (2)

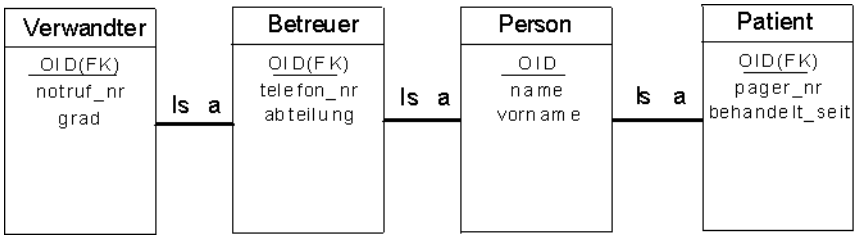


Abbildung 10: vertikale Partitionierung (2)

3.7.3. Abbildung von Attributen in Spalten

Alle primitiven Attribute einer Klasse können auf Spalten in den Tabellen einer Datenbank verteilt werden. Im Normalfall entspricht einem Attribut genau eine Spalte. Es sind allerdings Situationen denkbar, wo ein Attribut keiner Spalte entspricht, weil es nicht abgebildet werden soll (z.B. das Alter einer Person) oder kann (z.B. wenn ein Objekt Zustandsänderungen an seine Observer weiterleiten soll).

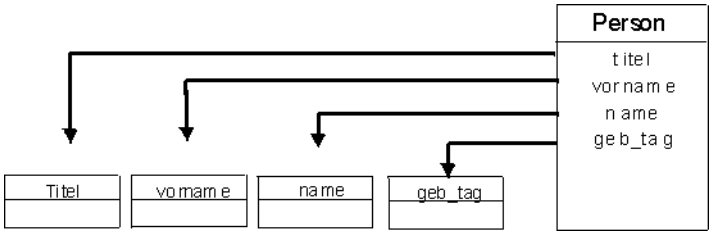
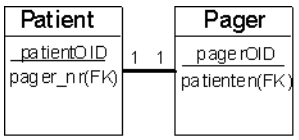


Abbildung 11: Abbildung trivialer Attribute

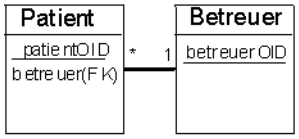
Weit problematischer ist die Abbildung von Attributen, die eine Beziehung zu einer anderen Klasse symbolisieren. Diese müssen beim Speichern der Objekte in der Datenbank nachgebildet werden, damit sie beim Auslesen wiederhergestellt werden können. Es gibt zwei Arten von Beziehungen zwischen Objekten, die unterschiedlicher Behandlung bedürfen: die Assoziation und die Aggregation (oder als zusätzliche Verschärfung die Komposition). Der Unterschied zwischen beiden ist, wie eng die Objekte aneinander gebunden sind. Eine Assoziation ist eine lose Beziehung zwischen zwei Objekten, man spricht von einer

hat-ein-Beziehung. Eine Aggregation dagegen ist eine *Teil-von-Beziehung*. Bei einer Aggregation müssen beim Laden eines Objektes auch alle Teile geladen werden. Beim Löschen werden im allgemeinen auch die Teile gelöscht. Obwohl die Überwachung und Umsetzung dieser Regeln Aufgabe der Applikation ist, kann dies auch auf Seiten der Datenbank geschehen, z.B. mittels Triggern.

Beziehungen zwischen Objekten werden im Relationenmodell auf Fremdschlüsselbasis nachgebildet. Ein Fremdschlüssel ist ein Attribut, das in einer Tabelle vorkommt und gleichzeitig in einer anderen Tabelle der oder ein Teil des Primärschlüssel ist.



Handelt es sich bei den zwei in Verbindung stehenden Objekten um eine 1:1-Beziehung, werden eingebettete Fremdschlüssel verwendet. Ebenso wird bei 1:n-Beziehungen verfahren.



Handelt es sich jedoch um eine n:m-Beziehung, so muß für diese eine weitere Tabelle geschaffen werden. Diese wird als Fremdschlüsseltabelle bezeichnet und besitzt als Attribute die Primärschlüssel der zueinander in Beziehung stehenden Klassen. Fremdschlüsseltabellen sind zwar eine Einschränkung in den Abbildungsmöglichkeiten von Beziehungen, da sie Objekte vertäuschen, die es im Datenmodell nicht geben muß. Häufig kann man die so entstandenen Tabellen aber auf natürliche Art um Attribute erweitern und dadurch ein neues Objekt erschaffen, das sich gut in die Klassenmodellierung einfügt.

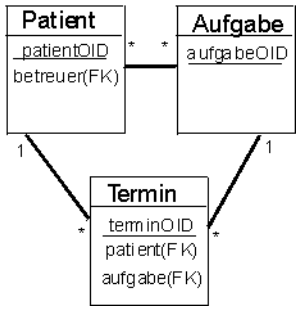


Abbildung 12: Beziehungstypen

3.7.4. Konkurrierende Zugriffe

In vielen Fällen wird ein Programm nicht nur von einem Benutzer zur selben Zeit ausgeführt. Arbeiten mehrere Nutzer synchron auf der Datenbank, besteht die Möglichkeit, daß sie auf dieselben Daten zugreifen, also mit denselben Objekten arbeiten. Um zu verhindern, daß ein Objekt bzw. seine Repräsentation in der Datenbank gleichzeitig verändert werden, muß ein geeigneter Kontrollmechanismus gefunden werden. Dieser basiert auf dem Prinzip, einzelne Tabellenzeilen für den Zugriff von anderen Transaktionen zu sperren. Es gibt zwei Strategien, mit solchen Zeilensperren zu arbeiten:

1. Pessimistisches Sperren

Bei dieser Variante der Zugriffssteuerung wird ein Objekt die gesamte Zeit, in der es sich im Hauptspeicher befindet, gesperrt. Erst wenn das Objekt wieder in die Datenbank geschrieben wurde, kann die Sperre gelöst werden. Das Verfahren garantiert, daß ein Objekt in der Datenbank nicht überschrieben wird, während es sich im Hauptspeicher befindet. Gleichzeitig können aber andere Nutzer nicht auf das Objekt zugreifen. Die Einschränkung der Parallelität kann zu drastischen Leistungsabfällen führen, so daß die Sperren hier nur solange wie unbedingt nötig gehalten werden sollten.

2. Optimistisches Sperren

In diesem Ansatz wird die Zeile der Tabelle nur für die Dauer des Zugriffs der Persistenzabbildung gesperrt. Das bedeutet, wenn ein Objekt gelesen werden soll, so wird eine Sperre auf die Zeile gesetzt. Ist das Objekt vollständig gelesen wurden, wird die Sperre aufgehoben. Soll nun ein geändertes Objekt gespeichert werden, wird wieder eine Sperre gesetzt. Dann wird verglichen, ob sich die Daten in der Zeile seit dem Lesen verändert haben. Ist dies der Fall, hat ein anderer Benutzer die Daten in der Zwischenzeit geändert. Die aktuelle Transaktion wird abgebrochen und zurückgerollt, da sie auf veralteten Ausgangsdaten basiert. Sind die Daten jedoch nicht verändert wurden, können die neuen Werte geschrieben und die Sperre gelöst werden.

Welches der beiden Verfahren eingesetzt wird, hängt auch vom Typ der Applikation ab. Das optimistische Verfahren benötigt einen gewissen Mehraufwand für den Vergleich der Tabellenzeile vor dem Schreibvorgang. Dies kann umgangen werden, wenn man in der Tabelle eine zusätzlichen Spalte einführt, die die Zeit der letzten Änderung bezeichnet. Beim Sichern muß nur verglichen werden, ob der Zeitpunkt, den das Objekt im Hauptspeicher enthält, mit dem in der Datenbank übereinstimmt. Allerdings läßt sich damit nicht verhindern, daß es bei vielen konkurrierenden Zugriffen oder bei lang andauernden Transaktionen zu einer hohen Zahl von Rollbacks kommt. Liegt aber der Anteil lesender Transaktionen höher als der von ändernden, so ist die optimistische Variante besser geeignet, weil durch parallele Lesezyklen der Gesamtdurchsatz gesteigert werden kann.

3.7.5. Einsatz von datenbankspezifischen Funktionen

Viele RDBMS bieten die Möglichkeit, Programmroutinen direkt auf dem Datenbankserver auszuführen. Diese heißen *Gespeicherte Prozeduren* (Stored Procedures). Gespeicherte Prozeduren können grundsätzlich in jeder Programmiersprache implementiert sein, auch in Java, falls das von der Datenbank unterstützt wird (z.B. bei DB2). Ihre Verwendung hat den Vorteil, daß die zu modifizierenden Daten nicht über das Netzwerk transportiert werden müssen. Außerdem sind Gespeicherte Prozeduren aufgrund ihrer Optimierung für das

darunterliegende Datenbanksystem um ein Vielfaches schneller. Allerdings gibt es auch eine Reihe von Nachteilen. Erstens sind Gespeicherte Prozeduren nicht ohne Änderungen auf andere Datenbanken übertragbar. Zweitens wird durch die lokale Ausführung der Datenbankserver stark belastet. Insgesamt läßt sich feststellen, daß der Einsatz von Gespeicherten Prozeduren im Client-Server-Modell durchaus sinnvoll ist. In einem 3-Schicht-Szenario obliegt die Überwachung der Programmlogik aber den Application Servern, so daß diese die Aufgaben der Gespeicherten Prozeduren übernehmen sollten.

Trigger sind im Prinzip auch Gespeicherte Prozeduren, die beim Eintritt gewisser Ereignisse automatisch aufgerufen werden. Trigger werden hauptsächlich zum Überwachen der referentiellen Integrität bei Modifikationen der Datenbank eingesetzt. Sie gehören zur laufenden Transaktion, wenn ein Trigger nicht erfolgreich ist, wird die Transaktion abgebrochen. Für die Verwendung von Triggern gilt im wesentlichen das vorhin gesagte. Ein Vorteil ist jedoch, daß Trigger von professionellen Daten-Abbildungsprogrammen (O2R Mapping) automatisch für die Zieldatenbank erzeugt werden.

Objektpuffer

Zugriffe auf die Datenbank sind um Potenzen langsamer als Zugriffe auf den Hauptspeicher. Aus diesem Grund ist es sinnvoll, von allen gelesenen Objekten eine logisch verbundene Kopie anzufertigen und Änderungen zunächst nur auf der Kopie vorzunehmen. Erst wenn die Transaktion mit einem Commit abgeschlossen wurde, wird die Datenbank aktualisiert. Dieses Modell läßt sich noch erweitern. Dazu führt man ein Wahrheitsfeld `dirty` ein, welches beim Erzeugen der Hauptspeicherkopie auf `false` gesetzt wird. Alle Operationen, die Attribute des Objekts verändern, setzen das Flag auf `true`. Somit kann festgestellt werden, ob ein Objekt überhaupt aktualisiert werden muß. Auch das Navigieren zwischen Objekten mit Beziehungen kann durch eine Hauptspeicherkopie beschleunigt werden, da Datenbankaufrufe nur für noch nicht im Speicher befindliche Objekte nötig sind. Dabei kommt dem Modell zu Gute, daß es zu jedem Objekt einen Objektidentifikator gibt. Anhand dieser eindeutigen ID kann die Identität von Objekten sehr leicht festgestellt werden. Wird ein Objekt von einer Applikation angefordert, läßt sich leicht ermitteln, ob das Objekt schon im Puffer vorliegt. Es kann also nicht vorkommen, daß zwei Pufferobjekte für dasselbe Datenbanktupel existieren.

Zusammenfassung Kapitel 3

Im vorliegenden Kapitel wurden die prinzipiellen Zugriffsmöglichkeiten auf Datenbanken aus Java erörtert. Es zeigte sich, daß in Bezug auf Mobtel objektorientierte Datenbanken wie POET trotz eines besseren theoretischen Ansatzes in der Praxis relationalen Datenbanken unterlegen sind. Allerdings verlangen letztere eine gewissenhafte Planung des Datenbankdesigns, da sich bei der Abbildung der Objekte eine Reihe von Problemen ergeben.

4. Enterprise JavaBeans

Mobtel wurde, wie bereits dargelegt, von Beginn an modular konzipiert. Es soll flexibel in Bezug auf die eingesetzte Hard- und Software sein und sich leicht um neue Funktionen erweitern lassen. Außerdem soll der Zugriff auf Mobtel von einer möglichst großen Zahl von Clienttechnologien möglich sein, um später auch andere Akteure einbinden zu können, die sich an verschiedenen geographischen Standpunkten aufhalten.

Aus diesem Grund wurde Mobtel als 3-Schicht-Architektur realisiert. Dabei sind Client und Server streng getrennt. Der Client arbeitet dabei auf Stellvertreterobjekten, die Funktionsaufrufe über ein Kommunikationsprotokoll an den Server weiterleiten und dessen Antwort zurückgeben. Die eigentliche Abarbeitung findet auf dem Server statt, ohne daß dies für den Client ersichtlich ist. Auf dem Client muß im Fall des Vorhandensein eines Web-Browsers mit Internetzugang keine Software installiert und gewartet werden.

An dieser Stelle werden zuerst die Grundlagen für Entfernte Prozeduraufrufe behandelt. Alle modernen Kommunikationsprotokolle bauen aber außerdem auf einem Komponentenmodell auf, was viele zusätzliche Vorteile bietet. Diese Modelle spezifizieren modulare Softwareeinheiten, die in einer Ablaufumgebung, einem Container, ausgeführt werden. Sie besitzen einen Lebenszyklus und können auf eine Reihe von Diensten zurückgreifen, die häufig benötigte Funktionen bereitstellen. Es werden die derzeitig relevanten Protokolle vorgestellt und Gemeinsamkeiten und Unterschiede aufgezeigt. Die Java 2 Enterprise Edition, auf der das Mobtel-Projekt aufbaut, wird genauer erläutert.

In einem weiteren Abschnitt werden Möglichkeiten der Persistenzabbildung von EJB diskutiert. Ferner werden Einschränkung des Java-Sprachumfanges benannt, die bei der Verwendung von EJB beachtet werden müssen. Der letzte Abschnitt befaßt sich mit dem Begriff und den Aufgaben von Application Server als Wirtsinstanz für EJB.

4.1. Einführung in die Verteilte Programmierung

4.1.1. Innerprozeßkommunikation

Heutige Betriebssysteme sind multitaskingfähig, das heißt, sie können mehrere Programme quasi gleichzeitig ausführen, indem sie die Prozessorzeit mittels eines geeigneten Algorithmus auf die einzelnen Anwendungsprozesse verteilen. Der Prozeß erhält dazu vom Betriebssystem einen definierten Speicherbereich zugewiesen und kann Methodenaufrufe zwischen Datenobjekten durch Sprung an die jeweilige Speicheradresse ausführen. Andererseits schirmt das Betriebssystem alle Prozesse untereinander ab, so daß kein Prozeß ei-

nen Aufruf außerhalb des eigenen Prozeßraumes tätigen kann. Damit wird die Stabilität des Gesamtsystems und die Sicherheit der Prozeßdaten gewährleistet und fehlerhafte oder schädliche Wechselwirkungen vermieden.

Ein Javaprogramm wird in einer Ablaufumgebung, der Virtuellen Maschine, ausgeführt. Diese stellt innerhalb des Prozesses weitere Dienste bereit, die auf die Datenobjekte zugreifen. Dies sind beispielsweise die Garbage Collection zum Entfernen unreferenzierter Objekte aus dem Speicher und die Bytecode-Überprüfung, die die Ausführung unzulässiger Java-Befehle verhindert. Damit diese Dienste innerhalb eines Prozesses ausgeführt werden können, startet die VM verschiedene Threads. Ein Thread ist ein Konzept zur Parallelisierung von Abläufen innerhalb von Prozessen. Der Vorteil von Threads liegt in der Möglichkeit, inhaltlich gleiche Aufgaben ressourcenschonend und effizient auf gemeinsamen Daten auszuführen, ohne dafür jeweils einen neuen Prozeß erzeugen zu müssen.

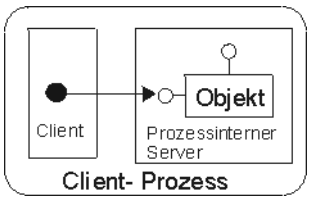


Abbildung 13: Innerprozeßaufruf

An dieser Stelle muß auf die Unterschiede zwischen Java-Threads und System-Threads verwiesen werden. System-Threads sind eine ferngesteuerte Einheit des Betriebssystems. Java-Threads sind eine Abstraktion der Programmiersprache für System-Threads. Erst wenn die Abarbeitung eines Java-Threads durch den Aufruf der Methode `start()` begonnen wurde, wird ein System-Thread erzeugt. Der Java-Thread arbeitet innerhalb der VM mit einem eigenen Stack Bytecode ab, der System-Thread arbeitet mit einem eigenen Stack VM-Code ab. Für die theoretische Betrachtung ist es ausreichend, den System-Thread zu verfolgen.

Der Aufruf lokaler Methoden innerhalb eines Prozesses ist die schnellste Kommunikationsmöglichkeit. Der Einsatz von Threads erfordert zwar einen gewissen Verwaltungsaufwand zur Synchronisation, bietet aber die Möglichkeit, durch die Vergabe unterschiedlicher Prioritäten Einfluß auf die Ausführungsgeschwindigkeit zu auszuüben.

4.1.2. Lokale Interprozeßkommunikation

Sollen verschiedene Prozesse auf einem System miteinander kommunizieren, so wird im allgemeinen mit Stellvertreterobjekten gearbeitet. Demjenigen Prozeß, der auf eine anderes Objekt außerhalb seines Adreßraumes zugreifen will, wird ein Proxy-Objekt (Stub) zur Seite gestellt. Dieses lokale Stellvertreterobjekt bietet genau dieselbe Schnittstelle (Metho-

den) wie das entfernte Objekt an. Ruft der Prozeß eine Methode des Stellvertreterobjekts auf, verpackt dieses die Methodenparameter (*Marshaling*) und überträgt sie mittels Interprozeßkommunikation an den Prozeß, der das eigentliche Objekt enthält. Dort werden die Daten von einem Skeleton-Objekt ausgepackt (*Unmarshaling*) und schließlich wird die gewünschte Methode ausgeführt. Das Ergebnis der Operation wird mittels derselben Technik dem Client zurückgegeben. Für diesen unterscheidet sich der Aufruf nicht von einem Innerprozeßaufruf. Allerdings dauert diese Prozedur erheblich länger, da mehr Objekte an der Verarbeitung beteiligt sind und zusätzlich die Prozeßgrenzen überschritten werden müssen. Des weiteren müssen bestimmte Bedingungen erfüllt sein, um die Funktionsparameter, welche ja ebenfalls Objekte sind, von Serverprozeß interpretieren lassen zu können. Entweder verfügt der Server über die Möglichkeit, eigene Objekte vom Typ der Parameter zu instantiieren, was einen Zugriff auf die kompilierten Dateien erfordert, oder die Parameter werden auf geeignete Weise beschrieben, so daß sie vom Serverprozeß zusammengesetzt werden können.

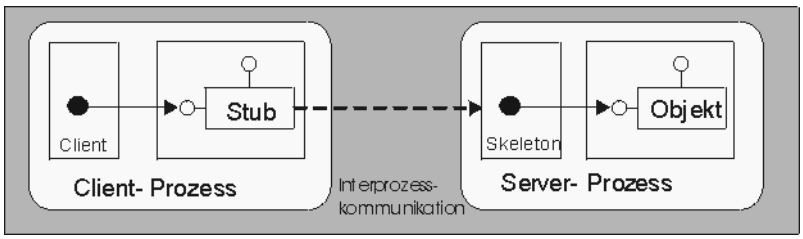


Abbildung 14: Interprozeßaufruf

4.1.3. Entfernte Interprozeßkommunikation

Befindet sich das Zielobjekt in einem anderen Prozeß auf einem anderen Rechner, wird ebenfalls ein Stub-Objekt auf Clientseite angelegt, das den Methodenaufruf weiter sendet. Allerdings muß in diesem Fall ein entfernter Prozeduraufruf (*Remote Procedure Call*) abgesetzt werden, den das Skeleton-Objekt entgegennimmt. Stub und Skeleton werden beim Übersetzen des Quellcodes automatisch mit Hilfe der Schnittstellenbeschreibungsdatei (IDL) erzeugt. Auch hier sind die beiden Proxy-Objekte nicht nur für die Methodenaufrufe zuständig, sondern auch für das Verpacken der Parameter in ein Standardformat. Die Kommunikation über RPC ist die langsamste der hier beschrieben Verfahren, da zusätzlich der Overhead für die darunterliegenden Netzwerkschichten anfällt.

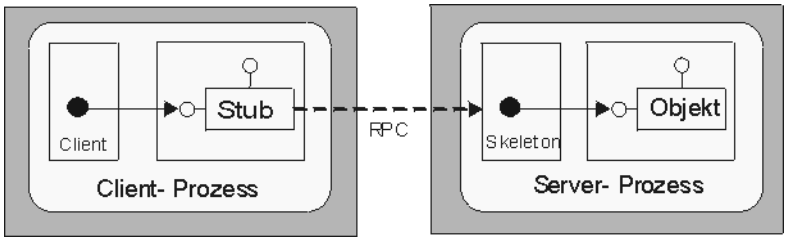


Abbildung 15: Entfernter Prozeduraufruf

RPC stellt die Basis für alle höheren verteilten Kommunikationsprotokolle dar.

4.2. Gebräuchliche Komponentenmodelle und -protokolle

Waren vor wenigen Jahren noch Client-Server-Systeme populär, so hat sich mit der Verbreitung des Internets eine neue Architektur durchzusetzen begonnen: *Mehrschichtsysteme*. Die ersten Mehrschichtsysteme waren Webserver, die HTML-Seiten dynamisch generierten und für diesen Zweck auf eine Datenbank zugriffen. Dabei wurde schnell klar, daß Lösungen für infrastrukturelle Probleme gefunden werden mußten, die im Client-Server-Modell in dieser Form nicht vorhanden waren.

Zum einen stammten die Clients aus geographisch weit verteilten Netzwerken und verfügten über eine völlig heterogene Hard- und Softwareausstattung. Sie waren in ihrer Mehrzahl über Telefonleitungen mit dem Server verbunden, was nur eine sehr geringe und unzuverlässige Datenübertragung ermöglichte. Zum anderen wurde der Webserver schon durch die große Zahl simultaner Zugriffe belastet. Das Ausführen von serverseitigen Anwendungen wie CGI benötigte zusätzliche Ressourcen. Für jede Anfrage wurde ein neuer Prozeß gestartet, eine Datenbankverbindung aufgebaut, Daten gelesen, die Verbindung geschlossen, die Ergebnisse an den Client zurückgegeben und der Prozeß beendet.

Schon bald wurde deutlich, daß sich komplexere Anwendungen mit dieser Architektur nicht verwirklichen lassen würden. Die Benutzer müssen eine große Zahl von verschiedenen Aktionen ausführen können, wobei es unter ihnen verschiedene Klassen mit unterschiedlichen Rechten geben kann. Die gesamte Programmlogik vorhandener Anwendung ließ sich nicht auf CGI portieren. Datenbanken waren auf Mehrbenutzerbetrieb ausgelegt, jetzt griff nur noch der Webserver zu, wobei dieser die Zahl der Zugriffe nicht bewältigen konnte. Der Engpaß lag augenscheinlich auf der mittleren Ebene.

Dies führte zu einer Trennung in verschiedene Ebenen. So gibt es eine Präsentationsebene, in der Browser oder GUI-Client arbeiten. Weiterhin gibt es eine (optionale) Web-Ebene, mit der der HTML-Client interagiert und die die graphische Darstellung übernimmt. In ihr laufen der Webserver und zusätzliche Erweiterungen wie Servlets oder Active Server Pages.

Von dort erfolgt eine Weiterleitung zur Ebene mit der Geschäftslogik, die sowohl die Anwendungsfälle bearbeitet als auch die Daten in die letzte Ebene speichert, die Persistenzebene.

Die Erfolge in der Objektorientierung und die geforderte Multibenutzerfähigkeit führten zur Entwicklung verschiedener Komponentenmodelle. *Komponenten* sind Objekte, die abgeschlossene Funktionen realisieren und ihre Leistungen über genau definierte Schnittstellen anderen Komponenten anbieten. Ziel der Verwendung von Komponenten ist primär eine Entkopplung der verschiedenen Aufgaben, die Klassen in herkömmlichen Programmiermodellen übernommen haben. Ein Hauptziel der objektorientierten Softwareentwicklung war die Wiederverwertbarkeit von Klassen. Dieses Ziel ist jedoch nicht in dem erhofften Ausmaß erreicht wurden. Die Gründe dafür liegen im besonderen darin, daß die Klassen zusätzlich zu ihren eigentlichen Aufgaben auch andere Funktionen integrieren mußten, beispielsweise eine Persistenzabbildung, eine Transaktionsverwaltung, Sicherheitsaspekte und Protokollierung. Die Chance, daß es einen weiteren Fall gibt, bei dem genau dieselben Anforderungen an die Klasse gestellt werden, ist gering. Die Adaption auf z.B. ein anderes Persistenzmedium kostet aber viel Zeit, da die Aufgaben in der Klasse nicht klar getrennt sind.

Komponenten sollen dagegen nur die Geschäftslogik implementieren. Alle anderen benötigten Dienste werden ihnen bereitgestellt. Zur Kommunikation untereinander und mit diesen Diensten wurden Protokolle entwickelt, die den Ort, auf dem sich eine Komponente befindet, komplett verbergen, also *verteilungstransparent* sind.

Ein zentrales Konzept der Komponententechnologie ist der *Container*. Ein Container bildet eine Schutzhülle um die Komponenten und bietet eine Reihe von Programmierschnittstellen sowohl für die Komponenten als auch für die Clients an. Weiterhin steuert er den Lebenszyklus. Der Container kreiert, aktiviert, passiviert und löscht Komponenten; diese werden über Rückrufschnittstellen über die entsprechenden Ereignisse benachrichtigt. Des weiteren steuert der Container die Inanspruchnahme von Diensten durch die Komponenten. Dazu stellt er sowohl einen programmatischen Zugriff auf die Dienste zur Verfügung als auch eine deskriptive Konfiguration mittels Textdateien, z.B. unter Verwendung von XML. Beispiele für letzteres sind die Vorgabe von Transaktionsverhalten und Sicherheitsrestriktionen für einzelne Komponenten.

Ein weiteres Merkmal von verteilten Komponententechnologien ist *Interception*. Ein Client greift niemals direkt auf eine Komponente zu. Alle Aufrufe werden vom Container abgefangen. Der kann auf die Art der Anfrage geeignet reagieren und sie eventuell an eine Instanz weiterleiten. Einige Funktionen wie das Erzeugen neuer Komponenten werden vollständig vom Container übernommen. Dadurch ist es ihm z.B. möglich, während des Startvorgangs Pools von Instanzen eines Komponententyps anzulegen und sie erst bei einem tatsächlichen Zugriff mit konkreten Werten zu initialisieren.

Komponentenmodelle, die verschiedene Programmiersprachen unterstützen, realisieren den Austausch der Komponentenschnittstellen durch die Definition in IDL (*Interface Definition Language*). Generatoren erzeugen den nötigen Programmcode zur Verständigung zwischen den Komponenten, dem Container und den Clients.

Einige Komponenten müssen ihren Zustand auch über das Ende ihres Lebenszykluses hinaus persistent halten. Dies kann entweder in der Verantwortung der Komponente selbst liegen (*selbstverwaltete Persistenz*) oder vom Container übernommen werden (*containerverwaltete Persistenz*). Im ersten Fall wird die Komponente über Rückrufschnittstellen benachrichtigt und kann die Abbildung selbst verwalten, im letzteren Fall ist die Persistenzvorschrift dem Container beim Aufspielen übergeben worden.

In beinahe allen Modellen existieren unterschiedliche Typen von Komponenten. Manche von ihnen sind an einen Client gebunden, andere sind zustandslos. Häufig gibt es auch persistente Typen. Die Installation von Komponenten auf dem Container erfolgt im allgemeinen in binärem Zustand. Erweiterte Informationen kann der Container aus einer mitgelieferten Konfigurationsdatei entnehmen.

4.2.1. Enterprise JavaBeans

Enterprise JavaBeans sind eine 1998 erstmals vorgestellte Spezifikation von SUN Microsystems. EJB sind der zentrale Bestandteil der Java 2 Enterprise Edition (J2EE). Sie werden ausschließlich in Java programmiert. Ihr Name lehnt sich an die clientseitigen JavaBeans an, funktional bestehen aber praktisch keine Übereinstimmungen. Jedes EJB besitzt ein *Home-* und ein *Remote-Interface*. Das Home-Interface dient zum Aufruf von Factory-Methoden, die vom Container ausgeführt werden. Dazu zählen das Erschaffen, Finden und Löschen. Das Remote-Interface bietet dagegen die eigentlichen Geschäftsmethoden an. Diese Aufrufe werden immer von einer Beaninstanz ausgeführt. Ein Client erzeugt zuerst einen neuen Kontext des Verzeichnisdienstes JNDI. JNDI ist eine herstellerneutrale API, die mit jeder Art von Namens- oder Verzeichnisdienst genutzt werden kann, z.B. auch LDAP. Die Definition der Interfaces erfolgt nicht über IDL, sondern nach den Konventionen von RMI.

Ein EJB besteht außer den Interfaces noch aus der eigentlichen Beanklasse und dem *Deployment Descriptor*, einer XML-Konfigurationsdatei. Diese enthält sämtliche Metainformationen über das Bean hinsichtlich des Transaktionsverhaltens, der Persistenzabbildung oder der Sicherheitsanforderungen. Beim Aufspielen von EJBs auf einen Container wird aus dem Home-Interface ein `EJBHome`-Proxy und aus dem Remote-Interface ein `EJBObject`-Proxy generiert, die die jeweiligen Schnittstellen implementieren. Es existiert ein EJBHome für jedes im Container vorhandene Bean und ein EJBObject für jede Referenz eines Clients auf ein EJB. Die Proxy-Objekte werden vom Container verwendet, um den Clients Zugriff auf das EJB zu gewähren. Für den Client geschieht dies transparent, er ruft weiter-

hin Home- und Remote-Interface auf. Die Proxy-Objekte fangen dessen Anfragen ab, leiten sie an den Container und dieser sie eventuell an eine konkrete Bean-Instanz weiter. Wird vom Container eine neue Instanz erzeugt, übergibt er ihr ein Kontext-Objekt, mit dem ein EJB beispielsweise Sicherheitsinformationen über den Aufrufer erhalten kann, die im Deployment Deskriptor festgelegt worden sind.

Die EJB-Spezifikation unterscheidet zwischen folgenden Komponententypen:

1. Stateless Session Beans

Hier handelt es sich um den einfachsten Typ. Stateless Session Beans sind zustandslose Komponenten. Sie können von mehreren Clients benutzt werden, da sie nach dem Ende eines Aufrufs ihre Attribute zurücksetzen. Stateless Session Beans definieren eine Reihe von Methoden, die der Container im Fall einer Änderung ihres Lebenszyklus' aufruft:

```
public void ejbCreate();
public void ejbRemove();
public void setSessionContext(SessionContext sc);
```

Wenn die Komponente erzeugt werden soll, ruft er ejbCreate() auf, wird das Bean entfernt, ejbRemove() und setSessionContext() bezeichnet die oben erwähnte Kontextzuweisung.

2. Stateful Session Beans

Im Gegensatz dazu sind Stateful Session Beans zustandsbehaftete Komponenten. Sie unterhalten eine spezielle Beziehung zu dem Client, der sie erschaffen hat, den sogenannten *conversational state*. Stateful Session Beans können, auch während sie von einem Client referenziert sind, ausgelagert werden. Um darauf reagieren zu können, definieren sie zusätzlich zu den oben erwähnten Methoden noch folgende:

```
public void ejbActivate();
public void ejbPassivate();
```

Sie werden bei der Aktivierung und Passivierung der Komponente aufgerufen. Obwohl der Container die Attribute bei der Reaktivierung automatisch mit den gültigen Werten versieht, sind die Methoden nützlich, wenn beispielsweise auf eine Datenbank zugegriffen werden soll. Ferner können Stateful Session Beans das Interface `SessionSynchronization` implementieren, das ihnen den Zustand der aktuelle Transaktion bekanntmacht.

3. Entity Beans

Entity Beans sind persistente Komponenten. Sie bilden Geschäftsobjekte ab, die ihren Zustand behalten müssen, auch wenn sie nicht referenziert sind. Jedes Entity Bean besitzt

einen Primärschlüssel, anhand welchem es eindeutig identifiziert werden kann. Entity Beans besitzen folgende Rückrufmethoden:

```
public PKClass ejbCreate(...);
public void ejbPostCreate(...);
public void ejbRemove();
public void ejbFindByPrimaryKey(PKClass pk);
public void ejbActivate();
public void ejbPassivate();
public void ejbLoad();
public void ejbStore();
public void setEntityContext(EntityContext context);
public void unsetEntityContext();
```

Entity Beans sind eng mit Einträgen in einer Datenbank oder einem anderen Speichermedium verbunden. Es gibt zwei verschiedene Methoden der Persistenzabbildung: die Komponenten-verwaltete (BMP) und die Container-verwaltete Persistenz (CMP). Im ersten Fall ist das Bean selbst für eine Zustandssicherung verantwortlich und muß die ejbLoad() und ejbStore()-Methoden sinnvoll ausfüllen. Im zweiten Fall wird die Abbildung während des Aufspielen des Beans durch den Deployment Deskriptor festgelegt. Es können mehrere Clients auf ein Entity Bean zugreifen, die Integrität wird durch den Transaktionsdienst gewährt.

Im Unterschied zu Session Beans gibt es hier noch eine weitere Art von Methoden, die im Home-Interface definiert werden, die Finder-Methoden. Sie werden zur Suche nach einem oder einer Menge von Entity Beans verwendet und vom Container ausgeführt. Benutzt das Bean CMP, so wird die Methode deskriptiv erklärt, nutzt sie BMP, muß die Methode implementiert werden.

4. Message Driven Beans

Dieser Beantyp wird erst in der neuen Version 2.0 definiert sein. Message Driven Beans empfangen und verarbeiten asynchrone Nachrichten über den Nachrichtendienst JMS. Auf diese Art ist sowohl ein Rückruf zu Clients als auch die Entwicklung lose gekoppelter Netze und die Integration anderer nachrichtenorientierter Systeme möglich. Message Driven Beans sind zustandlose Komponenten und besitzen weder Home- noch Remote-Interface, da sie nicht direkt von Clients angesprochen werden. Als Besonderheit besitzen sie die Methode onMessage(), die beim Erhalt einer Nachricht aufgerufen wird.

```
public void ejbCreate();
public void ejbRemove();
public void onMessage(Message inMessage);
public void setMessageDrivenContext(MessageDrivenContext mdc);
```

Message Driven Beans werden voraussichtlich Punkt-zu-Punkt und Verleger-Abonnent Beziehungen unterstützen.

4.2.2. CORBA

Seit der Version 3.0 des CORBA-Standard existiert auch hier eine Beschreibung für ein Komponentenmodell, das *CORBA Component Model* (CCM). Dieses schließt die EJB-Spezifikation fast vollständig ein, so daß EJBs auch als Corba-Komponenten (*CORC*) betrachtet werden können. Die Schnittstellen werden in Component IDL definiert, so daß Implementierungen mit mehreren Programmiersprachen möglich sind.

CORC besitzen eine größere Anzahl von Schnittstellen, sogenannten *Ports*, die in verschiedene Einsatzzwecke unterteilt sind:

- die *Äquivalenzschnittstelle*, die jede Komponente besitzen muß und die zur Beschreibung und zur Navigation zu anderen Schnittstellen dient
- *Facettes* entsprechen am ehesten den Remote-Interfaces der EJBs
- *Home* ermöglicht ebenfalls die Erschaffung oder das Auffinden von Instanzen aus ihrer Gesamtmenge
- *Receptacles* ermöglichen die Verwendung externer Schnittstellen in der Komponente
- *Emitter* und *Publisher* senden Ereignisse an einen oder mehrere Empfänger
- *Subscriber* läßt die Komponente Ereignisse empfangen

Die konkrete Implementierung einer Komponente heißt *Servant*. Ein Aufruf eines Servants wird durch den POA (*Portable Object Adapter*) abgefangen. Der POA hält eine Referenzliste, mit der er Aufrufe von Objekten an Servants weiterleiten kann. Dabei können mit Hilfe von Richtlinien Einstellungen für die Servants festgelegt werden, z.B. ob es sich um ein persistentes Objekt handeln soll. Auch bei CCM gibt es Rückrufschnittstellen und Kontextobjekte.

CORBA stellt neben den im vorigen Abschnitt genannten Komponententypen noch eine Reihe weiterer zur Verfügung: *Servicekomponenten* und *Prozeßkomponenten*. Erstere sind zustandslos und nicht persistent, letztere sind persistent und einem Client zugeordnet. Die Persistenzabbildung wird über den *Persistent State Service* realisiert. Ein *Storage-Objekt* ist eine abstrakte Definition für ein Speicherziel. Ob sich dahinter eine Relationale Datenbank oder ein anderes Medium verbirgt, ist der Komponente verborgen. Ein Generator erzeugt die eigentliche Zugriffsfunktionalität. Eine CORBA-Komponente kann ihren Zustand in einzelne Segmente zerlegen und diese separat auslagern. Analog zu EJBs kann die Persistenzabbildung von der Komponente oder dem Container übernommen werden.

CORBA-Komponenten werden ebenfalls in Form von Archivdateien aufgespielt und enthalten auch einen XML-Deskriptor. Weiterhin baut das CCM natürlich auf den bewährten CORBA-Diensten auf (siehe vergleichende Tabelle am Abschnittsende).

4.2.3. COM

Das *Component Object Modell* (COM) von Microsoft ist die älteste der Komponentenarchitekturen. Sie ist nur für Windows-Betriebssysteme verfügbar. Ursprünglich handelte es sich bei COM um eine rechnerlokale Spezifikation, erst die Erweiterung DCOM (*Distributed COM*) machte Kommunikation über Rechnergrenzen möglich. Später wurde die Architektur um den Microsoft Transaction Server (MTS) erweitert. Dabei handelt es sich um eine transaktionsorientierte Middleware, die einerseits die Komponenten auf einer modernen Mittelschicht vereint und andererseits ihren Zustand in verfügbare Ressourcen wie den Shared Memory Dispenser und den SQL-Server abbildet. In Windows 2000 verschmelzen COM, DCOM und MTS zur Bezeichnung COM+.

Auch COM-Komponenten können in mehreren Schnittstellen Funktionen für Clients anbieten. Alle erben dabei von der Basisschnittstelle `IUnknown`, die sämtliche Lebenszyklusmethoden definiert und die Methode `QueryInterface` zum Aufruf der verfügbaren Schnittstellen für diese Komponente anbietet.

COM-Komponenten liegen als dynamische Bibliothek (DLL) oder als ausführbares Programm (EXE) vor. Ein Home-Interface wie bei EJBs gibt es nicht, jede DLL oder EXE stellt Factories zum Instantiieren von Objekten bereit. Alle Schnittstellen und Komponenten besitzen eindeutigen Identifikatoren. Bei Aktivierung sorgt das Laufzeitsystem für eine Verbindung zu den Factories, entweder durch Registrierungsinformationen oder durch Nutzung des *Active Directory Service Interface* (ADSI), das eine ähnliche Funktionalität wie JNDI bietet. Der Workflow bei der Abarbeitung eines Aufruf durch mehrere COM+-Komponenten wie als Aktivität bezeichnet. Auch er besitzt eine eindeutige ID (Casuality ID).

COM+-Komponenten liegen wie EJBs in binärer Form vor. Sie werden zu Paketen (DLLs) zusammengefaßt und über die Microsoft Management Konsole auf den Server aufgespielt. Alle Konfigurationsinformationen werden dabei im COM+-Katalog abgelegt. COM+-Komponenten laufen ebenfalls in einem Container (DLLHOST.EXE) ab. Der Container fängt wie bei den anderen Architekturen alle Client-Aufrufe ab und kann dadurch Objekte bei Bedarf initialisieren oder auf Vorrat halten (Pools). Eine ursprünglich für Windows 2000 vorgesehen Lösung einer In-Memory-Datenbank zum Caching von Objektinstanzen wurde aufgrund diverser Mängel zurückgezogen. Ähnliches ist dem geplanten Komponenten-Lastverteilungsmechanismus (component load balancing) widerfahren.

Dem Remote-Interface ähnlich ist der *Kontext-Wrapper*, der dieselben Schnittstellen wie die Komponente anbietet und als Interception-Proxy dient. Auch COM+-Komponenten können auf den sie übergebenden Container zugreifen, um Sicherheitsrestriktionen zu erfragen oder Transaktionen zu steuern. Implementiert eine COM+-Komponente das Interface `IObjectControl`, wird es über Aktivierung und Passivierung benachrichtigt.

COM+ stellt außerdem zwei neue Dienste für Komponenten zur Verfügung. Der *Microsoft Message Queue Server* (MSMQ) ermöglicht das asynchrone Aufrufen von Methoden durch Speicherung und Übermittlung der Aufrufe. Er wird durch *Queued Components* genutzt. Zusätzlich gibt es bei COM+ einen echten Ereignisdienst, der eine verteilte Übermittlung zwischen Lieferanten und Abonnenten verwirklicht, die Ereignisse zusätzlich filtern können.

4.2.4. Schlußbetrachtung

Die hier vorgestellten Komponentenarchitekturen weisen mehr substantielle Gemeinsamkeiten als Unterschiede auf. Das Prinzip

- des Komponenten-umgebenden Containers, der Aufrufe abfängt
- der Bereitstellung von Schnittstellen für Operationen auf entfernten Objekten
- der Konfigurationsmöglichkeit von Komponenten beim Aufspielen auf den Server
- der Unterstützung der Komponenten durch die Bereitstellung von Basisdiensten

ist in allen Modellen vorhanden. Unterschiede bestehen im Reifegrad und in der Unterstützung von Programmiersprachen und Betriebssystemen. Zusammenfassen läßt sich dies in folgender Tabelle [STA 00]:

	EJB	**CCM**	**COM+**
Basisarchitektur	Container & Interception	Container & Interception	Container & Interception
Methoden zum Finden/ Kreieren	Finder & Factory	Finder & Factory	Factory
Schnittstellen pro Komponente	Ein Interface	Mehrere Interfaces	Mehrere Interfaces
Aktivierung durch	Container	Container & POA	Container & Laufzeit
Komponentenarten	Session, Entity, (Message)	Session, Service, Prozeß, Entity	Stateless Session
Kommunikationsprotokoll	IIOP/ RMI	IIOP	DCE/ SOAP
Persistenz	Serialization	PSS	Datenbank
Ereignisdienst	nicht unterstützt	Notification Dienst	COM+ Events
Transaktionsmonitor	JTS/ OTS	OTS	MTS
Sicherheitssupport	Java Security Package	CORBA Security	COM+ Security
Asynchrone Kommunikation	JMS	AMI/ TII	Queued Components
Verzeichnisdienst	JNDI	CosNaming	ADSI
Skalierbarkeit	+	++	o

Verfügbare Interoperabilitätsprotokolle	RMI-IIOP	RMI-IIOP COM	IIOP
Integrationsmöglichkeit mit	CCM	EJB	nicht vorhanden
Installation und Konfiguration	Abhängig vom Hersteller	Installations-Deployment-Prozeß	COM+ Explorer Registry
Metadaten	XML	XML	proprietär
Ad-hoc-Networking	Jini	nicht vorhanden	UPnP

Tabelle 4: Vergleich der Komponententechnologien

CORC ist das universellste der vorgestellten Modelle, aber deswegen auch das komplizierteste. Es unterstützt alle Programmiersprachen und Betriebssysteme. CORBA hat sich seit langem im Einsatz als stabiles und leistungsfähiges System bewährt. Allerdings sind zur Zeit noch keine Produkte auf Basis von CORC verfügbar

COM+ hat den Vorteil, daß es integraler Bestandteil von Windows 2000 ist. Die tiefe Einbettung in das Betriebssystem sorgt für eine sehr schnelle Architektur. Die Kosten für das Gesamtsystem sind durch die Nutzung billiger Industrie-PCs und die Quersubventionierung der Middlewareplattform durch alle Käufer von Windows 2000 Server sehr gering. Allerdings besitzt Window bekannte Schwächen im Bereich der Stabilität. Die Skalierbarkeit wird durch fehlendes Load-Balancing auf die maximal mögliche Zahl von 32 Prozessoren begrenzt. Eine Entscheidung für Windows bedeutet weiterhin die bedingungslose Unterordnung unter die Evolutionsstrategie von Microsoft, da dies der einzige Anbieter von COM-Implementierungen bleiben wird. COM+ eignet sich für kleine, abgegrenzte Projekte, für die nicht auf Systeme anderer Hersteller zugegriffen werden muß.

EJB sind zur Zeit die erste Wahl, wenn es um eine Verteilte Komponentenarchitektur in heterogenen, erweiterbaren Systemen geht. In den letzten Jahren hat sich die EJB-Spezifikation zu einem Punkt entwickelt, wo nicht mehr zu befürchten ist, daß EJB in Zukunft nicht weiter unterstützt werden. Die große Anzahl an verfügbaren Servern läßt sowohl eine auf gewünschten Spezialeigenschaften begründete Auswahl als auch eine Wechseloption zu.

4.3. Die Java 2 Enterprise Edition

4.3.1. Bestandteile der J2EE 1.2

Im Umgang mit Enterprise JavaBeans, Application Servern und J2EE werden populäre Begriffe häufig inkorrekt dargestellt. Zum Verständnis ist es aber von Bedeutung, den genauen Aufbau zu kennen. Die Java 2 Plattform, Enterprise Edition besteht aus fünf Dokumenten bzw. Softwarebibliotheken:

- *Spezifikation*

Bei der Spezifikation handelt es sich um das zentrale Dokument, in welchem beschrieben wird, wie eine Implementation der Plattform zu geschehen hat. Sie listet alle Anwendungsschnittstellen auf, die zur Verfügung gestellt werden müssen und beschreibt detailliert die Unterstützung für Container, Clients und Komponenten. Sie ist nicht identisch mit der EJB-Spezifikation.

- *Programmiermodell*

Hierbei handelt es sich um ein Programmierhandbuch, in dem ausführlich beschrieben wird, wie die einzelnen Aspekte der J2EE genutzt werden können. Es enthält Kapitel über das Design und die Arbeitschritte zum Erstellen von J2EE-Applikationen, beschreibt Optimierungen und Beispiele. Dabei handelt es jedoch nur um Richtlinien, die sich beim Entwickeln von komponentenbasierter Software bewährt haben.

- *Plattform*

Die Plattform ist die Gesamtzahl der zusammenarbeitenden APIs. Sie besteht aus den binären Bibliotheken und baut auf der Java 2 Plattform, Standard Edition auf.

- *Referenzimplementation*

Die J2EE Referenzimplementation (RI) ist eine vollständige Beispielimplementierung aller in der Spezifikation genannten Technologien und Dienste. Sie enthält einen voll funktionstüchtigen Application Server, eine ausführliche Dokumentation und Beispiele. Sie demonstriert die prinzipielle Umsetzbarkeit der J2EE-Spezifikation und gibt Entwicklern die Möglichkeit, den Umgang mit J2EE zu erlernen oder eigene Programme zu testen.

- *Kompatibilitätstestsuite*

Die Testsuite besteht aus einer Reihe von Programmen, die alle Klassen und Methoden einer J2EE-Implementation auf Konformität mit den Anforderungen der Spezifikation überprüft. Außerdem wird die Interaktion zwischen den einzelnen Schichten auf Korrektheit und Konsistenz getestet. Sie gibt den Herstellern von J2EE-Produkten die Möglichkeit, ihre Produkte auf versteckte Fehler zu untersuchen. Ferner soll sie die Portabilität von Anwendungen zwischen verschiedenen Produkten sicherstellen.

4.3.2. Aufgabenverteilung bei der Komponentenentwicklung

Die J2EE-Plattform beschreibt sechs verschiedene, klar getrennte Rollen für die Entwicklung, Installation und Wartung einer Applikation. Die Aufteilung soll die Verantwortlichkeiten der einzelnen Parteien im Lebenszyklus einer J2EE-Anwendung definieren. Einige der Rollen sind nicht spezifisch für die J2EE-Plattform, andere kommen nur beim deren Einsatz vor. Generell können mehrere Aufgaben von einer Person übernommen werden, aber zwischen jeder Rolle besteht eine Vertragsschnittstelle, die dem Ausführenden alle Freiheiten läßt, solange er den Vertrag erfüllt.

- *J2EE Product Provider*

Der Produktanbieter stellt eine integrierte Plattform aller Container, J2EE-APIs und Dienste, die in der Spezifikation definiert wurden, bereit. Er kann optional weitere Dienste anbieten. Weiterhin liefert er die Werkzeuge zur Installation und zum Management. Er ermöglicht es dem Installateur, Komponenten auf das Produkt aufzuspielen. Der Administrator verwaltet mit Hilfe dieser Werkzeuge das Produkt und die Anwendung.

Interessanterweise wurde diese Funktion aus *Server-Provider* und *Container-Provider* zusammengelegt. Damit wurde das interessante Konzept des portablen Containers, der sich in beliebige Server integrieren läßt, scheinbar fallengelassen.

- *Application Component Provider*

Der EJB-Lieferant bietet wiederverwendbare Komponenten an, die spezielle Aufgaben übernehmen. Er liefert dabei eine JAR-Datei mit den Java-Klassen und einem Deployment Deskriptor. Dieser enthält die Abhängigkeiten der Pakete untereinander und die benötigten Ressourcen. Er wird von dem Installateur an die Infrastruktur des Kunden angepaßt.

Die Programmierung von EJBs wird sich zu einem neuen Markt der Softwareindustrie entwickeln. Der Einsatz universeller Frameworks, die grundlegende und wiederkehrende Anwendungsfälle abbilden, erlaubt die Konzentration auf die eigentlich umzusetzenden

Aufgaben. Beispielsweise wird es bald eine Vielzahl von E-Commerce-Shopsystemen auf EJB-Basis geben.

- *Application Assembler*

Der Bean-Monteur setzt eine Applikation aus den Komponenten zusammen, die er vom EJB-Lieferanten bekommen hat. Der Ansatz geht davon aus, daß EJBs in einer derartigen Vielfalt existieren, daß sich ein Anwendungsproblem allein durch Zusammensetzen vorgefertigter Komponenten lösen läßt. Der Bean-Monteur benutzt die deklarativen Vorgaben in den Deployment Deskriptoren und die grafische Oberfläche des Produkt- oder des Werkzeuganbieters.

- *Deployer*

Der Installateur benutzt die fertige J2EE-Anwendung und installiert sie auf dem Server in einer bestimmten Umgebung. Er muß die in den Deployment Deskriptoren definierten Abhängigkeiten auflösen und die benötigten Ressourcen bereitstellen. Er nutzt dazu in der Regel die Werkzeuge des Produktanbieters. Der Installateur paßt also eine Applikation an die Erfordernisse in einem bestimmten Einsatzszenario an, dazu gehören beispielsweise die Sicherheitsfunktionen, das Erzeugen eines Datenbankschemas und das Festlegen der Transaktionsstufen.

- *System Administrator*

Der Administrator ist für den Betrieb der Infrastruktur und des Netzwerks verantwortlich. Das schließt die Betreuung und Wartung der laufenden Systeme ein. Dazu nutzt er die Verwaltungswerkzeuge des Produktanbieters.

- *Tool Provider*

Ein Werkzeuganbieter stellt Hilfsmittel zum Entwickeln, Zusammensetzen und Testen von Applikationskomponenten bereit. Dabei kann es sich um Werkzeuge zum konzeptionellen Entwurf wie UML-Designer, Entwicklungsumgebungen zum Programmieren der EJBs, oder um Installations- und Administrationstools für den EJB-Server oder –Container handeln.

Diese Rolle wurde neu in die Spezifikation aufgenommen. In Zukunft soll der Zusammenhang mit der Interaktion mit anderen Rollen noch genauer benannt werden.

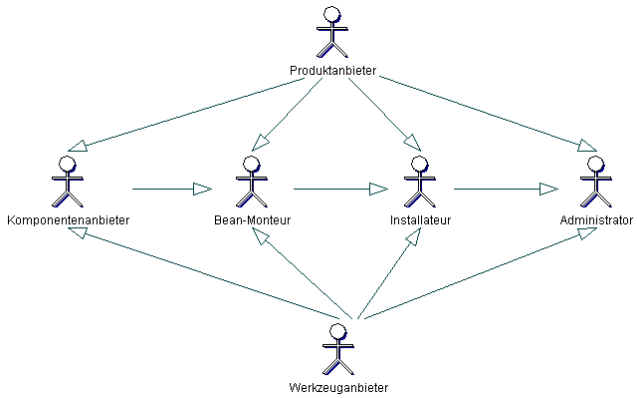


Abbildung 16: Rollen in J2EE

4.3.3. Komponenten-Container-Modell

Die J2EE-Spezifikation ist geschaffen worden, um für alle denkbaren Anwendungen von *Enterprise Information Systems* eine einheitliche Umgebung bereitzustellen. Zu diesem Zweck wurden vier verschiedene Klassen von Komponenten und ihren Containern definiert. Diese unterscheiden sich in ihrem Ablauf (auf der Client- oder der Serverseite) und ihrer Orientierung (auf Applikationen oder Webdienste). Unter Komponenten werden dabei unabhängige Softwareeinheiten auf Anwendungsebene verstanden. Container bilden ein Rahmenwerk für die Komponenten und stellen ihnen Dienste wie Lebenszyklusmanagement und parallele Ausführung zur Verfügung.

- *Application Clients*

Applikationen sind java-basierte Programme, die typischerweise auf einem Arbeitsplatz-PC eingesetzt werden und eine grafische Bedienoberfläche besitzen. Die J2EE-Sezifikation beschreibt, wie Applikationen innerhalb eines Client-Containers verwaltet werden, so daß sie zusätzliche Funktionalität eines J2EE-basierten Servers nutzen können.

- *Applet Clients*

Applets sind Java-Applikationen, die innerhalb von Web-Browsern laufen. Die J2EE-Spezifikation beschreibt eine Methode zum Verwalten von Applets in einem Applet-Container, so daß sie sich gegenüber einem J2EE-Server wie J2EE-Clients verhalten.

- *Web Application Server*

Es gibt zwei Arten von Web-Komponenten: Java Servlets und JavaServer Pages. Sie nehmen Anfragen einer Applikation entgegen und generieren eine Antwort in Web-Formaten wie HTML, XML oder WML. Ein Servlet-Container stellt Netzwerkverbindungen zur Ver-

fügung, dekodiert Parameter und formatiert Antworten. Ein JSP-Container stellt zusätzlich einen Kompilierungsmechanismus für die Übersetzung von JSP-Seiten in Servlets bereit. Ein Web-Container regelt die Zugriffe auf den J2EE-Server. Alle Container müssen mindestens das HTTP-Protokoll unterstützen.

- *EJB Application Server*

EJB-Komponenten laufen innerhalb des EJB-Containers. Dieser stellt Dienste wie Transaktionssicherung und Persistenz zur Verfügung. Wie die anderen Container stellt auch dieser alle Funktionen der Java 2 Standard Edition zur Verfügung.

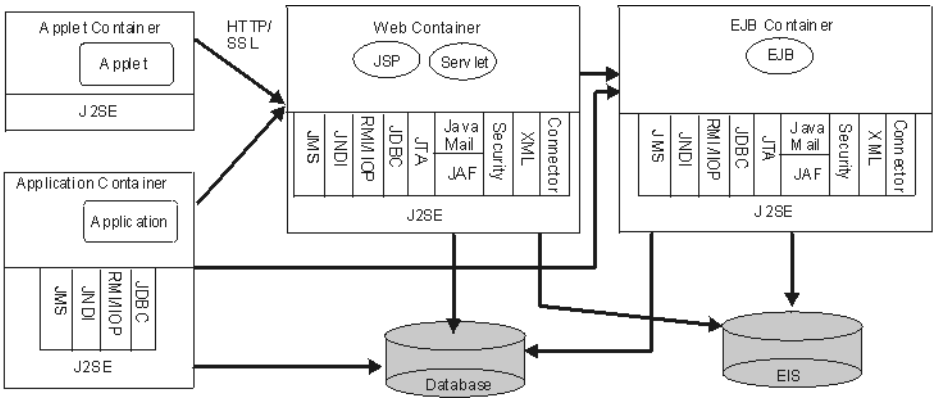


Abbildung 17: J2EE-Architektur

Die Klassifikation dient einer logischen und strukturierten Zusammenführung der Java-Komponenten. Sie sagt nicht aus, daß die vier Typen gleich gewichtet sind bzw. überhaupt vorhanden sein müssen.

4.3.4. Plattform-APIs

Die Plattformdienste vereinfachen die Programmentwicklung und erlauben eine Anpassung an Benutzerwünsche zum Installationszeitpunkt. Der Zugriff auf lokale Ressourcen erfolgt dabei auf einheitliche Art und Weise.

JNDI

In einer Umgebung mit verteilter Datenverarbeitung müssen Clients ein Verfahren nutzen können, das ihnen die Lokalisierung und Verwendung entfernter Ressourcen so ermöglicht, als ob sie sich auf der selben Maschine befinden würden. Dazu dient ein Namensdienst.

Das *Java Naming and Directory Interface* wurde entwickelt, um auf verschiedene Namens- und Verzeichnisdienste zugreifen zu können. Beispiele sind DNS, LDAP, NDS, NIS oder das lokale Dateisystem. JNDI ist lediglich eine Schnittstellenspezifikation, zur Implementierung wird ein Dienstanbieter benötigt, den jedes J2EE-Produkt enthalten muß. JNDI beinhaltet eine der wichtigsten Funktionen in einer J2EE-Anwendung, da Referenzen auf andere Komponenten oder Ressourcenfabriken (Datenbanken usw.) erst durch JNDI gefunden und angesprochen werden können.

JTA/ JTS

Eine Transaktion besteht aus mehreren Operationen, bei denen Datenstrukturen modifiziert werden. Um einen konsistenten Übergang zu gewährleisten, wird eine Transaktion als atomare Einheit behandelt, deren Änderungen im Bedarfsfall rückgängig gemacht werden können. J2EE unterstützt nur flache (nichtverschachtelte) Transaktionen.

In einer Internet-Umgebung besteht für den Client häufig keine Möglichkeit, selber Transaktionen zu steuern oder an ihnen teilzunehmen. Deshalb gibt es sowohl komponentengesteuerte als auch containergesteuerte Transaktionen. Letztere sind für Entity Beans verpflichtend. Dabei wird die Semantik über ein Transaktionsattribut festgelegt.

Die *Java Transaction API* erlaubt Applikationen die Nutzung von Transaktionen unabhängig von einer gewählten Implementation. JTA spezifiziert Schnittstellen zwischen dem Transaktionsmanager und den an der Transaktion beteiligten Parteien: der Applikation, dem J2EE-Server und dem Ressourcenmanager.

Der *Java Transaction Service* beschreibt eine Java-Anbindung zum CORBA Transaction Service (OTS), der verteilte Transaktionen unterstützt. J2EE benötigt JTA, aber nicht unbedingt JTS. Er kann durch einen anderen JTA-kompatiblen Transaktionsdienst ersetzt werden.

JDBC

Java Database Connectivity ermöglicht einen Datenbank-unabhängigen Zugriff von der J2EE-Plattform auf eine große Anzahl von Datenquellen. Insbesondere wird über JDBC der Verbindungsaufbau, die Authentifikation, die Transaktionsverwaltung und die Abfrageausführung geregelt. Die Applikationen greifen über einen speziellen Ressourcenadapter (Treiber) auf die Datenbank zu.

In J2EE ist zusätzlich das *JDBC 2.0 optional package* integriert. Es erweitert die API um Unterstützung für Verbindungspooling, Namensauflösung über JNDI, Verteilte Transaktionen und einer speziellen Datenbanktabellen-ähnlichen Java-Klasse.

JDBC ist die am häufigsten verwendete Möglichkeit zum Zugriff auf externe Informationssysteme.

Connector

Konnektoren sind spezielle Ressourcenadapter für den Zugriff auf Enterprise Informationssysteme (EIS), wie z.B. CICS von IBM oder R/3 von SAP. Sie sind spezifisch auf ein EIS abgestimmt, folgen aber einem standardisierten Design ähnlich zu JDBC. Es ermöglicht die Programmierung plattformunabhängiger und herstellerunabhängiger Anwendungen für alle J2EE-Server, die die Konnektor-API unterstützen. Der Konnektor verbirgt Details wie Transaktionsverwaltung und Sicherheitsfunktionen der gewünschten Middleware. Ein Konnektor soll mit jedem J2EE-kompatiblen Server arbeiten können, muß aber nur ein EIS unterstützen. Der Code für den Zugriff auf ein SAP-System wird sich also von dem eines Baan- oder Peoplesoft-Systems unterscheiden.

Konnektoren sind noch kein verpflichtender Teil der J2EE Version 1.2.

Security (JAAS)

Die J2EE-Plattform besitzt einen Sicherheitsdienst, der überprüft, ob Benutzer autorisiert sind, die angeforderten Ressourcen zu nutzen. Dies geschieht in zwei Stufen. Die erste ist die Authentifikation, in welcher ein Benutzer seine Identität preisgibt, normalerweise durch Name und Paßwort. Ein validierter Benutzer heißt *principal*. Es sind verschiedene Authentifikationsmechanismen möglich: über HTML-Formulare (auch verschlüsselt), über Dialog-Fenster von J2EE-Clients oder mittels Zertifikaten.

Die zweite Stufe ist die Autorisierung, bei der das System überprüft, ob die aktuellen Sicherheitseinstellungen dem Nutzer das Recht zum Ausführen der gewünschten Aktion zubilligen. Diese Einstellungen werden im allgemeinen im Deployment Deskriptor getroffen, wo für jede EJB-Methode eine eigene Zugriffsregel gesetzt werden kann. Diese bezieht sich immer auf eine Rolle (*role*), eine logische Zusammenfassung einzelner Nutzer zu einer Anwendergruppe. Die Einstellungen werden vom Bean-Installateur beim Aufspielen auf den Server festgelegt und sind dadurch sehr flexibel.

Ein EJB kann über sein Kontext-Objekt auch selber herausfinden, wer der Aufrufer einer Methode ist und ob er sich in einer bestimmten Rolle befindet. Dieses Verfahren hat aber den Nachteil, daß bei Änderungen der Quellcode neu kompiliert werden muß.

Mit JAAS 1.0 (*Java Authentication and Authorization Service*) bietet Java eine Standard-API für Identitätsprüfungen an. Leider ist JAAS noch kein Teil der J2EE-Spezifikation.

XML (JAXP)

Die *Extensible Markup Language* (XML) ist ein universeller Standard zur Strukturierung des Inhalts in elektronischen Dokumenten. XML kann flexibel an die Erfordernisse einer Applikation angepaßt werden. In XML gibt es praktisch keine vordefinierten Tags wie in HTML, sondern sie legt nur den Aufbau der Datei und einige grammatische Regeln fest. Die Logik muß von der Applikation interpretiert werden.

XML wird in J2EE hauptsächlich für die Deployment Deskriptoren genutzt. Ein Deployment Deskriptor für EJBs besteht aus zwei XML-Dateien. Die erste (`ejb-jar.xml`) beschreibt den generellen Aufbau der EJBs, ihren Namen, ihre Interfaces, Referenzen und anderes. Die verwendete *Document Type Definition* (DTD) hat die eine *public-ID* und definiert daher global gültige Regeln für den Aufbau des Deskriptors. Die zweite Datei ist herstellerabhängig und wird erst beim Installieren eines Beans auf einem J2EE-Server erstellt. Sie enthält beispielsweise die Informationen für containergesteuerte Persistenz und die Quellen für Datenbanken.

Leider gibt es in der aktuellen Spezifikation keine Anbindung an XML-Parser (JAXP), es steht aber zu vermuten, daß sich dies zukünftig ändern wird. Durch den Bedarf einer integrativen, lose gekoppelten Lösung für Verteilte Systeme gewinnt SOAP zunehmend an Bedeutung. Dieses Kommunikationsprotokoll setzt auf XML und HTTP auf und ermöglicht den Zugriff auf beliebige Komponentenmodelle.

4.3.5. Kommunikations-APIs

Die Kommunikationsprotokolle stellen Mechanismen für die Zusammenarbeit von Objekten auf verschiedenen Servern zur Verfügung. Die J2EE-Spezifikation verlangt die Unterstützung einer Reihe populärer Protokolle, jeder Hersteller kann aber zusätzliche bereitstellen.

Internetprotokolle

Die J2EE baut auf einer Reihe von Standardprotokollen auf. Diese sind TCP/ IP, HTTP und SSL. Ersteres ist das Basisprotokoll im Internet. HTTP für vom Web-Container für die Verbindung mit HTML-Browsern eingesetzt und SSL bietet verschlüsselte und damit sichere Kommunikation über HTTP.

RMI

Remote Method Invocation ist ein Mechanismus zum Aufruf von Objektmethoden auf entfernten Rechnern. Es ist nur für Java verfügbar und integriert sich nahtlos in die traditionelle

Entwicklung von Java-Code. Es verbirgt alle netzwerkbezogenen Aspekte wie Marshalling, ermöglicht dynamisches Herunterladen von Klassen und die automatische Aktivierung entfernter Objekte.

RMI-IIOP/ IDL

RMI hat einige Schwachstellen in den Punkten Interoperabilität und Performanz. Aus diesen Gründen wurde das *Internet Inter-ORB Protocol* als Kommunikationsprotokoll in die Spezifikation aufgenommen. Dies ermöglicht auch eine Integration von J2EE mit bestehenden CORBA-Systemen und führt zu einer sehr offenen und leistungsfähigen Architektur. Das JavaIDL-Subsystem erlaubt den Zugriff von J2EE auf CORBA-Objekte, RMI-IIOP dient genau für den umgekehrten Weg.

JMS

Der *Java Message Service* ist eine API zur Nutzung von nachrichtenorientierter Middleware. Er erlaubt verteilten Objekten, auf einem asynchronen, zuverlässigen Weg zu kommunizieren. Die asynchrone Ausführung kann das Gesamtverhalten eines Systems verbessern, da die Antwort nicht sofort erfolgen muß und der Sender nicht blockiert ist.

Es werden zwei Kommunikationsmodelle unterstützt, *Punkt-zu-Punkt* und *Publizieren-Abonnieren*. Dem ersten Modell liegen Warteschlangen zugrunde. Nachrichten werden in diese eingefügt und solange gespeichert, bis sie vom Konsumenten gelesen wurden. Beim zweiten Modell werden Nachrichten Themen zugeordnet. Produzenten erzeugen Nachrichten und publizieren diese unter einem Thema. Konsumenten abonnieren ein Thema und bekommen über Multicast alle Nachrichten zugestellt. Die Übermittlung geschieht dabei praktisch in Echtzeit.

Zusätzlich werden persistente Nachrichten, Empfangsbestätigung, Prioritäten, eine einstellbare Lebensdauer und Teilnahme an Transaktionen unterstützt.

JavaMail/ JAF

Die JavaMail API erlaubt das Versenden von E-Mails. Wie bei anderen APIs werden auch hier nur Schnittstellen definiert und die eigentliche Implementierung des Mail-Dienstes bzw. der verwendeten Protokolle verborgen. JavaMail baut auf dem *JavaBeans Activation Framework* (JAF) auf. JAF integriert die Unterstützung von MIME-Datentypen. Es übernimmt die Abbildung von Dateiendungen und erlaubt das Ansehen und Editieren von Multimedia-Dateien.

4.3.6. Web-Client-APIs

Soll ein HTML-Client auf EJBs zugreifen, so geschieht dies über eine Zwischenschicht. Auf dieser befindet sich ein Webserver, der Anfragen entgegennimmt und ein Web-Container, der die webserverseitigen Komponenten beinhaltet, ihnen Dienste zur Verfügung stellt und ihren Lebenszyklus regelt.

Servlets

Servlets sind Java-Komponenten, die innerhalb des Webcontainers ausgeführt werden. Sie sind durch die Java Servlet API standardisiert und liegen in binärer Form vor. Auch Servlets werden nicht direkt angesprochen. Der Web-Server erhält eine Anfrage, für deren URL ein Verweis auf ein Servlet existiert. Der Web-Server leitet diese Anfrage an den Web-Container weiter, der aus einem Pool von Servlet-Instanzen eine auswählt, um die Anfrage zu beantworten.

Beim ersten Aufruf wird das Servlet geladen und gestartet. Servlets werden nach der Abarbeitung einer Anfrage nicht beendet, sondern verbleiben in einem wartenden Zustand. Trifft eine neue Anfrage ein, kann diese ohne das zeitaufwendige Erzeugen eines neuen Prozesses beantwortet werden. Eine Servletinstanz kann auch von mehr als einem Thread gleichzeitig abgearbeitet werden, so daß auf die korrekte Behandlung von mehreren Threads z.B. bei der Modifikation von Klassenvariablen geachtet oder die Schnittstelle `SingleThreadModel` implementiert werden muß.

Java-Servlets sind nicht auf das HTTP-Protokoll beschränkt. Ein Servlet ist eine Java-Klasse, die das Interface `javax.servlet.Servlet` implementiert bzw. von der Klasse `javax.servlet.GenericServlet` abgeleitet wird. Ebenso können sie XML-, XHTML- oder WML-Dokumente erzeugen.

Ein Servlet weist einen Lebenszyklus auf, in dem definiert ist, wie das Servlet geladen und initialisiert wird, wie Anforderungen bearbeitet werden und wie es beendet wird. Die Lebenszyklusmethoden sind in der `javax.servlet.Servlet`-Schnittstelle definiert:

- `init()`: Diese Methode wird aufgerufen, wenn das Servlet vollständig geladen worden ist. Hier kann z.B. das Aufsuchen und Binden von EJBs geschehen. Init() besitzt ein Konfigurationsobjekt des Typs `ServletConfig` als Parameter. Damit kann auf den Servlet-Kontext zugegriffen werden. Dieser stellt ähnlich wie der EJB-Kontext globale Funktionen über Servlets bereit, z.B. lassen sich damit mehrere Servlets hintereinander ausführen (*servlet chaining*)
- `service()`: Hier findet die eigentliche Abarbeitung der Anfrage statt. Die Methode hat zwei Parameter: Über `ServletRequest` erhält das Servlet Zugriff auf die vom Client gesendeten Anforderungsparameter, z.B. Formulardaten und kann diese auch

modifizieren. `ServletResponse` liefert den Ausgabestrom, in den es die Antwort zurück schreibt und dazu relevante Einstellungsdaten.
- `destroy()`: Bevor eine Servletinstanz der Speicherbereinigung übergeben wird, besteht hier die Möglichkeit, geöffnete Ressourcen zu schließen oder freizugeben.

Die Servlet-API stellt eine Reihe von Funktionen zum komfortablen Umgang mit Web-Applikationen zur Verfügung. Dazu gehören Cookies, das Lesen und Parsen der Anfrage-Parameter, eine integrierte Unterstützung für Sitzungsmanagement und die Möglichkeit, Objektreferenzen an die Sitzung zu binden.

JSP

JavaServer Pages stellen eine Erweiterung des Servlet-Konzepts dar. Sie wurden als Gegenstück zu den beliebten Active Server Pages von Microsoft entwickelt. JSP sind HTML-Seiten, in die Java-Code eingebettet ist. Auch ihre Aufgabe ist es, dynamischen Content zu generieren. JSP werden beim ersten Aufruf in ein Servlet übersetzt und dann compiliert. JSP haben also Zugriff auf alle Funktionen der Java-APIs, die auch Servlets nutzen können.

Ihr Vorteil gegenüber Servlets besteht in der besseren Lesbarkeit des Quellcodes für Programmierer, die mit anderen Web-Skriptsprachen vertraut sind. Vom Standpunkt des Softdesigns (MVC) handelt es sich jedoch um einen Rückschritt, da die Präsentationsschicht mit der Controllerschicht vermischt wird. Dennoch sind JSP eine große Verbesserung, da man durch sie HTML-Code kapseln und komfortabel editieren kann. In Servlets muß jede HTML-Anweisung mittels `println()` in den Ausgabestrom geschrieben werden. Das führt zu sowohl zur unleserlichem Javacode als auch zur Unmöglichkeit, den HTML-Code zu warten und bei Änderungen anzupassen.

Die Vorteile von Servlets liegen in der höheren Ausführungsgeschwindigkeit, da sie bei einer Anfrage nicht kompiliert werden müssen. Weiterhin werden Fehler zur Kompilierungszeit erkannt. Servlet sind echte Java-Klassen und ihre Programmierung wird durch Entwicklungsumgebungen besser unterstützt.

Üblicherweise werden bei größeren Projekten Servlets zum Empfang der Anfrage, zum Auswerten der Daten und zur Abarbeitung des Workflows eingesetzt. Zugriffe auf Datenbanken oder EJBs werden in Java-Beans gekapselt und Referenzen auf diese an die Sitzung gebunden. Die Antwort wird dann mit Hilfe einer oder mehrerer JSP-Schablonen zusammengesetzt.

Die Einbindung von Java-Befehlen in JSP erfolgt entweder über das in fast allen Web-Sprachen übliche Skript-Tag `<% ... %>`, oder über die jsp-eigene und XML-konforme Variante `<jsp:declaration> </jsp:declaration>`. Es werden sechs verschiedene Arten von Syntax-Elementen unterschieden:

JSP Tags werden für Initialisierungszwecke und den Zugriff auf JavaBeans verwendet. Die Anweisungen werden dynamisch zur Laufzeit ausgewertet.

```
<jsp:useBean id="login" class="LoginBean" />
<jsp:getProperty name="login" property="lastname" />
<jsp:include page="/index.html" />
```

Direktiven werden zu Übersetzungszeit ausgewertet. Referenzierte Dateien werden statisch eingebunden.

```
<%@ page language="java" imports="java.util.*" %>
<%@ include file="mobtel_copyright.html" %>
```

Deklarationen deklarieren Variablen zur Benutzung innerhalb des Dokuments. Sie werden bei der Übersetzung in ein Servlet in die init()-Methode eingefügt.

```
<%! String username="gast"; %>
<%! int time=Date.getCurrentTime(); %>
```

Ausdrücke werden in einen String konvertiert und in den HTML-Code eingefügt. Sie entsprechen println()-Anweisungen und erwarten kein Semikolon am Zeilenende.

```
<%= user.getName() %>
```

Scriptlets sind eingebetteter Java-Code, der in der service()-Methode des Servlets ausgeführt wird.

```
<% if (isLoggedIn() == true) { %>
<h3>Willkommen</h3>
<% } else { %>
<h3>Sie müssen sich zuerst anmelden.</h3>
<% } %>
```

Kommentare werden im Gegensatz zu HTML-Kommentaren von der Servlet-Engine entfernt.

```
<%-- ein JSP Kommentar --%>
```

Ab der Version 1.1 der JSP-Spezifikation ist es möglich, JSP-Tags zu definieren und sie zu einer eigenen *Tag Library* zusammenzufassen. Dazu muß einer Java Beans-Klasse eine XML-Datei (*Tag Library Descriptor*) hinzugefügt werden, die die verfügbaren Attribute beschreibt.

```
<%@ taglib uri="http://rnvs43/tags" prefix="mobtel" %>
…
<mobtel:tagname>
…
</mobtel:tagname>
```

Eine Reihe von Variablen ist in JSP-Seiten jederzeit und ohne Definition verfügbar. Diese heißen implizite Objekte. Die nachfolgende Tabelle stellt sie mit ihren Servlet-Alternativen gegenüber.

JSP-Objekt	Servlet-Objekt	Zweck
page	this	Zugriff auf definierte Variablen und Methoden innerhalb der JSP-Seite
request	HttpRequest	Zugriff auf Parameter einer HTTP-Anforderung, kann mittels *forward* weitergeleitet werden
session	HttpSession	Zugriff auf Daten einer Sitzung
application	ServletContext	Zugriff auf alle Daten einer Applikation, d.h. einem WAR-File oder einem URL-Namensraum des Webservers
response	HttpResponse	Wird in JSP nicht verwendet
config	ServletConfig	Zugriff auf Initialisierungsparameter
exception	Throwable	Zugriff auf Fehlernachricht und Stack
out	PrintWriter	Zugriff auf die Daten des Ausgabestroms
pageContext	-	Zugriff auf Attribute

Tabelle 5: Implizite Objekte in JSP

4.4. Persistenz in EJB-Servern

Wie schon im Abschnitt über das EJB-Komponentenmodell angesprochen, gibt es zwei Typen von EJB, Session Beans und Entity Beans. Beide können auf Datenbanken zugreifen. Entity Beans können zusätzlich zwischen komponentenverwalteter und containerverwalteter Persistenz wählen. Die Auswahl hängt dabei einerseits von dem Objekt ab, das mit dem EJB modelliert werden soll. Andererseits müssen auch Aspekte wie Performanz und die Integration mit vorhandenen Datenbanken berücksichtigt werden. Im folgenden wird dabei von Relationalen Datenbanksystemen ausgegangen, da mit diesen schon ausreichend Erfahrung gesammelt werden konnte.

Im J2EE-Programmiermodell wird folgende Unterscheidung getroffen [SUN 00]: Session Beans nutzen Datenbankzugriffe in einfachen Applikation, bei denen auf das Ergebnis nicht von mehreren Clients zugegriffen werden soll. Die ausgelesenen Daten enthalten kein Geschäftsobjekt (business entity). Entity Beans werden dagegen verwendet, wenn obiges nicht zutrifft oder die Datenbankschicht vor dem Session Bean verborgen werden soll. Diese Aufteilung läßt eine Reihe von Abbildungsmöglichkeiten zu:

4.4.1. Persistenz mit Session Beans

Es kann Gründe geben, auf den Einsatz von Entity Beans zu verzichten. Zum einen müssen sie erst in EJB-Servern der Version 1.1 vorhanden sein, zum anderen gab und gibt es Probleme im Bereich der Performanz und Skalierbarkeit. Obwohl der EJB-Container einen Vorrat von Instanzen aller EJB anlegt, muß davon ausgegangen werden, daß ein Entity Bean aus Ressourcengründen auf einen Sekundärspeicher ausgelagert und später wieder eingelesen wird. Wird eine Methode eines Entity Beans angefordert, so ruft der Container die Methode `ejbLoad()` auf, um die Instanzvariablen zu synchronisieren. Nach Abschluß ruft er `ejbStore()` auf. Beide Methoden liegen im Raum der Transaktion, in der die Geschäftsmethode ausgeführt wird. Während `ejbLoad()` die Variablen aus der Datenbank liest, werden sie mit `ejbStore()` aktualisiert. Es sind also immer zwei Zugriffe nötig. Weiterhin ist ein Entity Bean bis zur Beendigung der Transaktion an den umgebenden EJB-Container gebunden.

Stateless Session Beans werden nicht ausgelagert und können bei jedem Methodenzugriff von einem anderen EJB-Container bereitgestellt werden. Dies macht eine skalierbare Lastverteilung möglich. Wird nun die eigentliche Persistenzabbildung von Stateless Session Beans übernommen, können diese für effiziente Datenbankzugriffe sorgen, indem sie beispielsweise ganze Objektlisten lesen und verwalten.

Dieses Verfahren hat allerdings gravierende Nachteile. Zuerst stellt die Vorgehensweise einen gravierenden Verstoß gegen das Programmiermodell und gegen die Kapselung logischer Objekte dar. Zum anderen gehen viele der durch EJB-Container angebotenen Vorteile wie eine granulierte Transaktions- und Sicherheitsverwaltung verloren.

4.4.2. Persistenz mit BMP

Entity Beans können die Persistenzabbildung selbst übernehmen, indem sie gewisse Lebenszyklusmethoden ausfüllen, die der EJB-Container aufruft. Datenbankzugriffe müssen beim Erschaffen, Löschen, Laden, Sichern und Finden von Objekten spezifiziert werden. Dabei wird die JDBC-API verwendet. Ein Beispiel für das Erzeugen eines einfachen Entity Beans wäre folgende Methode:

```
public String ejbCreate(String name) throws CreateException {

   try {
      InitialContext ic = new InitialContext();
      DataSource ds = (DataSource) ic.lookup(dbName);
      con =  ds.getConnection();

      String insertStatement =
         "insert into betreuer values (?)";
      PreparedStatement prepStmt =
```

```
                con.prepareStatement(insertStatement);

      prepStmt.setString(1, name);
      prepStmt.executeUpdate();
      prepStmt.close();
      con.close();
   } catch (Exception ex) {
      throw new EJBException("ejbCreate: " +
         ex.getMessage());
   }

   this.name = name;
   return name;
}
```

BMP wird zur Zeit noch häufig verwendet, da der Einsatz von JDBC-Code und SQL-Queries in Zeichenkettenform eine große Portabilität zwischen verschiedenen Produktanbietern und Datenbanken ermöglicht.

Der große Nachteil dieses Ansatzes liegt in der schweren Lesbarkeit und Wartbarkeit. Ändert sich das Datenbankschema oder müssen Funktionen hinzugefügt werden, so müssen auch die Zugriffsmethoden aller Entity Beans geändert werden.

4.4.3. Persistenz mit CMP

Die eleganteste Variante besteht in der Verwendung von containergesteuerter Persistenz. Dabei wird die Abbildung mit Hilfe von grafischen Werkzeugen erzeugt und anschließend im herstellerspezifischen Teil des Deployment Deskriptors abgelegt. Ebenso komfortabel lassen sich Finder-Methoden hinzufügen. Änderungen können ohne Neukompilierung vorgenommen werden. Häufig lassen sich die Auswirkungen von Deklarationen testweise auf der Datenbank ausführen, so daß Fehler schnell erkannt werden können. Ein Beispiel ist die Verwendung des Java-Datentyps `java.util.Date`, der ein Datumsobjekt kapselt. Der korrespondierende SQL-Datentyp heißt bei Oracle `DATE`, im Microsoft SQL Server `DATETIME` und in der IBM DB2 `TIMESTAMP`.

Die Trennung von Geschäftslogik und Datenbankzugriffscode setzt auch eine der primären Entwicklungsgründe für Komponentenmodelle durch: Daß Objekte nur ihre aus dem logischen Entwurf folgenden Aufgaben umsetzen und ihnen Dienste wie Persistenz vom Container bereitgestellt werden.

Die Mächtigkeit der Abbildung hängt bei CMP jedoch vom EJB-Container ab. Einige Hersteller bieten eine automatische Erzeugung des Datenbankschemas und umfangreiche Lösungsmöglichkeiten zu Problemen an, die im Abschnitt über objektrelationale Abbildungen besprochen worden. Andere implementieren nur die einfache Abbildung von Attributen auf Spalten.

4.4.4. Persistenz mit CMP über ein eigenes Persistenzframework

Eine letzte Möglichkeit ist die Einbettung eines eigenen Persistenzframeworks. Leider sind die Schnittstellen zwischen EJB-Container und Persistenz-Engine in der J2EE-Spezifikation nicht beschrieben. Daher ist eine Lösung auf dieser Basis immer herstellerspezifisch. Borland bietet beispielweise ein *Custom Service Provider Interface* an, mit dem die Borland Persistenz-Engine gegen eine Eigenentwicklung ausgetauscht werden kann. Diese wird über einen CORBA-Adapter an den Container angeschlossen. Dazu muß lediglich im Deployment Deskriptor die Eigenschaft `ejb.cmp.manager` auf den Klassennamen der Implementation verweisen.

Der Container reicht die Persistenzbedingungen und den Transaktionskontext über IIOP an das eigene Framework weiter bzw. ruft die im Custom SPI definierten Methoden auf. So lassen sich eigene Anbindungen an Datenbanken oder andere EIS implementieren.

4.4.5. Einschätzung

Die hier vorgestellten Varianten ermöglichen Persistenz auf sehr unterschiedliche Art und Weise. Die Verwendung von Session Beans dürfte für aktuelle Projekte keine Rolle mehr spielen, da sie dem Programmiermodell zu stark entgegen steht. BMP wird solange Verwendung finden, wie es keine umfassende Verpflichtung für die Abbildungsmächtigkeit des EJB-Containers gibt. Besonders die kommerziellen Anbieter von EJB-Komponenten können auf BMP nicht verzichten, um ihre Produkte einem großen Anwenderkreis zugänglich zu machen.

Mittelfristig wird sich CMP durchsetzen, wenn es gelingt, die oben genannten Probleme zu beseitigen. Es existieren in diesem Bereich bereits eine große Zahl von objektrelationalen Abbildungswerkzeugen, z.B. TopLink oder CocoBase. Auch im Mobtel-Projekt wurde ausschließlich CMP eingesetzt. Da hier keine vorhandenen Datenbanken integriert werden mußten und die Persistenz-Engine des verwendeten Application Servers eine überdurchschnittlich mächtige und komfortable Abbildung erlaubte, konnten die Vorzüge von CMP vollständig ausgenutzt werden. CMP ist einer der größten Vorteile einer EJB-basierten Anwendung, da der komplette Datenbankzugriffscode nicht vom Programmierer, sondern von der Persistenz-Engine bereitgestellt wird. Der Programmcode bleibt dadurch logisch strukturiert, übersichtlich und kompakt, die Entwicklungszeit wird verkürzt und die Wartbarkeit verbessert.

Weiterhin muß die Integration von Nicht-RDBMS stark verbessert werden, um eine wirkliche Zusammenführung der Computersysteme eines Unternehmens zu ermöglichen. Die Spezifizierung einer allgemeingültigen Schnittstelle zwischen Container und Persistenzabbildung wäre dazu eine grundlegende Voraussetzung.

4.5. Schwächen des EJB-Modells

Seit der Vorstellung der EJB-Spezifikation 1.0 im Sommer 1998 sind drei Jahre vergangen. In dieser relativ kurzen Zeitspanne hat praktisch jeder bedeutende Hersteller in der Softwareindustrie (außer Microsoft) eine Integrationsmöglichkeit für EJBs entwickelt. Bei der breiten Unterstützung dieses Komponentenstandards stellt sich natürlich die Frage, welche Schwachpunkte die J2EE-Plattform und die EJB-Architektur im speziellen aufweisen.

Generelle Unterschiede zu anderen Komponentenmodellen wurden im entsprechenden Kapitel behandelt, so daß hier nur die EJB-bezogenen Mängel besprochen werden sollen.

Die ersten Einschränkungen betreffen Java als verwendete Programmiersprache. So dürfen einige Klassen und Methoden von EJBs nicht verwendet werden:

- *Statische Attribute* von EJBs müssen als `final` (konstant) deklariert werden. Dies ist notwendig, um EJBs auf verschiedene JVMs verteilen zu können.
- Die Verwendung von *Threads* ist verboten, da dies einen Konflikt mit der Arbeit des Containers hervorrufen würde.
- EJB dürfen keine *AWT-Funktio*nalität für grafische Ein- und Ausgabe benutzen, da sie auf Serverseite in Containern ausgeführt werden. Diese Aufgaben übernehmen Clients.
- Sie haben keinen *Zugriff auf Dateien* über das `java.io`-Paket. Da der Ausführungsort von EJBs variieren kann, muß der Zugriff über Ressourcenmanager oder die Nutzung von Umgebungsvariablen erfolgen.
- Das Aufnehmen von *Socket-Verbindungen* ist nicht erlaubt. Statt dessen können EJBs auf URLs wie auf Datenbanken über Ressourcenreferenzen zugreifen.
- Alle Operationen, die gegen gewählten die *Sicherheitsrestriktionen* verstoßen könnten, wie z.B. das Installieren eines neuen Securitymanagers, sind verboten.
- Es gibt Einschränkungen bei der Benutzung des *Reflection-APIs*. Betriebssystembibliotheken dürfen ebenfalls nicht verwendet werden.
- Der Verwendung des `this()`-Zeigers als Parameter in Methodenaufrufen ist untersagt. Statt dessen muß `getEJBObject()` angewendet werden.

Alle diese Restriktionen beruhen auf notwendigen Gegebenheiten aus der Komponentenarchitektur. Für die meisten Fälle gibt es alternative Vorgehensweisen. Es existieren aber weitere Problempunkte, die schwerer auf die Eignung als Anwendungsplattform einwirken.

Vererbung

Vererbung auf Komponentenebene ist nicht definiert. Es ist zwar für die Beanklasse oder das Remote-Interface möglich, von Superklassen zu erben, aber nur auf Java-Objektlevel. Einstellungen für Sicherheit, Transaktionen, Persistenz usw. können nicht vererbt werden.

Jedoch ist die prinzipielle Möglichkeit für Vererbung von Komponenten nicht unumstritten, da durch die Abhängigkeit von Oberklassen der Black-Box-Ansatz von Komponenten verletzt wird.

Typsystem

Java benutzt statische Typprüfung. Zur Übersetzungszeit wird die Typverträglichkeit von Anweisungen überprüft. Bei EJBs ist die Typprüfung aufgeweicht, z.B. wird die Synchronisation des Home- und Remote-Interfaces mit der Beanklasse zur Compile-Zeit nicht vorgenommen.

Beziehungen

Es existiert keine formale Beschreibung von Beziehungen zwischen EJBs. Insbesondere die Behandlung der referentiellen Integrität und das Laden von Objektgraphen ist nicht spezifiziert.

Callbacks

In der aktuellen Version gibt es keinen Ereignisdienst. Es ist also nicht möglich, einen Client bei Eintritt eines Ereignisses zurückzurufen. Durch den Einsatz von JMS kann hier ein Umweg gefunden werden.

Sprachelemente

Die im letzten Abschnitt beschriebenen Einschränkungen der Sprachelemente können den Einsatz bestehender Frameworks verhindern.

Fat Objects

Zu jedem Bean existieren eine Reihe von Helferklassen und Verhüllschichten, die bestimmte Dienste zur Verfügung stellen. Dies führt zu einem hohen Aufwand für Methodenaufrufe auch zwischen EJBs (Parameter-Marshalling usw.). Als Folge werden aus Performanzgründen Beantypen programmiert, die eine mächtige Funktionalität bereitstellen und zum großen Teil intern abarbeiten. Dies widerspricht aber ebenfalls der Objektorientierung, wo logische Entitäten auf Klassen abgebildet werden sollen (*fine-grained objects*).

Portabilität

Der größte zur Zeit bestehende Nachteil ist die geringe Portabilität der EJBs zwischen verschiedenen Application Servern. Er ist eine Folge der Unterspezifizierung von J2EE und der daraus folgenden Nutzung herstellerspezifischer Lösungen, die sich nicht ohne Modifikationen übertragen lassen.

Einige der angesprochenen Nachteile wird die in Kürze erscheinende Version 2.0 der EJB-Spezifikation behandeln. Andere sind systemimmanent und können aus technischen Gründen nicht behandelt werden. Das Wissen über potentielle Konflikte muß bei einer Evaluierung des Einsatzes von EJBs vorhanden sein.

4.6. Application Server

4.6.1. Einleitung, Begriffe und Aufgaben

Application Server sind keine neuen Produkte. Ihr Funktionsumfang wurde noch vor einigen Jahren unter dem Begriff „Middleware“ zusammengefaßt. Im Zeichen der Globalisierung entstand durch Übernahmen und Fusionen eine immer größere Zahl weltweit operierender Firmen. Durch die Verbreitung des Internets konnten die Zweigstellen mit relativ geringem Aufwand miteinander vernetzt werden und auf gemeinsame Daten zugreifen. Kein geringes Problem war dabei die Heterogenität der Hard- und Software. Jede größere Firma besaß ein eigenes EDV-Zentrum, eigene Rechner, Programme und Protokolle, eigene Richtlinien für das Erstellen von Firmendokumenten, eigene Standards für die Erledigung der Alltagsarbeit usw.

Eine gleichzeitige Umstellung der gesamten EDV-Architektur auf einen gesamtheitlichen Ansatz ist aus offensichtlichen Gründen unmöglich, da der Geschäftsbetrieb weder unterbrochen noch gestört werden darf. Weiterhin hat z.B. das Jahr-2000-Problem gezeigt, daß selbst die Änderung einfacherer Randbedingungen in Datenbeständen den Einsatz riesiger personeller und finanzieller Ressourcen erfordert. Application Server versprechen hier einen Ausweg. Sie verlangen nicht die Umstellung zu einem Zeitpunkt, sondern während eines Zeitraumes, indem sie die Zusammenarbeit von alter und neuer Technik ermöglichen.

Es ist nicht möglich, den Begriff *Application Server* genau zu definieren, da es sich um ein Modewort handelt. Er wird in den unterschiedlichsten Zusammenhängen verwendet. Es kristallisieren sich jedoch eine Reihe von Eigenschaften heraus, die ein Application Server besitzen sollte:

1. Moderne Application Server bauen auf einem objektorientierten *Komponentenmodell* auf und arbeiten in Verteilten Systemen. Die Komponenten verwenden eine direkte, synchrone und eng gekoppelte Architektur.
2. Die Wurzeln der Application Server liegen in der transaktionalen Verarbeitung. Deshalb ist ein *Transaktionsmonitor* in jedem Application Server enthalten.
3. Die Integration bestehender Datenhaltungsysteme mit moderner Komponententechnik ist die Hauptaufgabe von Application Servern. Daher besitzen sie einen ORB (*Object Request Broker*).
4. Für die Zusammenarbeit zwischen den einzelnen Komponenten und den Zugriff auf Backend-Ressourcen ist ein *Messaging-System* (MOM) nötig. Es ermöglicht die indirekte, asynchrone und lose gekoppelte Kommunikation.
5. Um Zugriffe über das Internet via Browser zu ermöglichen, besitzen Application Server im allgemeinen einen *Web-Server*, der um zusätzliche Funktionalität zum Ausführen serverseitiger Komponenten oder Skripte erweitert wurde.

4.6.2. Hersteller von Application Servern

Die Verwirrung um den Begriff Application Server ist nicht verwunderlich, wenn man die Entstehungsgeschichte der einzelnen Implementierungen betrachtet. Alle großen Hersteller von Software in den oben genannten Kategorien sind heute mit einem eigenen Application Server am Markt vertreten. Dabei wurde Wissen von kleineren Firmen aufgekauft und in die vorhandenen Produkte integriert. Die Folge ist, daß die entstandenen Application Server eine Lastigkeit zugunsten des Spezialgebiets des Herstellers aufweisen.

Es ist wichtig, im Auge zu behalten, daß ein Unterschied zwischen Application Servern und J2EE-Servern besteht. So werden Lotus Domino, Apple WebObjects oder Microsoft MTS/COM+ nicht ohne Berechtigung ebenso bezeichnet.

Die wichtigsten EJB-basierten Application Server sind:

- *WebLogic* von BEA Systems. BEA ist vor allem für die CORBA-Implementation M3 und den Transaktionsmonitor Tuxedo bekannt. Mit dem Kauf von WebLogic wurden die Produkte zu einer Einheit integriert und als erster EJB-Application Server angeboten.
- *WebSphere* vom IBM basierte ursprünglich auf der freien Servlet-Engine Servlet-Exec. Inzwischen ist mit WebSphere der Zugriff auf Transaktionsmonitore wie CICS, Messaging-Systeme wie MQ Series und auch OS/390 Großrechner möglich.
- Der *Borland Application Server* baut auf dem Visigenic CORBA-ORB auf. Dies hat den Vorteil, dass er eine ausgezeichnete Unterstützung Verteilter Anwendungen ermöglicht.

- Weitere wichtige Application Server stammen von ATG, Iona, HP Bluestone, Silverstream, Sun und Sybase.

Diese Hersteller decken fast den gesamten Markt ab. Sie alle haben die Kompatibilitätstest bestanden, die nötig sind, um sich als *J2EE-compliant* bezeichnen zu dürfen, was hohe Anforderungen an die Einhaltung der J2EE-Spezifikation und die Portabilität der Anwendungen stellt. An dieser Stelle soll erwähnt werden, daß es auch im Bereich Open Source EJB-Server gibt. Einige von ihnen decken ebenfalls die gesamte J2EE-Spezifikation ab:

- *JBoss* ist ein vollständiger Applikation Server, für den neue API-Versionen sehr schnell verfügbar sind.
- *JOnAS* ist ein EJB-Container, der die EJB-Spezifikation 1.1 erfüllt.
- *Orion* ist ein sehr kleiner und schneller Server, der für nicht-kommerzielle Zwecke ebenfalls kostenlos ist.

4.6.3. Ausblick und Zukunft

Nachdem BEA im Mai 1999 den ersten Application Server auf den Markt brachte, hatte sich die Zahl innerhalb von einem Jahr auf ca. 30 Anbieter erhöht. Die ersten Produkte waren keine vollständigen Implementierungen der J2EE-Spezifikation und erst mit der Version 1.2 im Dezember 1999 stand ein brauchbares Gerüst zur Verfügung. Im letzten Jahr haben alle Anbieter ihre Produkte vervollständigt und verfeinert, so daß sie jetzt ungefähr dasselbe Leistungsspektrum abdecken.

In den letzten Monaten wurde vor allem die Integration mit Entwicklungswerkzeugen vorangetrieben. Damit konnte die Produktivität bei der Erstellung von Anwendungen erhöht werden. Weiterhin ist ein Trend zur Bereitstellung aufbauender Server in den Bereich Personalization, Commerce, Workflow und Wireless Web Services zu erkennen.

Im diesem Jahr wird sich das Marktvolumen für Application Server laut Gartner auf ca. 4 Milliarden Mark erhöhen. Die Verteilung der Marktanteile wird sich nicht wesentlich ändern, solange der Markt stark wächst. Die Hersteller werden versuchen, durch Zusatzfunktionalität Kunden zu gewinnen, etwa durch die

- Unterstützung des CORBA-Komponentenmodells
- Load-Balancing und Clustering
- der Bereitstellung von Management-Werkzeugen
- Kundenservice und Trainingsprogrammen oder
- Unterstützung neuer Technologien wie UUID, Konnektoren, SOAP

Es ist unwahrscheinlich, daß sich der Markt für Application Server auf zwei oder drei Anbieter konsolidieren wird. Eher wird es wie bei SQL-Datenbanksystemen einen gemeinsamen Standard, aber viele und unterschiedlich große Anbieter geben.

Zusammenfassung Kapitel 4

In diesem Kapitel wurden die Gründe für den Einsatz einer verteilten Programmumgebung und die Vorteile der Verwendung eines Komponentenmodells gezeigt. Es wurde deutlich, daß EJB am besten den Anforderung des Mobtel-Projekts entsprechen, da sie einerseits plattformunabhängig sind (im Gegensatz zu COM), von einer großen Zahl von Herstellern unterstützt werden (im Gegensatz zu CORBA) und außerdem das am einfachsten zu programmierende Modell darstellen. Weiterhin bieten sie als Teil der J2EE standardisierten Zugriff auf eine Vielzahl optionaler Dienste und Funktionen, wie z.B. die Integration mit Servlets und Java Server Pages.

Für die Datenbankabbildung werden Entity-Beans mit containergesteuerter Persistenz verwendet. Dieses ist das bevorzugte Persistenzverfahren laut den Programmierrichtlinien von SUN und wird vom Application Server optimal unterstützt. Es kombiniert einfache Entwicklung, Optimierung der Datenbankzugriffe zur Laufzeit durch den EJB-Server und gute Wartbarkeit des Programmcodes.

Die Einschränkungen des Java-Sprachumfangs durch EJB, z.B. im Bereich Threads und lokaler Ressourcenzugriffe, sind im allgemeinen dem verteilten Laufzeitmodell und der containergestützten Instanzenverwaltung geschuldet und stellen keine Behinderungen bzw. den generellen Verzicht auf bestimmte Funktionen dar.

Der für Mobtel verwendete Borland Application Server ist eines der leistungsfähigsten und umfangreichsten Produkte, die in diesem Segment erhältlich sind. Die zusätzliche Integration von CORBA ermöglicht die direkte Einbettung von Nicht-Java-Programmen.

5. Cluster zum Erreichen von Skalierbarkeit und Hochverfügbarkeit

Für Systeme, die auf zentralen Application Servern aufbauen, ist es von eminenter Wichtigkeit, daß diese ständig verfügbar sind. Im Fall des Mobtel-Projekts kann sonst kein Betreuer Termine eingeben oder deren Ausführung überwachen. Die Patienten können ebenfalls nicht auf das System zugreifen, so daß sie keine neuen Daten erhalten. Es müssen also Maßnahmen getroffen werden, um Ausfallzeiten zu planen und zu minimieren. Für den Fall eines Systemstillstandes müssen Notfallstrategien entwickelt werden.

Dieses Kapitel zeigt die Möglichkeit, die Verfügbarkeit über das durchschnittliche Niveau zu heben bzw. Pseudoausfälle durch Überlastung zu vermeiden. Dies kann sowohl durch den Einsatz von Hardware, als auch durch Software geschehen. Besondere Beachtung wird dabei dem Clustern von J2EE-Servern zuteil, da EJB aufgrund ihrer implizit enthaltenen Verteilbarkeit während ihrer Lebenszeit zwischen verschiedenen Serverinstanzen wechseln können. Der Zugriff auf redundante oder replizierte EJB sowie die Behandlung der unterschiedlichen EJB-Typen ist herstellerabhängig. Die am weitesten verbreiteten Konzepte werden beispielhaft erläutert.

5.1. Einleitung

Ein Server in einem Netzwerk bietet anderen Computern, den Clients, verschiedene Dienste an. Ist die Nutzung für den Client aufgrund eines Fehlers nicht oder nur mit gravierenden Einschränkungen möglich, spricht man von einem *Ausfall*. Den Anteil der Zeit, in dem ein Computersystem tadellos arbeitet, nennt man *Verfügbarkeit*:

$$Verfügbarkeit = \frac{Gesamtzeit - Ausfallzeit}{Gesamtzeit} * 100\%$$

Wie leicht ersichtlich ist, liegt die Verfügbarkeit in realen Szenarien unter 100%. Um die Verfügbarkeit zu maximieren, muß die Ausfallzeit minimiert werden. Gründe für Ausfälle lassen sich in zwei verschiedene Kategorien einteilen: geplante Ausfälle und Desaster. Erstere sind Ausfälle, die vom Bedienungspersonal vorsätzlich herbeigeführt werden, z.B. für Datensicherung oder Wartung und deshalb planbar sind. Letztere treten unvermutet auf und sind daher schwieriger zu handhaben. Die Zeitspanne zwischen Auftreten, Bewußtwerden, Ursachenerkennung und –behebung des Fehlers ist im vorhinein nicht kalkulierbar.

Dazu sollen zwei Beispiele betrachtet werden:

- Standardsystem: Bereitschaft von 12 Stunden je Tag, 5 Tage je Woche, 52 Wochen je Jahr = 3120h
- Kritisches System: Bereitschaft von 24 Stunden je Tag, 7 Tage je Woche, 52 Wochen je Jahr = 8760h

Für die gewünschte Verfügbarkeit ergeben sich folgende Ausfallzeiten:

Verfügbarkeit	Ausfallzeit in Stunden	
	Standardsystem	Kritisches System
99%	31:12	87:36
99,5%	15:36	43:48
99,99%	0:18	0:52
99,999%	0:02	0:05
100%	0	0

Tabelle 6: Ausfallzeiten

Aus der obigen Tabelle folgt, daß ein kritisches System, daß im Nonstop-Betrieb arbeiten soll, bei einer Verfügbarkeit von 99% jährlich beinahe vier komplette Tage ausfällt. Diesem System steht außerdem im Gegensatz zum Standardsystem keine Zeit für geplante Ausfälle zur Verfügung. Offensichtlich gibt es für viele Anwendungen die Notwendigkeit, solche Spannen zu verkürzen. Letztlich muß auch Beachtung finden, daß viele Applikationen eine maximale Ausfallzeit nicht überschreiten dürfen. Eine Unterbrechung von 6 Minuten kann 100 Mal akzeptabel sein, eine Unterbrechung von 10 Stunden vielleicht niemals.

5.1.1. Definition

Unter *Hochverfügbarkeit* versteht man ein System, welches über das gewöhnliche Maß hinaus in der Lage ist, typische Gründe für ein Auftreten von Ausfallzeiten zu kompensieren. Hochverfügbare Systeme sind nicht auf Rechenautomaten beschränkt, auch die Bremseinrichtung von Kraftfahrzeugen oder die Sehkraft des Menschen (Anzahl der Augen) können Ausfälle bis zu einem bestimmten Grad tolerieren.

Die Grundlage zum Erreichen hoher Verfügbarkeit ist die *Redundanz*, d.h. die Eigenschaft, daß wichtige Komponenten mehrfach vorhanden sind. Dies ermöglicht es, im Fall einer Fehlfunktion eines Bauteils den Betrieb weiterzuführen und keinen Ausfall zu verursachen. Sinnvollerweise müssen alle Teile, an denen Fehler auftreten können, redundant ausgelegt werden. Man sagt dann, daß System hat keinen *alleinigen Ausfallpunkt* (SPOF: Single Point of Failure). Weitere wichtige Kenngrößen sind die durchschnittliche Laufzeit zwischen zwei Ausfällen (MTBF: Mean Time Between Failure) und die durchschnittliche Reparaturzeit im Fall eines Ausfalls (MTTR: Mean Time To Repair).

Allgemein akzeptierte Werte für Hochverfügbare Systeme sind [BR 00]:

- eine Verfügbarkeit von 99,9%
- eine MTBF von 8000 Stunden
- eine MTTR von 8 Stunden.

Weiterhin sollte ein Hochverfügbares System in der Lage sein, im Falle des Ausfalls irgendeiner Komponente mindestens 60% der Nutzer eine Performanz bereitzustellen, die 80% der Leistung ursprünglichen Systems entspricht.

5.1.2. Gründe für Ausfälle

Jeder Teil eines Rechensystems kann ausfallen. In erster Linie sind dies die einzelnen Bauteile des Computers, die mechanischen und elektrischen Verschleißprozessen unterliegen, also der Ausfall eines Prozessors, einer Festplatte, einer Netzwerkkarte oder eines Speichermoduls. Im allgemeinen kann man diese Fehler durch Austausch beheben. Auch Softwarefehler können zum Systemstillstand führen, beispielsweise durch Verletzung geschützter Speicherbereiche, unzureichende Treiber, Beschädigung der Partitionstabelle, Deadlocks oder fehlendem Platz auf Datenträgern. Weiterhin hängt der Betrieb auch von dem Funktionieren der umgebenden Infrastruktur ab. Letztlich können Fehler auch durch menschliches Versagen, Sabotage oder Vandalismus hervorgerufen werden.

Um Systemausfälle konsequent zu vermeiden, muß man untersuchen, welche Defekte mit einer größeren Wahrscheinlichkeit auftreten als andere und weshalb. Obwohl nicht jeder Ausfall auf eine (einzige) Ursache zurückverfolgt werden kann und sich für ein konkretes System durchaus Unterschiede ergeben können, wurden in der Praxis folgenden Daten gemessen [GH 99, S.32ff]:

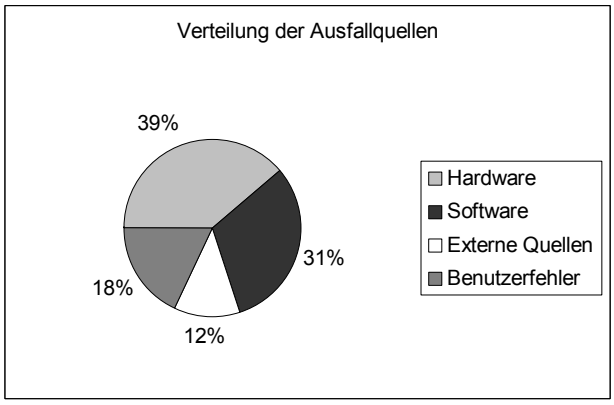


Abbildung 18: Ausfallursachen (1)

Die Gründe lassen sich noch weiter aufschlüsseln:

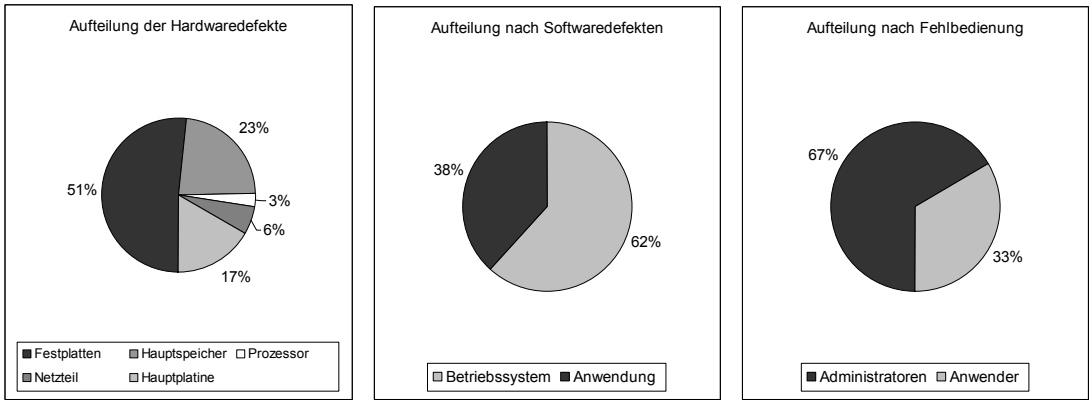


Abbildung 19: Ausfallursachen (2)

Benutzerfehler werden zu zwei Dritteln von Administratoren verursacht und nur zu einem Drittel von Anwendern. Die Hauptursache für externe Fehler ist die Stromversorgung, seltener die Internetanbindung und statistisch unwahrscheinlich, aber mit den größten Auswirkungen auf den Betrieb, Havarien wie Feuer oder Wassereinbruch.

5.1.3. Grundlagen für Hochverfügbarkeit

Die Stabilität eines Computersystems hängt von vielen Komponenten ab. Dabei muß unbedingt auf eine ausgewogene Berücksichtigung aller Einflußfaktoren geachtet werden, da für einen Ausfall des Systems ein einziger Fehler hinreichend sein kann. Nachfolgend lassen sich folgende Klassen und Elemente einteilen [WC 98]:

Vermeidung von Hardwarefehlern

- Auswahl qualitativ hochwertiger Komponenten
- redundante Auslegung (mehrfache Netzteile, Hauptspeicher mit Fehlererkennung und –reparatur, umschaltbare Netzwerkkarten, RAID-Systeme)
- Austauschbarkeit im laufenden Betrieb
- Ersatzteillager

Vermeidung von Softwarefehlern

- Datensicherung
- Möglichkeit, Daten im laufenden Betrieb zurückzuspielen (Online-Restore)
- automatischer Neustart bei Anwendungsfehlern
- Erhöhung der Fehlertoleranz eigener Anwendungen durch die Bereitstellung von Behandlungsroutinen/ Transaktionsprotokollen

Vermeidung von Bedienfehlern

- Einsatz Überwachungs- und Diagnoseprogrammen
- geschulte Mitarbeiter für Wartung und Support
- Entwickeln einer Notfallstrategie mit klaren Verantwortlichkeiten

Vermeidung von externen Fehlern

- unterbrechungsfreie Stromversorgung
- Klimaanlage
- Einbruchsicherung

5.1.4. Die Rolle der Clients

Obwohl der Ausfall eines Clients wahrscheinlicher ist als der Ausfall eines Servers, bedürfen die Clientrechner im Netzwerk keiner besonderen Hochverfügbarkeitslösung. Sie sind vom Softwarestand (relativ) identisch und daher austauschbar bzw. leicht wiederherstellbar. Zu beachten sind allerdings Fälle, in denen alle Clientrechner von Problemen gleichzeitig betroffen sein können. Beispielsweise werden in größeren Firmen Softwareupdates automatisiert außerhalb der Arbeitszeiten eingespielt. Diese müssen vorher ausgiebig auf Wechselwirkungen getestet werden.

5.1.5. Die 5 Stufen der Hochverfügbarkeit

Jeder Einsatz von Rechnern hat andere Anforderungen an Hochverfügbarkeit. Während bei einem Arbeitsgruppenserver ein Neustart im Fehlerfall genügen mag, muß ein Authentifizierungssystem immer einsatzbereit sein. Daher unterscheidet man 5 aufeinander aufbauende Stufen:

- Stufe 1 besteht aus einem einzelnen Server mit redundanten Komponenten wie Prozessor, Lüfter, Netzteil oder Hauptspeicher mit Fehlererkennung.

- Stufe 2 beinhaltet zusätzlich redundante Festplattenlaufwerke (RAID).

- Stufe 3 fügt mindesten einen weiteren Rechner hinzu. Dieser wird mit dem ersten zu einem Cluster verschmolzen. Im Fehlerfall schaltet eine Software auf den zweiten Rechner um.

- Stufe 4 erweitert das Cluster-Modell um die Eigenschaft, daß eine Anwendung auf jedem Rechner im Cluster zur selben Zeit läuft. Dies erhöht zwar nicht die Verfügbarkeit, steigert aber die Arbeitsleistung.

- Stufe 5 erfordert schließlich eine Verteilung der geclusterten Rechner über eine größere Distanz, um auch Einwirkungen von Naturkatastrophen usw. auszuschließen.

5.2. Cluster

Die bisher aufgezeigten Konzepte basieren auf der Idee, die Ausfallsicherheit des Computersystems zu erhöhen. Cluster wählen einen anderen Ansatz. Sie nutzen die Tatsache, daß der Client eine bestimmte Dienstleistung in Anspruch nehmen will, also an dem Dienst an sich interessiert, nicht an dem Computer, der den Dienst bereitstellt. Dazu werden mehrere Rechner in die Lage versetzt, diesen Dienst offerieren zu können.

Ein wichtiger Punkt ist bisher unerwähnt geblieben. Nicht in jedem Fall, in dem ein Server auf Anforderungen nicht mehr reagiert, muß ein physischer Defekt vorliegen. Ebenso kann es sich um ein Überlastungsproblem handeln. Der Server arbeitet dabei korrekt, aber die Zahl der an ihn gestellten Anfragen ist so hoch, daß er nur einen Teil davon abarbeiten kann. Die restlichen Anfragen werden in einer Warteschlange zwischengespeichert. Nach Überschreiten einer gewissen Zeitspanne geht der Client davon aus, daß ein Server- oder Verbindungsproblem vorliegt und wirft eine Fehlermeldung auf. Ist es nicht möglich oder nicht gewünscht, das Gerät durch ein leistungsfähigeres zu ersetzen, kommen häufig Cluster zum Einsatz.

Als *Cluster* bezeichnet man eine lose gekoppelte Gruppe unabhängiger Server, welche nach außen als ein System erscheinen. Die einzelnen Computer im Cluster werden *Knoten* genannt. Das Ziel von Clustern ist es, den Clients Dienste und Ressourcen anzubieten, dabei aber die Aufgaben auf die Knoten zu verteilen und so für eine erhöhte Verfügbarkeit und bessere Skalierung zu sorgen. Daraus leitet sich die Notwendigkeit ab, weitere Begriffe zu definieren.

Als *Lastverteilung* (Load Balancing) bezeichnet man die Fähigkeit, eintreffende Anfragen auf alle Knoten des Clusters zu verteilen, die diesen Dienst anbieten. So kann beispielweise der Aufruf eines Servlets zu irgendeinem Rechner im Cluster weitergeleitet werden, der eine Servlet-Engine ausführt. Die Zuteilung der Anfragen unterliegt dem Verteilungsalgorithmus. Häufig genutzt werden *zufallsbasiert* (random), *die-Reihe-rum* (round-robin), *gewichtet-nach-Server* (weight-based) oder *gewichtet-nach-Last* (load-based).

Failover ist die Möglichkeit, einen Dienst beim Versagen des ausführenden Server von einem anderen Clusterknoten übernehmen zu lassen. Im Idealfall geschieht dies für den Client völlig transparent, d.h. seine Anfrage wird einfach von einem anderen Server beantwortet. Häufig wird jedoch die aktuelle Transaktion oder Sitzung abgebrochen, der Dienst steht aber bei der nächsten Anfrage wieder zur Verfügung.

5.2.1. Softwarecluster: Der Microsoft Cluster Server (MSCS)

Der Microsoft Cluster Server ist einer der zentralen Bestandteile der Windows NT 4.0 Server Enterprise Edition. Dabei handelt es sich um eine Erweiterung der Standardversion des NT 4.0 Servers. In Windows 2000 ist ein Clusterservice ab der Advanced Edition integriert. Der MSCS besteht in der Regel aus 2 Knoten, die über ein Netzwerk (Interconnect) direkt miteinander verbunden sind. Beide Systeme besitzen eine eigene Systemfestplatte, greifen jedoch mittels eines SCSI-Adapters auf Datenlaufwerke am gemeinsamen Bus zu.

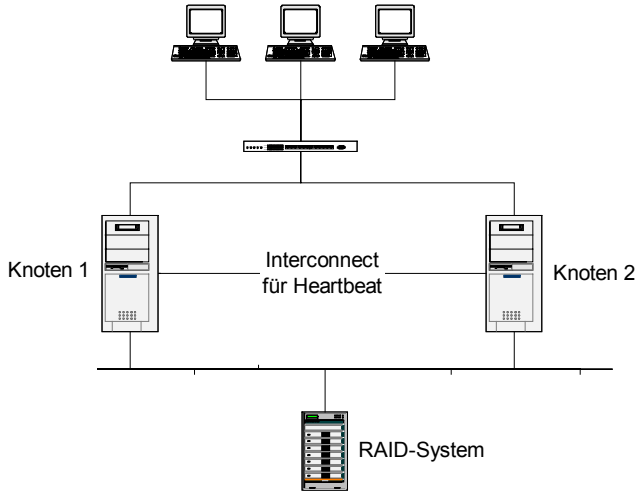


Abbildung 20: Aufbau des MSCS

Die beiden Cluster-Rechner müssen nicht aus identischen Komponenten aufgebaut sein; sie dürfen sogar leistungsmäßig differenzieren. Dies ist aber nicht empfohlen. Der MSCS basiert auf dem Shared-Nothing-Modell, d.h. nur ein Knoten kann gleichzeitig von der gemeinsamen Festplatte lesen oder schreiben. Die Clusterknoten kommunizieren über das Interconnect mittels sogenannter *Heartbeats* und erkennen, ob der jeweilige Partner noch verfügbar ist. Weiterhin gibt es eine Quorum-Ressource, auf der alle Konfigurationsdaten des Clusters gespeichert sind und die im Fall eines Interconnect-Versagens als Notfallkommunikationskanal dient. Diese steht immer beiden Knoten offen. Beide Rechner besitzen eine eigene IP-Adresse und einen eigenen DNS-Namen. Zusätzlich werden bei der Installation eine weitere IP-Adresse und ein weiterer Name festgelegt, unter denen der Cluster angesprochen wird.

Alle physikalischen und logischen Einheiten wie Netzwerkkarten, IP-Adressen oder Softwareprogramme werden als Ressourcen bezeichnet. Zwischen Ressourcen können Abhängigkeiten definiert werden, so daß beide auf einem Knoten aktiv sein müssen. Ressourcen können zu Gruppen zusammengefaßt werden. Eine Gruppe wird einem Knoten zugeordnet, auf dem sie standardmäßig läuft. Kommt es bei diesem zu einer Störung, erfolgt ein

automatischer Failover auf den anderen Knoten. Steht der erste Rechner wieder zur Verfügung, wird die Gruppe rückübertragen (Failback). Ein Fehler in einer Ressource bewirkt einen Failover der ganzen Gruppe. So lassen sich logische Einheiten hierarchisch strukturieren.

Leider profitiert nicht jede Software gleichermaßen von der Existenz eines Clusters. Am besten hat sich der Einsatz von Microsoft-eigener Software bewährt, z.B. des SQL Servers. Dieser kann in seiner aktuellen Version im Aktiv/Aktiv-Modus installiert werden. Dabei wird auf jedem Knoten ein SQL Server installiert, die Datenbanken aber auf dem gemeinsamen Laufwerk gespeichert. Im normalen Betrieb laufen dann beide Instanzen, wobei der Client nur den virtuellen Cluster-SQL-Server anspricht. Im Fehlerfall übernimmt ein SQL Server alle Datenbanken. Eine laufende Query wird aber abgebrochen und muß neu gestartet werden.

Die Vorteile eines MSCS lassen sich aber im wesentlichen nur von clusterfähiger (cluster-aware) Software nutzen. Dazu existiert eine spezielle Cluster-API in Form einer DLL. Im Lieferumfang ist eine spezielle DLL vorhanden, mit der auch nicht-clusterfähige Software eingesetzt werden kann. Bei einer zu Probezwecken installierten POET-Datenbank konnte ein Failover und Failback erfolgreich getestet werden. Hierzu wurde POET auf einem Knoten installiert. Beim Ausfall dieses Knotens werden die Registrierungsinformationen auf den anderen Knoten gespiegelt.

Als Fazit läßt sich sagen, daß der MSCS sehr sensibel auf die verwendete Hardware- und Softwarekonfiguration reagierte. Fehler hierbei können schnell zu einer Neuinstallation führen. Außerdem ist es im Test beim Failover zu den berüchtigten Bluescreens gekommen, so daß sich die Frage nach einer effektiven Erhöhung der Verfügbarkeit stellt. Manche Software reagiert zudem sensibel auf die Zeitspanne, die nötig ist, um eine fehlgeschlagene Ressource auf dem zweiten Knoten neu zu starten. Vorteile im Einsatz des MSCS bestehen vor allem in der Wartbarkeit des Systems bei Soft- oder Hardwareinstallationen und gegenüber anderen Lösungen im vergleichsweise günstigen Preis durch die Nutzung von Standardkomponenten.

5.2.2. Hardwarecluster: Marathon Endurance Server

Der Endurance Server von Marathon Technologies stellt eine auf aktive Redundanz ausgelegte Clusterlösung dar. Zuerst werden die Funktionen eines Rechners auf zwei Elemente aufgeteilt: die eigentliche Applikation läuft auf einem *Compute-Element*, das nur aus CPU und Speicher sowie aus einer MIC (Marathon Interface Card) besteht. Alle anderen Komponenten wie z.B. die I/O-Funktionen (Festplatte, Netzwerk) werden von einem anderen Element, dem *I/O-Prozessor* übernommen. Zusammen bilden sie ein Marathon-*Tupel*. Der Endurance Server besteht aus zwei Tupeln, die so ausgelegt sind, daß ein Compute-Element

und ein I/O-Prozessor ausfallen können. Beide Compute-Elemente arbeiten absolut synchron, d.h. beide führen zur selben Zeit genau denselben Befehl aus.

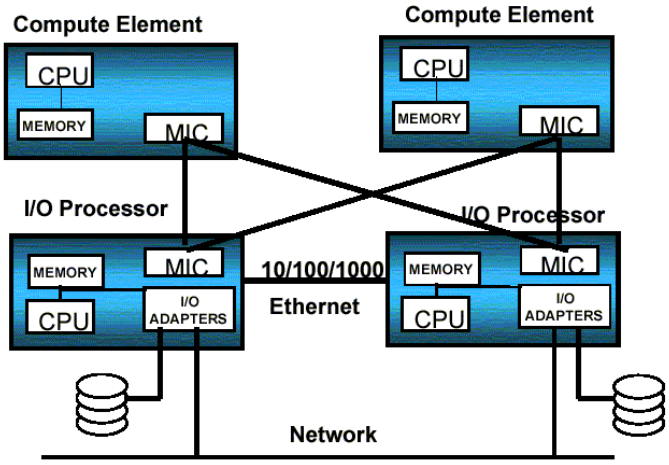


Abbildung 21: Architektur des Endurance Servers (Quelle Marathon)

Zur Zeit wird als Betriebssystem nur Windows NT unterstützt. Da die Abarbeitung auf beiden Tupeln gleichzeitig abläuft, gibt es kein Failover. Vielmehr wird beim Ausfall eines Servers die Abarbeitung kurz angehalten, der Speicherinhalt des anderen übertragen und dann fortgesetzt. Die Umschaltzeit ist dadurch sehr kurz. Vorteilhaft ist weiterhin die hohe Softwarekompatibilität, da die Anwendungen nicht bemerken können, daß sie auf mehreren Rechner laufen. Nachteilig sind einerseits die hohen Kosten des Systems, andererseits kann deterministisches Fehlverhalten von Software beide Knoten synchron stillstehen lassen.

5.3. J2EE-Cluster

Das Clustern von EJBs wird in der Spezifikation zur Zeit nicht erwähnt. Gleichwohl ist es, wie gezeigt wurde, von immanenter Wichtigkeit für kritische und großangelegte Projekte. Deshalb bietet praktisch jeder Hersteller von Application Servern eine solche Funktion an. Generell existieren drei Möglichkeiten, für EJBs Clusterfunktionen zur Verfügung zu stellen:

1. Über den Namensdienst (JNDI)
2. Über den EJB-Container
3. Über die clientseitigen Stubs

Die meisten auf dem Markt befindlichen Möglichkeit nutzen Option 1. Der Namensdienst ist die erste Instanz, die ein Client kontaktiert, wenn er mit einem EJB-Server Verbindung

aufnehmen will. Er enthält in einer Baumstruktur Einträge für die clientseitigen Stubs aller Home-Interfaces und ist deshalb ideal geeignet, den noch zustandslosen Client zu initialisieren. Option 2 ist am einfachsten zu implementieren, aber für größere Clusterfarmen nicht geeignet. Die letzte Option hat den Vorteil, daß das Clustern zum Teil auf Seiten des Client stattfinden kann.

Unabhängig davon existieren 2 Typen von J2EE-Clustern. Beim ersten handelt es sich um das schon beim MSCS beschriebene *Shared-Nothing*-Modell. Hier verfügt jeder Rechner über ein eigenes Dateisystem; Applikationen und Daten liegen lokal vor. Um die Konsistenz bei Updates zu garantieren, sind umfangreiche Replikationsmaßnahmen erforderlich. Im Gegensatz dazu greifen *Shared-Disk*-Cluster gemeinsam auf ein Datenlaufwerk zu. Updates werden hier zentral eingespielt. Wenn es jedoch zum Ausfall des Speichersystems kommt, kann kein Clusterknoten weiterarbeiten. Als Shared-Disk-Cluster kann man Rechner bezeichnen, die mehrere EJB-Container ausführen.

5.3.1. Clustering über JNDI

Der JNDI-Namensdienst stellt den Einstiegspunkt für alle J2EE-Clients dar. Zuerst wird ein neuer Namenskontext mittels `new InitialContext()` erzeugt, dann erfolgt ein Aufruf von `lookup()` mit dem gewünschten JNDI-Namen des EJBs. Das erhaltene Objekt kann dann zum Home-Interface des EJBs gecastet werden.

Um nun Redundanz zu erzielen, wäre es am leichtesten, mehrere Objektverweise an einen JNDI-Namen zu binden, so daß im Fehlerfall einfach eine andere Objektreferenz zurückgegeben werden kann. Leider muß der Name für eine Objektreferenz immer eindeutig sein, so daß dies mit nur einem JNDI-Baum nicht möglich ist.

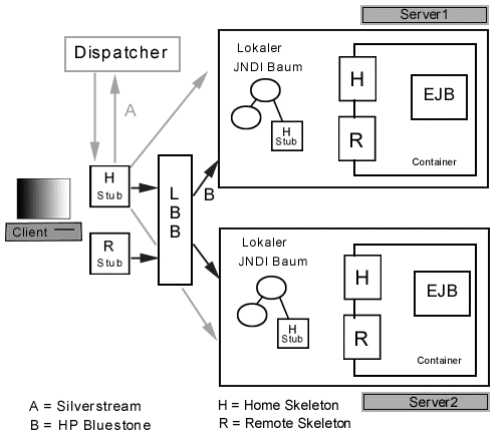


Abbildung 22: Nebeneinander existierende JNDI-Bäume

Es gibt verschiedene Ansätze, diese Einschränkung zu umgehen. Der erste besteht in mehreren *nebeneinander existierenden JNDI-Bäumen*. Jeder Clusterknoten hält einen eigenen JNDI-Baum. Er hat keinen Kontakt zu den JNDI-Bäumen anderer Knoten. Dadurch ist ein Failover nicht möglich. Vorteile dieser Herangehensweise sind die fehlenden Synchronisationsprobleme zwischen den Knoten und die konfigurationslose Erweiterbarkeit durch einfaches Hinzufügen neuer Server.

Allerdings muß der Programmierer hier im Fehlerfall selber dafür sorgen, daß er eine neue Objektreferenz bekommt. Da die Remote-Interfaces an eine konkrete Serverinstanz gekoppelt sind, muß lookup() ein weiteres Mal angerufen werden. Um aber zu erfahren, auf welchem Server dieses EJB noch aufgespielt wurde, bedarf es einer Dispatcherklasse (Silverstream Application Server), was die Portabilität der Anwendung einschränkt. Eine andere Möglichkeit ist ein Proxy Load Balance Broker (HP Bluestone), der sicherstellt, daß alle Methodenaufrufe nur an aktive Serverinstanzen versendet werden. Damit ist zwar ein automatischer Failover möglich, allerdings verlangsamt der Balancer jeden Methodenaufruf etwas. Da ein Failover-Vorgang aber sehr selten ist, wird der LBB praktisch immer umsonst aufgerufen. Er kann jedoch gut zur Lastverteilung verwendet werden.

Eine andere Herangehensweise besteht in dedizierten Namensservern (Sybase). Diese verwalten einen *zentralen JNDI-Baum*. Beim Start eines EJB-Servers werden alle Objekte an den lokalen und an den zentralen JNDI-Baum gebunden. Wenn ein Client eine Referenz auf ein EJB erhalten möchte, muß er zuerst eine Anfrage an den Namensserver senden. Von dort bekommt er eine Referenzliste, die alle Server enthält, die das EJB bereitstellen. Der Client wählt dann den ersten Server aus und bekommt Home- und Remoteinterface von diesem. Dieser zweistufige Prozeß ist natürlich etwas zeitaufwendiger. Wenn beim Aufruf einer Methode ein (nicht applikationsbegründeter) Fehler auftritt, fragt der clientseitige CORBA-Stub automatisch eine neue Referenz vom nächsten Server der Liste ab. Skalierung ist hier nur zum Zeitpunkt der Anfrage an die Namensserver möglich.

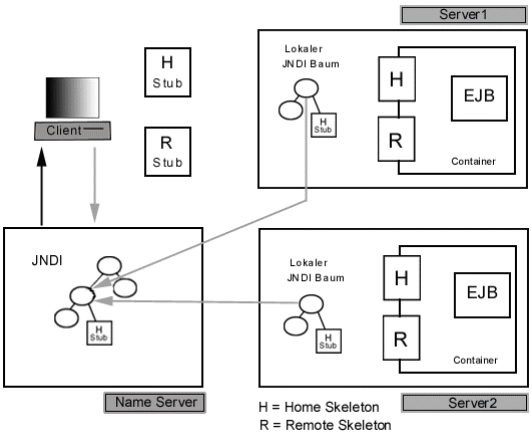


Abbildung 23: Zentraler JNDI-Baum

Für jeden 10. Application Server sollte zudem ein weiterer Namensserver in den Cluster integriert werden. Namensserver stellen hier alleinige Ausfallpunkte dar, da ohne sie ein Client keinen Kontakt zum Server herstellen kann. Weiterhin muß beachtet werden, daß das Hinzufügen eines Namensservers alle im Cluster aktiven Application Server zwingt, ihren gesamten lokalen JNDI-Baum zu replizieren, was besonders beim Start sehr zeitintensiv ist.

Die weitere Möglichkeit ist die Verwendung eines *gemeinsamen globalen JNDI-Baumes*. WebLogic 6.0 propagiert beim Start seinen lokalen JNDI-Baum via IP-Multicast zu allen anderen Servern im Cluster. Jeder Knoten bindet alle Objekte sowohl in den lokalen wie auch in den gemeinsamen JNDI-Baum. Durch das Vorhandensein eines globalen und eines lokalen Namensbaumes kennt jeder Server alle verfügbaren Objekte. Kommt ein EJB auf mehr als einem Server vor, so wird dafür ein spezielles Home-Objekt in den Namensbaum eingefügt, das alle Orte seines assoziierten EJBs kennt. Dieses Objekt liefert Remote-Interfaces zurück, die ebenfalls alle möglichen Server kennen und so ein Failover durchführen können. Das Objekt kann dabei auch Skalierungsaufgaben wahrnehmen, indem es bestimmte Knoten in vordere Listenplätze stellt.

Nachteilig ist auch hier die langwierige Synchronisation und die hohe Netzwerkbelastung. Sie wächst durch die Verwendung von Multicast linear und damit niedriger als die Punkt-zu-Punkt-Synchronisation zentraler JNDI-Bäume. Vorteilhaft ist dagegen der Verzicht auf dedizierte Namensserver und die Schnelligkeit eines lokalen Lookups verglichen mit einem Remote Lookup.

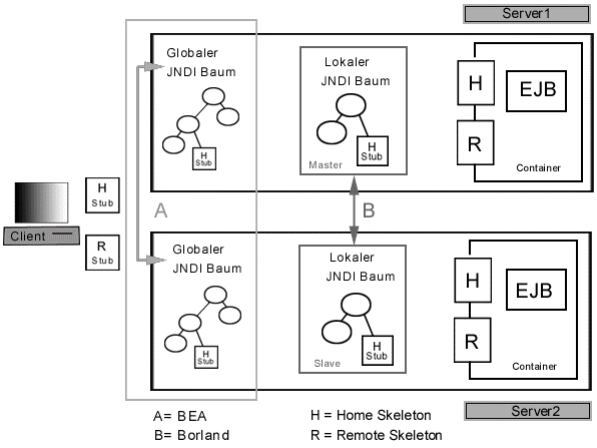


Abbildung 24: Gemeinsame JNDI-Bäume

Borland verwendet eine ähnliche Lösung, einen *replizierten globalen Namensraum*. Dabei existieren mehrere Instanzen des Namensdienstes, von denen einer der Masterdienst ist und die anderen im Slavemodus arbeiten. Die Clients verwenden dabei immer den Masterna-

mensdienst. Dieser speichert alle Referenzen auf einer gemeinsamen Persistenzlösung. Fällt der Masterdienst aus, so übernimmt der Slavedienst dessen Aufgaben und wird zum Masternamensdienst. Nun binden sich alle Clients an den neuen Masternamensdienst. Wenn der ursprüngliche Masternamensdienst wieder aktiv ist, übernimmt er wieder die Namensauflösung aller neuen Clientzugriffe, die bestehenden Clients bleiben an den Slavedienst gebunden. Dabei handelt es sich um keine vollständige Clusterlösung, da ein Ausfall der Persistenzlösung einen Ausfall der Namensdienste mit sich bringt.

Load Balancing wird hier durch die zugrunde liegende CORBA-Implementierung des Namensdienstes erreicht. Wenn eine Gruppe von EJBs auf mehrere EJB-Container aufgespielt wird, bindet der Namensdienst mehrere Objektreferenzen an einen Namen. Ruft ein Client ein EJB auf, entscheidet der Cluster mittels eines Round-Robin-Verfahrens, auf welchen Container die Referenz zeigen soll. Fällt dieser Container aus, fängt der Client-Stub die Fehlermeldung ab und erhält vom Namensdienst eine neue Referenz.

5.3.2. Clustering über den EJB-Container

Clustering kann auch durch den Anbieter des EJB-Containers realisert werden. So kann ein Container bei einer create(...)-Anfrage feststellen, daß er ausgelastet ist, z.B. da er ständig EJBs ein- und auslagern muß. Dann leitet er die Anfrage an einen anderen Container weiter. Dieser erstellt ein neues EJB und übernimmt auch sämtliche folgenden Anfragen. Dabei handelt es sich um reines Load-Balancing.

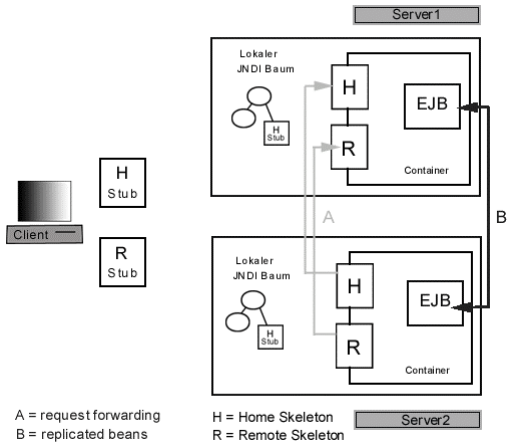


Abbildung 25: Container Clustering

Außerdem ist es denkbar, für jedes EJB eine Backup-Instanz auf einem anderen Container auszuführen und am Ende einer Transaktion zu replizieren. Die Kopie wird dabei nur dann benutzt, wenn es beim ersten Container zu einem Fehler kommt. Dann wechselt der Remote-Stub zum zweiten Container und benutzt dessen Instanz.

Da bei diesen Ansätzen die Container konfiguriert werden müssen, um sich gegenseitig erkennen zu können, stellt dies eine weniger geeignete Lösung für größere Projekte dar.

5.3.3. Clustering über die clientseitige Stubs

Die Stub- und Skeleton-Klassen des Home- und Remote-Interfaces werden erst während des Aufspielens auf den EJB-Container erzeugt. Deshalb ist es möglich, spezielle Load-Balancing und Failover-Algorithmen in die Stub-Klassen zu integrieren. Diese laufen in der Virtuellen Maschine des Clients und können deshalb sehr schnell auf Fehler reagieren. Allerdings müssen hier alle Optionen schon zur Deployzeit bekannt sein, da sie fest einkompiliert werden.

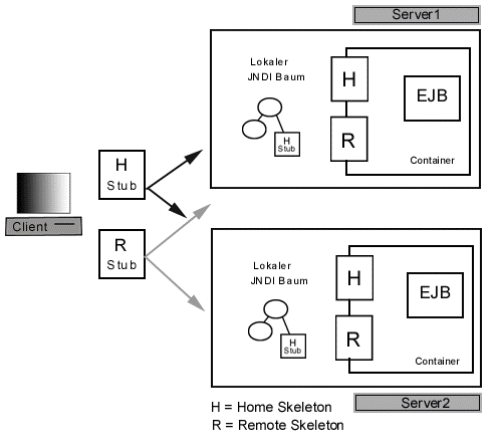


Abbildung 26: Stub Clustering

5.4. Clustering von EJBs

Es ist nicht nur nötig, die Erreichbarkeit von Servern zu erhöhen, damit diese Operationen auf EJBs ausführen können. Ebenfalls müssen die Auswirkungen eines Failover auf die einzelnen Arten von EJBs bedacht werden. Viele Hersteller ermöglichen zwar generell Clustering, d.h. die Anwendung bleibt auch nach dem Ausfall eines Servers erreichbar. Die Folgen für den Client können aber sehr unterschiedlich sein und vom einem transparenten Failover bis zu einem kompletten Verlust der aktuellen Sitzungsdaten führen. Grundsätzlich ist ein Failover nur zwischen zwei EJB-Methodenaufrufen möglich.

5.4.1. Clustern von Stateless Session Beans

Da diese, wie schon aus dem Namen zu erkennen ist, keinen Zustand bezogen auf den Client besitzen, kann einfach eine identische Kopie bereitgestellt werden. Dies wird auch

innerhalb des EJB-Containers so gehandhabt. Da alle Kopien gleich sind, kann ein anderer Container ein solches Bean anbieten, ohne daß der Client dies bemerkt.

5.4.2. Clustern von Entity Beans

Entity Beans repräsentieren persistente Objekte, die in einem Datenspeicher gesichert werden. Daher unterscheiden sich Entity Beans bezüglich ihres Zustands. Da sie nach jeder erfolgreich beendeten Transaktion gesichert werden, befindet sich der letzte gültige Zustand immer in der Datenbank. Es wird also ein Entity Bean mit den Werten aus der Datenbank initialisiert. Die zwischenzeitlich vorgenommenen Änderungen gehen verloren. Dieses Verhalten ist aber problemlos bzw. erwünscht, da durch den Ausfall des Entity Beans ein Rollback des Transaktionsmonitors erfolgt ist, der alle beteiligten Komponenten in ihren ursprünglichen Zustand zurückversetzt hat.

5.4.3. Clustern von Stateful Session Beans

Stateful Session Beans sind die am schwierigsten zu behandelnden Objekte. Da sie einen Zustand besitzen, der aber nicht persistent ist, gibt es keine triviale Methode, diesen wiederherzustellen. Im allgemeinen werden die folgenden beiden Varianten benutzt:

- *In-Memory-Replication* schreibt eine Kopie jedes Session Beans auf einen Backup-Server. Beim Failover wird diese Kopie benutzt und eine neue Kopie geschrieben.
- *Session Persistence* schreibt den Inhalt von Session Beans bei jeder Änderung oder nach einem Zeitintervall auf ein Speichermedium, von dem es wieder gelesen werden kann. Der Nachteil dabei ist, daß eventuell nicht jede Änderung übernommen wurde (bei einem Zeitintervall) und daß das Speichermedium einen alleinigen Ausfallpunkt darstellt.

5.4.4. Clustern von Message Driven Beans

Da dieser Beantyp erst in der kommenden Version der EJB-Spezifikation definiert wird, ist die Unterstützung zur Zeit sehr gering. Message Driven Beans werden nicht direkt über lookup() aufgerufen, sondern nutzen einen Nachrichtendienst. Dieser muß ebenfalls inklusive aller Nachrichtenschlangen und –themen clustert werden und die Wiederherstellung der Nachrichten zum Zeitpunkt des Ausfalls muß gesichert sein.

Zusammenfassung Kapitel 5

Wie gezeigt wurde, gibt es vielfältige Mittel, die Ausfallsicherheit von Rechnern zu erhöhen. Dabei dürften Hard- und Softwarecluster wegen des hohen Preises, der Festlegung auf eine Plattform und der trotzdem bestehende Restrisiken keine Alternativen für Mobtel sein. Viel interessanter ist der Einsatz von J2EE-Clustern, da diese sowohl die Skalierbarkeit als auch die Verfügbarkeit verbessern können und keine zusätzliche Hard- oder Softwareschicht benötigen. Im Hinblick auf die stark steigenden Preise für Rechnersysteme ab einer gewissen Leistungsgrenze (So sind Multiprozessorsysteme meist deutlich teurer als das Produkt der Einprozessorvarianten) dürften Cluster aus Standard-PCs zukünftig eine interessante Alternative für ununterbrochen arbeitende Systeme werden.

6. Compperf – Ein praktisches Beispiel einer J2EE-Applikation

Im letzten Kapitel wird eine Beispielanwendung auf Basis wird EJB untersucht. Obwohl diese kein Teil des Mobtel-Projekts ist, kann sie als eine Art kleinere Referenzimplementierung betrachtet werden. Die angewendeten Strategien und die gewonnenen Erkenntnisse gelten auch für ähnlich geartete Anwendungen. Im Vordergrund stehen dabei Leistungsuntersuchungen der J2EE-Architektur und speziell der Konfiguration, die so ähnlich auch für Mobtel zur Anwendung kommt.

6.1. Einführung

In den vorigen Kapiteln wurde gezeigt, daß die EJB-Spezifikation von SUN eine interessante und geeignete Architektur für die Entwicklung verteilter Anwendungen ist. Die Vielzahl namhafter Hersteller in der IT-Industrie, die diesen Standard unterstützen, zeigt die zukünftige Bedeutung von J2EE-Applikationen für die Entwicklung und den Einsatz auf zentralen Servern oder Serverfarmen. Viele Firmen testen heute schon den Einsatz von EJB-Application Servern, und doch besteht vielerorts die Befürchtung, das EJB-Modell könnte den örtlichen Gegebenheiten nicht gewachsen sein.

Tatsächlich sind es jedoch nicht nur Bedenken wegen der andauernden Weiterentwicklung der APIs oder der etablierteren Konkurrenz, die dazu beigetragen haben, daß die Zahl der J2EE-Anwendungen noch sehr gering ist. Es fehlt zur Zeit auch an Erfahrung, die über das Stadium von Prototypen oder kleinen Spezialanwendungen hinausgeht. Es fehlen weiterhin Vergleichs- und Kompatibilitätsbenchmarks zwischen den verschiedenen Herstellern und ebenso standardisierte Testverfahren für den Lastdurchsatz der Server.

Es soll an dieser Stelle eine Testapplikation vorgestellt werden, die die softwarearchitektonischen Möglichkeiten vorstellt, die jeder J2EE-kompatible Application Server beherrschen sollte. Produktspezifische Aspekte wurden nicht untersucht. Weiterhin zeigt sie die generelle Machbarkeit von J2EE-Applikationen im Hinblick auf die Umsetzung gegebener Objektmodelle und die Performanz beim Zugriff multipler Clients. Sie orientiert sich lose am TPC-W Benchmark [TPCW], der eine Web-Applikation auf Basis eines Online-Buchshops darstellt.

6.2. Postulationen

Besondere Berücksichtigung wird dabei dem Lastverhalten bei steigender Zahl parallel zugreifender Clients sowie den Antwortzeiten bei verschiedenen Füllgraden der Datenbank zuteil. Folgende Vermutungen werden aufgestellt:

1. Bei einer linearen Zunahme konkurrierender Clients wird eine fast lineare Zunahme der Antwortzeit erwartet. Hier sollten die Vorteile von Objekt- und Verbindungspooling und –chaching zu Tage treten. Erst ab dreistelligen Clientzahlen wird die Antwortzeit exponentiell steigen, da dann die Hardwareressourcen ein teilweises Auslagern erzwingen.

2. Der Füllgrad der Datenbank wird eine größere Auswirkung auf die Leistungsfähigkeit der Applikation haben, als dies bei anderen, speziell zustandslosen Architekturen der Fall ist. Da im EJB-Modell die Entity-Beans direkt an die Datenbank gekoppelt sind, wird ihr Zustand zu Beginn jeder Transaktion geladen und nach deren Ende geschrieben. So erfordert beispielsweise eine Abfrage nach allen Patienten eines Betreuer nicht nur einen Datenbankzugriff (SELECT * FROM PATIENT WHERE BETREUER=<name>), sondern für jedes Tupel der Ergebnismenge einen weiteren (SELECT <attribute des beans> FROM PATIENT WHERE <spalte> = <primary key des Beans>).

3. Die Ausführungszeit wird für die einzelnen Typen von Anfragen sehr verschieden sein. Am schnellsten werden statische Inhalte (HTML-Seiten) zurückgeliefert, gefolgt von dynamischen Anfragen mit festen Parametern (HTML-Einfügemaske) und Aktionen, die nur wenige Objekte in ihrem Transaktionsbereich haben (Erzeugen eines Objekts und Hinzufügen zur Datenbanktabelle). Am längsten werden Auswahloperationen auf großen Datenmengen und kaskadierende Löschoperationen dauern.

6.3. Aufbau der Testapplikation

Bei Compperf handelt sich um eine vollständige und kompatible J2EE-Applikation. Sie liegt streng nach der J2EE-Spezifikation als EAR-Datei vor und enthält eine JAR-Datei für die Server- und Clientklassen und eine WAR-Datei für die Web-Applikation. Jedes Archiv enthält außerdem 2 XML-Deskriptoren, mit denen sich alle relevanten Einstellungen vornehmen lassen.

Inhaltlich liegt der Anwendung die Vergabe von zeitlich festgelegten Aufgaben von Betreuern an die von ihnen geleiteten Patienten zugrunde. Dabei besitzt jeder Patient einen

Betreuer und eine beliebige Zahl von Terminen, die ihm von diesem Betreuer zugewiesen wurden. Ein Termin besteht neben dem Patienten, der ihn ausführt und dem Betreuer, der ihn hinzugefügt hat, aus genau einer Aufgabe, die zu einer bestimmten Zeit ausgeführt werden soll. Ferner soll jeder Patient über eine Adresse verfügen und Betreuer und Patienten sollen aus einer gemeinsamen Klasse Person abgeleitet werden. Damit ergibt sich folgendes Objektmodell:

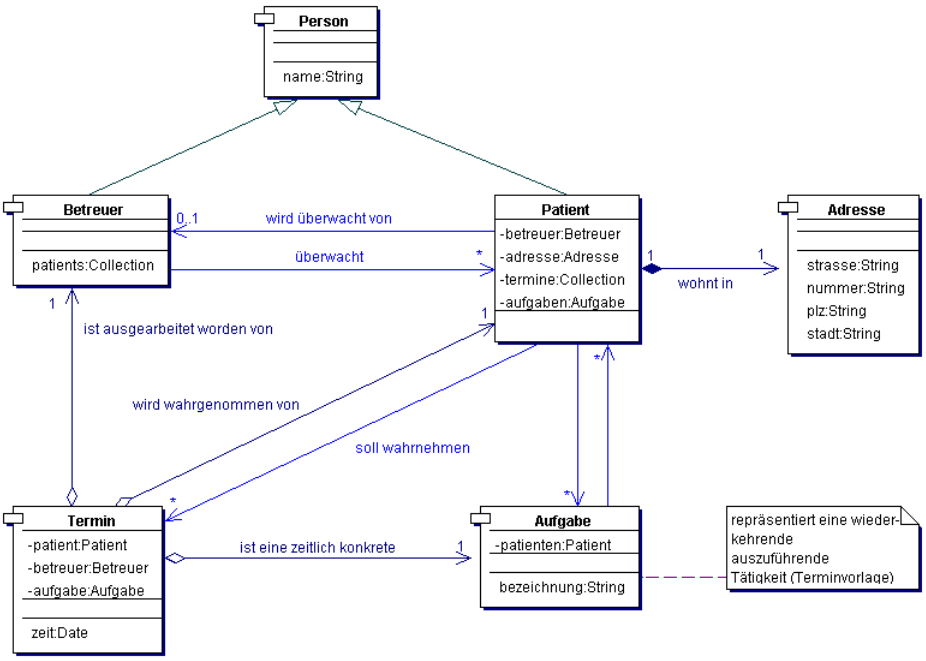


Abbildung 27: Objektmodell

6.4. Persistenzschicht

Obige Abbildung zeigt alle gerichteten Assoziationen zwischen den Objekten an. Der hohe Grad der Verknüpfung untereinander ist gut geeignet, die Fähigkeit von EJB-Container und Persistenz-Engine zu demonstrieren. Die Objekte *Betreuer*, *Patient*, *Aufgabe* und *Termin* wurden als Entity-Beans implementiert. Für alle wurde *container managed persistence* (CMP) verwendet. Dadurch erfolgt die Abbildung auf die Datenbank, ohne daß direkter JBDC-Code programmiert werden muß. Als Primärschlüssel dienten dabei die Namen bzw. Bezeichnung. Einen Sonderfall stellt die Klasse Termin dar. Sie besitzt einen zusammengesetzten Primärschlüssel, der durch die Klasse TerminPK dargestellt wird. Der Schlüssel besteht aus den Attributen Betreuername, Patientenname und Zeit. Das ermöglicht die Durchsetzung einer Grundbedingung, nämlich daß ein Patient zu einem Zeitpunkt nur eine Aufgabe von seinem Betreuer zugeordnet bekommen darf.

Die Speicherung der Daten erfolgt in der SQL-Datenbank in Tabellen, so daß sich eine vernünftige Abbildungsstruktur überlegt werden muß. Das Datenmodell beinhaltet alle

wichtigen Typen von Objektrelationen. So besteht zwischen Patient und Adresse eine einfache 1:1-Beziehung, die in der Datenbank durch eine Tabelle *Adresse* mit einem Primärschlüssel *Patientenname* abgebildet wird. Der EJB-Container muß also das <name> Attribut des Entity-Beans *Patient* auf mehrere Tabellen verteilen. *Adresse* wird in diesem Fall als „abhängiges Objekt" bezeichnet, da sein Lebenszyklus an das zugehörige Enterprise-Bean gekoppelt ist.

Ein Patient besitzt immer genau einen Betreuer, dieser kann aber mehrere Patienten haben. Dieser Fall ist eine 1:n-Beziehung. Sie wird durch eine Fremdschlüsselbeziehung in der Tabelle *Patient* gelöst, indem dort auch der Primärschlüssel von *Betreuer* abgespeichert wird. Der EJB-Container kann diese Beziehung in beide Richtungen auflösen, wenn er mit entsprechenden Finder-Methoden konfiguriert wurde.

Die komplizierteste Beziehung ist die m:n-Beziehung zwischen Patient und Aufgabe. Sie wird durch eine Fremdschlüsseltabelle aufgelöst. Dazu bietet sich *Termine* an, da dort die Primärschlüssel von *Patient* und *Aufgabe* enthalten sind.

Die Applikation soll beispielhaft aber noch eine ganze Reihe weiterer Techniken präsentieren. So erben *Patient* und *Betreuer* von der Oberklasse *Person*. Durch die Unterstützung von Vererbung lassen sich ähnlich geartete Strukturen und Methoden zusammenfassen. Allerdings wird Vererbung im Relationenmodell der Datenbanken nicht unterstützt. Compperf verwendet hier die horizontale Partitionierung, d.h. die Attribute der Oberklasse sind in den Tabellen der Unterklassen enthalten.

Damit ergibt sich folgende Datenbankstruktur:

Adressen

refname	varchar(50)	NOT NULL
nummer	varchar(50)	
plz	varchar(50)	
stadt	varchar(50)	
strasse	varchar(50)	

Aufgaben

bezeichnung	varchar(50)	NOT NULL

Betreuer

name	varchar(50)	NOT NULL

Patienten

name	varchar(50)	NOT NULL
betreuer	varchar(50)	

Termine

patient	varchar(50)	NOT NULL
betreuer	varchar(50)	NOT NULL
zeit	date	NOT NULL
aufgabe	varchar(50)	

Tabelle 7: Datenbankschema

Wie man sehen kann, ist das Schema sehr einfach aufgebaut. Es sind keine Beziehungen zwischen den Tabellen definiert; diese haben nicht einmal einen Primärschlüssel. Für die Datenintegrität ist in diesem Beispiel einzig der EJB-Container verantwortlich. Der völlige Verzicht auf die konsistenzerhaltenen Funktionen in Datenbanken wie Trigger, Constraints oder referentielle Bezüge auf Datensätze in anderen Tabellen ist in der Praxis wenig sinnvoll. Diese Zugriffe können von der Datenbank lokal um eine Vielfaches schneller vorgenommen werden. Das Beispiel könnte aber auch auf mehrerer Datenbanken verteilt werden. Dazu wäre ein JDBC2.0-XA-Treiber nötig, der *two phase commit* (2PC) unterstützt. In diesem Fall müßte der Application Server oder ein Transaktionsmonitor die Datenintegrität sicherstellen, da ein Datenbankrechner nicht ohne weiteres auf einen anderen zugreifen kann.

6.5. Benutzungsszenarien

Aus dem Objektmodell lassen sich eine Reihe von Anwendungsfällen definieren, wobei sich hier auf die grundlegendsten beschränkt wurde. Ein Nutzer, der die Applikation startet, muß zuerst seinen Betreuernamen eingeben. Benutzer können vom Administrator hinzugefügt und entfernt werden. Auf eine Authentifizierung wurde im Beispiel verzichtet. Der Betreuer kann einen seiner Patienten auswählen, seine Termine ansehen und ihm einen neuen Termin zuteilen. Ferner kann er auch neue Aufgaben erstellen, die als Terminvorlage verwendet werden. Im Prinzip handelt es sich um Einfüge-, Lösch- und Auswahloperationen auf den 4 Objekten, die als Entity-Beans modelliert wurden.

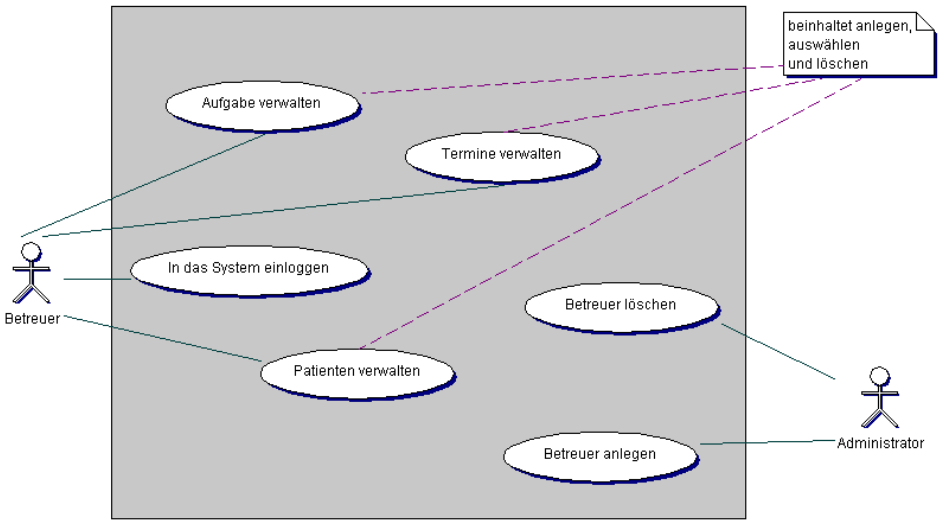


Abbildung 28: Anwendungsfälle

6.6. Sitzungsschicht

Für den Zugriff auf Entity-Beans sollen laut J2EE-Programmiermodell [SUN00] Session-Beans benutzt werden. Session-Beans sind transiente Objekte, deren Zustand nicht über ihren Lebenszyklus hinaus gehalten wird. Im Beispiel finden beide Arten von Session-Beans Verwendung: *LoginBean* ist ein zustandsloses Session-Bean, d.h. daß es nicht an einen konkreten Client gebunden ist. Seine Instanzen sind aus Sicht des Clients alle gleich. Während eines Funktionsaufrufs können auch solche Session-Beans einen Zustand besitzen (z.B. durch den Inhalt ihrer Instanzvariablen), aber nach Ende des Aufrufs geht dieser verloren, der Client kann nicht noch einmal darauf zugreifen. LoginBean deckt die Anwendungsfälle „Betreuer hinzufügen“ und „Betreuer entfernen“ ab. Der Administrator ist ein virtueller Akteur, da er im Datenmodell nicht vorkommt. Deshalb können seine Aktionen ohne Bindung an einen Benutzer ausgeführt werden.

CareBean ist ein zustandsbehaftetes Session-Bean. Der Zustand besteht im wesentlichen aus dem Betreuernamen, der beim Login eingegeben werden muß. Während der gesamten Sitzung besteht eine virtuelle Bindung des Clients, der die Referenz hält, an das Session Bean. Das heißt nicht, daß für jeden Client ständig „sein“ Session-Bean existiert, aber wenn der Client eine Methode aufruft, weist ihm der EJB-Container ein Session Bean mit eben diesem Zustand zu. CareBean kapselt alle anderen Anwendungsfälle. Da sie technisch gesehen alle gleich aufgebaut sind, würde eine weitere Granulierung keine Vorteile bringen.

Compperf benutzt containergesteuerte Transaktionen. Der EJB-Container beginnt eine neue Transaktion bevor eine Bean-Methode aufgerufen wird und beendet sie, nachdem sie ausgeführt wurde (oder im Fehlerfall). Das Transaktionsverhalten wird durch Attribute gesteuert. Alle Entitiy-Bean-Methoden und alle Methoden von CareBean haben das Attribut „required“. In diesem Fall ist die Ausführung innerhalb einer Transaktion verpflichtend. Praktisch bedeutet das, daß beim Aufruf einer Methode von CareBean durch den Client eine neue Transaktion beginnt, da zu diesem Zeitpunkt noch keine besteht. Ruft CareBean Methoden anderer Beans auf, so geschieht dies im Rahmen der aktuellen Transaktion. Anschließend wird die Transaktion für gültig erklärt (commit) oder bei Auftritt eines Fehler abgebrochen (abort). Schlägt ein Methodenaufruf fehl, werden alle vorhergehenden Änderungen zurückgesetzt (rollback). Eine Ausnahme hiervon bildet LoginBean. Als zustandloses SessionBean hat es das Transaktionsattribut „supports“. Hier wird eine schon bestehende Transaktion übernommen. Im Beispiel beginnt der Client keine Transaktion, so daß erst beim Aufruf einer Methode eines Entity-Beans von LoginBean die transaktionale Abarbeitung beginnt.

6.7. Präsentationsschicht

Die Compperf-Anwendung läßt sich vom jedem Browser mittels eines HTML-Frontends bedienen. Auf Serverseite wird jede Anforderung vom Web-Server an ein Servlet weitergeleitet. Dieses generiert aus der Anfrage dynamisch HTML-Seiten. Eine webbasierte Anwendung kann eine Vielzahl gleichzeitiger Clientanfragen unterstützen. Das Servlet bildet aber nur eine Hülle für die Clientanfragen, die es an die Session-Beans weiterreicht. Es könnte leicht ein Java-GUI-Client entwickelt werden, der dieselben Aufgaben erfüllt. Weiterhin implementiert es eine einfache Workflow-Engine, die aus den Anfrageparametern gültige Übergänge zu weiteren Aktionen bestimmt.

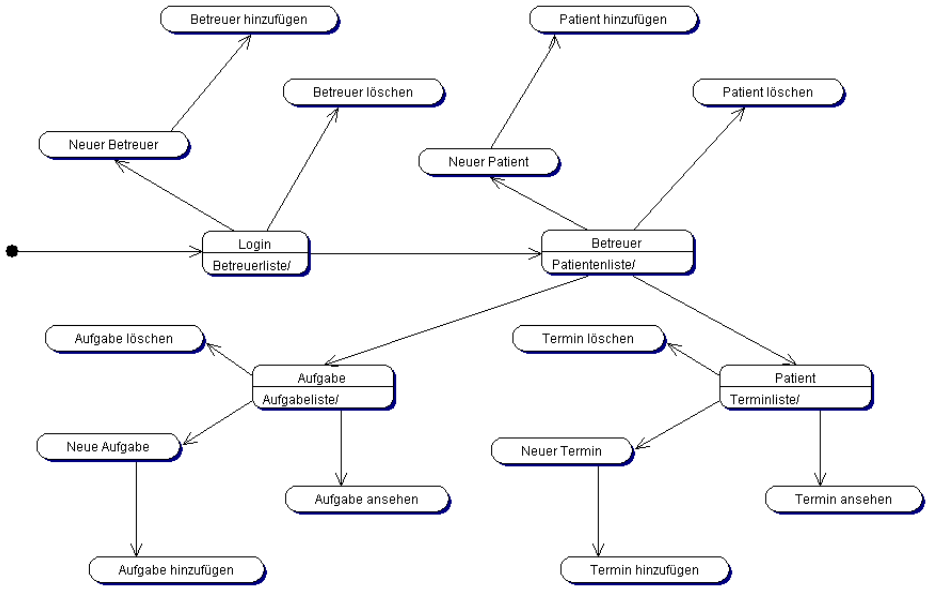


Abbildung 29: Navigationsworkflow

Das Servlet verwendet Funktionen der Servlet-API, um eine virtuelle permanente Verbindung zum Client aufzubauen (HttpSession). Während alle anwendungsfallrelevanten Aktionen den EJBs übertragen werden, kann das Servlet über die Sitzung eine dynamische Menüleiste bereitstellen, die in obiger Abbildung aus Gründen der Übersichtlichkeit nicht zu sehen ist. Die Menüleiste ermöglicht neben einem expliziten Logout den Sprung zu den zentralen Objektlisten und kann von jeder Seite aufgerufen werden. Natürlich ist nur der Sprung zu solchen Listen möglich, die von der Applikationslogik erreicht werden können. Beispielsweise kann ein Betreuer, der eine Aufgabe hinzugefügt hat, nur dann zur Terminliste eines Patienten springen, wer er zuvor innerhalb der Sitzung schon einen Patienten zum Betreuen ausgewählt hatte. Wie man aus Abbildung zum Navigationsworkflow ersehen kann, wurden die Anwendungsfälle nicht 1:1 auf HTML-Masken abgebildet.

6.8. Test-Clients

An dieser Stelle ist das Compperf-Beispiel bereits eine vollständige J2EE-Applikation. Um jedoch Tests bezüglich des Skalierungsverhalten durchführen zu können, war es notwendig, noch eine Reihe weiterer Clients zu entwickeln, um Anfragen automatisch und mit gewissen Randbedingungen durchführen und auswerten zu können.

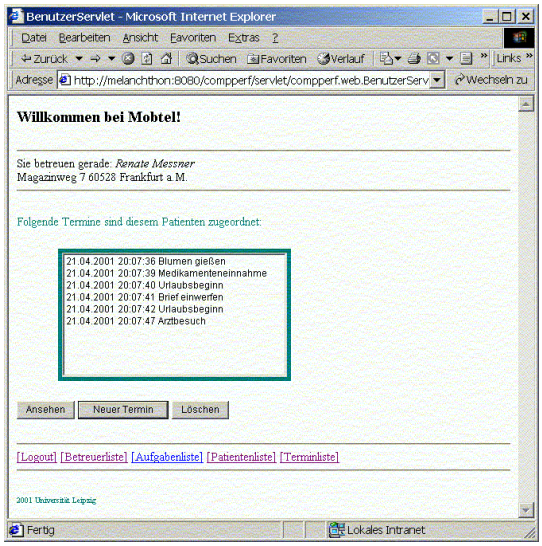


Abbildung 30: HTML-Client

Zuerst wurde das Programm *PopulateDB* zum Füllen der Datenbank mit vorgebbaren Werten entwickelt. Es liest Werte aus einer Textdatei und fügt sie in die Datenbank ein. Dadurch kann sie mit gewünschten realen Datensätzen gefüllt werden. Die von PopulateDB abgeleitete Klasse *PopulateBIGDB* erlaubt das Kreieren von Datenbanken mit einem beliebigen Füllgrad. Diese Datenbanken wurden später für die Skalierungstests verwendet.

Die Klasse *SimulatedWebClient* emuliert einen Browser. Sie sendet ebenfalls GET-Anfragen an das Servlets und wertet die erhaltene Antwort aus, indem sie sie nach enthaltenen Eingabefeldern, Buttons und Links durchsucht. Auf Grundlage dieser Informationen und anhand einer Wahrscheinlichkeitstabelle entscheidet der Client, welchen Weg er im Workflow als nächsten zurücklegen will. Dieses Verhalten kommt dem eines realen Nutzer sehr viel näher als das sukzessive Abrufen aller HTML-Screens.

Zur Steuerung mehrerer Clients als Threads wurde das Programm *SimulatedWebApplet* entwickelt. Dieses Programm ist ein grafisches Frontend für SimulatedWebClient, ermöglicht aber gleichzeitig den Einsatz einer beliebigen Zahl dieser Clients parallel. Weiterhin kann

eine Pausenzeit zwischen den Anfragen der Threads eingestellt werden, was die Realität zusätzlich erhöht.

Eine genauere Beschreibung der Funktionsweisen der Klassen sowie eine ausführliche Dokumentation befindet sich im Anhang. Auf Performanzoptimierungen oder das Ausnutzen produktspezifischer Eigenarten, die nicht der J2EE-Spezifikation entsprechen, wurde bewußt verzichtet. Die nachfolgende Abbildung zeigt das gesamte Klassenmodell:

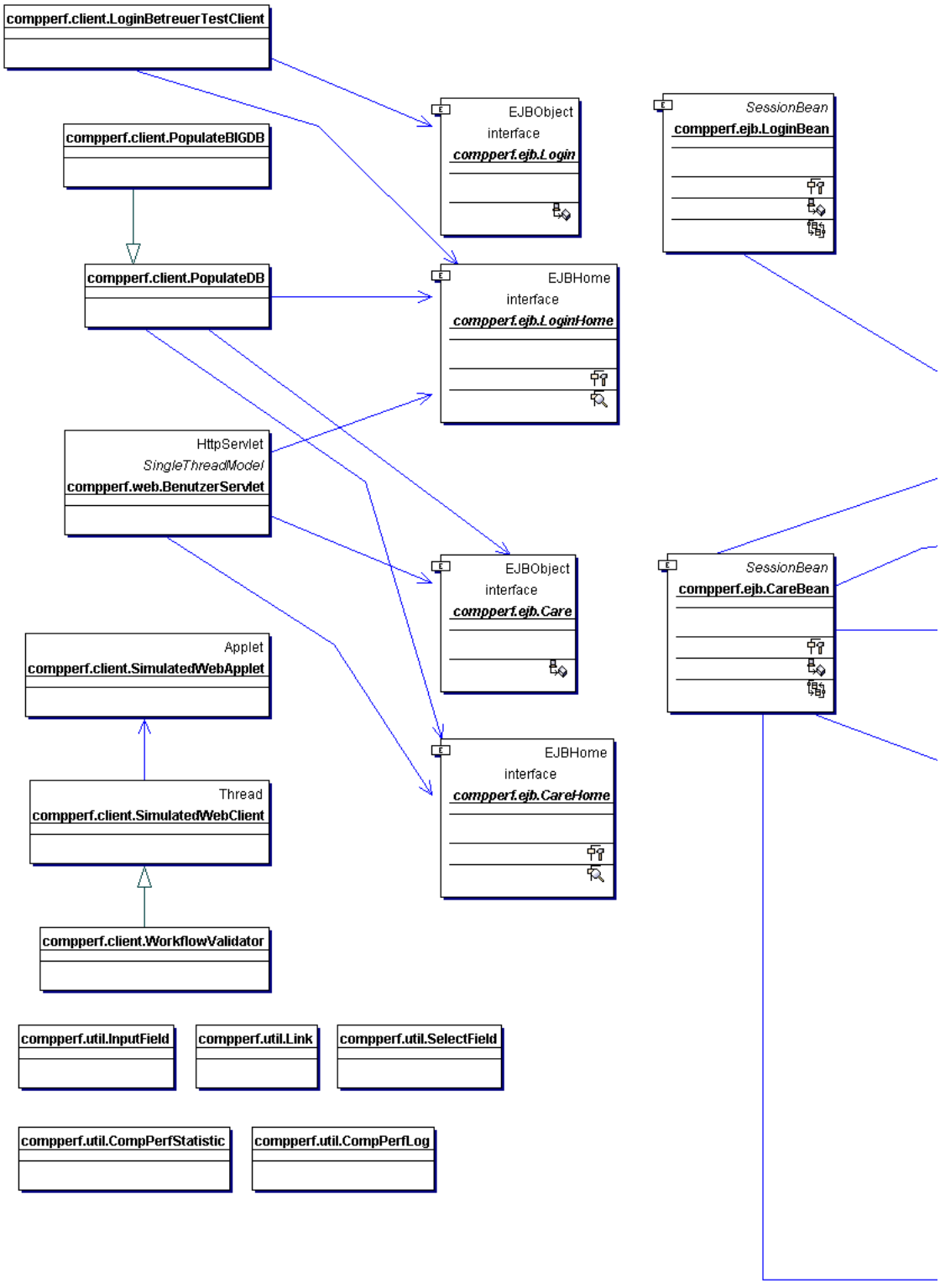
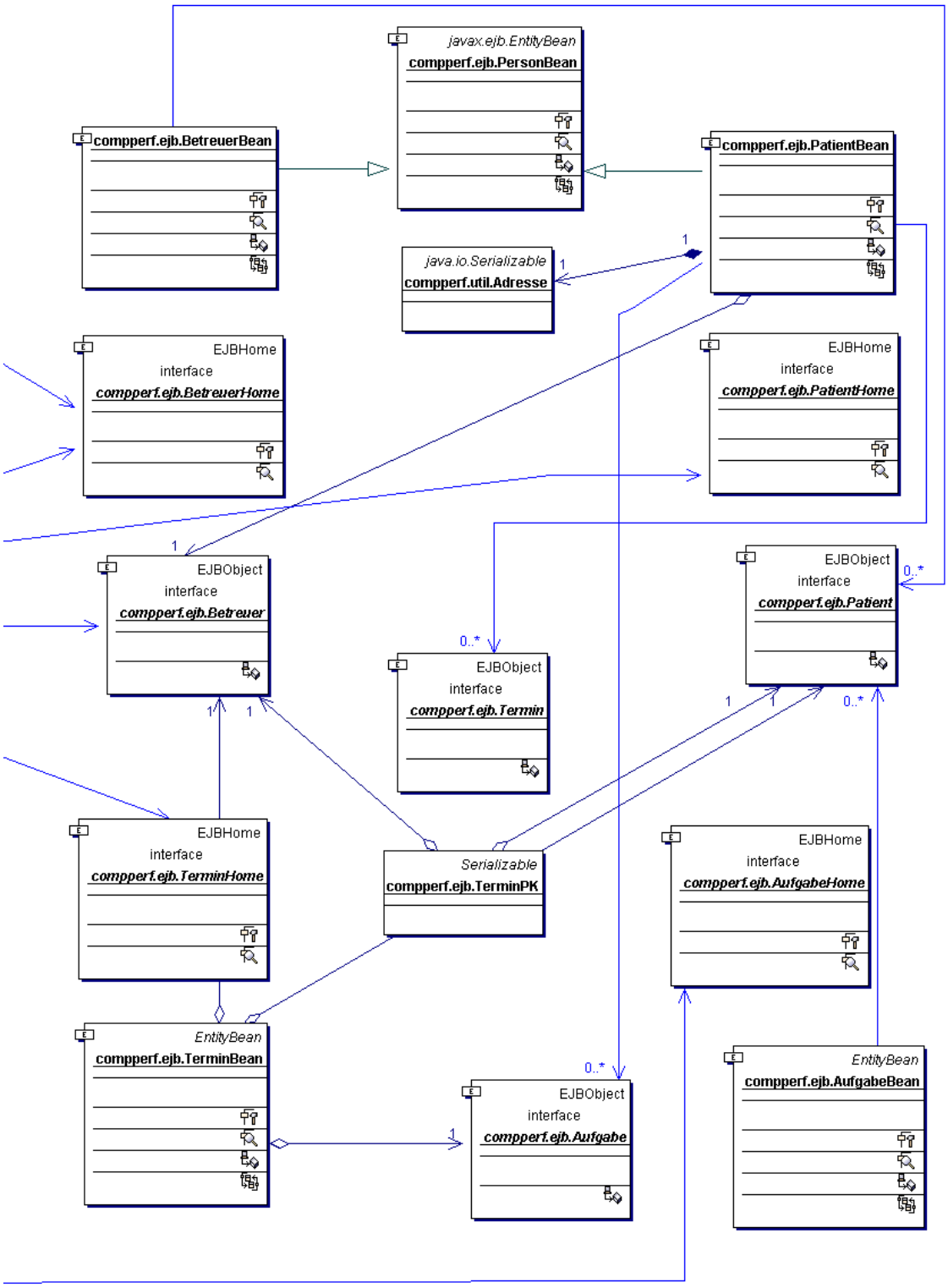


Abbildung 31: Klassendiagramm


javax.ejb.EntityBean
compperf.ejb.PersonBean
compperf.ejb.BetreuerBean
compperf.ejb.PatientBean
java.io.Serializable
compperf.util.Adresse
1
1
EJBHome
interface
compperf.ejb.BetreuerHome
EJBHome
interface
compperf.ejb.PatientHome
1
EJBObject
interface
compperf.ejb.Betreuer
EJBObject
interface
compperf.ejb.Patient
0..*
0..*
EJBObject
interface
compperf.ejb.Termin
1
1
1
1
0..*
EJBHome
interface
compperf.ejb.TerminHome
Serializable
compperf.ejb.TerminPK
EJBHome
interface
compperf.ejb.AufgabeHome
EntityBean
compperf.ejb.TerminBean
0..*
EJBObject
interface
compperf.ejb.Aufgabe
1
EntityBean
compperf.ejb.AufgabeBean

Testaufbau und Messungen

Für alle Messungen wurde die folgende Hard- und Softwarekonfiguration verwendet:

1. Datenbankserver (333MHz, 128MB RAM, Windows 2000 mit SQL Server 2000)
2. Applikationsserver (500MHz, 256MB RAM, Windows 2000 mit BAS 4.5)
3. Clientrechner (derselbe, auf dem auch der Application Server lief)

Die Rechner waren durch ein 10MBit Ethernet verbunden. Eine Clusterlösung konnte mangels eines zusätzlichen Rechners nicht getestet werden. Aus dem schon erwähnen TCP-W-Benchmark wurden folgende Meßgrößen übernommen:

- *WIRT* bezeichnet die Web Interaction Response Time. Dabei handelt es sich um das Zeitintervall zwischen Senden der Anfrage und dem vollständigen Erhalt der Antwort. Die WIRT kann für jede Seite im Navigationsworkflow einzeln berechnet werden. Sie wird zwangsläufig zwischen den einzelnen Typen stark differenzieren.
- *WIPS* bezeichnet die Web Interactions Per Second und ist damit ein Maß für die Anzahl von Anfragen, die ein Server in einer Sekunde beantworten kann. Sie ist aussagekräftiger für den gesamten Systemdurchsatz, da sie im allgemeinen über die gemittelte WIRT aller Clients berechnet wird.

Für die Versuchsreihen wurden Datenbanken mit 3 verschiedenen Füllgraden angelegt:

	#Betreuer	#Patienten	#Aufgaben	#Termine
Datenbank A	10	100	10	1.000
Datenbank B	20	200	20	4.000
Datenbank C	50	500	50	25.000

Tabelle 8: Füllgrad der Testdatenbanken

Zuerst wurde gemessen, wie sich eine steigende Zahl von Abfragen auf die WIPS auswirkt. Dazu wurde die Datenbank A mit einem einzelnen Client und einer Ruhezeit zwischen den Abfragen von 100ms verwendet.

#Requests	100	200	400	800	1600	3200	6400
Anwortzeit (in s)	26	47	130	291	602	1332	2786
WIPS	3,85	4,26	3,08	2,75	2,66	2,40	2,30

Tabelle 9: WIPS für einen Client

Wie aus der Tabelle ersichtlich ist, schwingt sich die WIPS nach einer kurzen Zeit auf einen annähernd konstanten Wert ein.

Aus obigen Testläufen wurde auch die Verteilung der Antwortwartezeiten auf die einzelnen Typen von Anfragen ermittelt (WIRT).

Typ	Aktion	∅Antwort-zeit (in ms)	Typ	Aktion	∅Antwort-zeit (in ms)
0	Startseite	419	10	Aufgabenliste	628
1	Betreuerliste	410	11	Neuer Termin	611
2	Neuer Betreuer	262	12	Termin hinzufügen	493
3	Betreuer hinzufügen	446	13	Termin löschen	181
4	Betreuer löschen	2235	14	Termin ansehen	218
5	Betreuer anmelden	532	15	Neue Aufgabe	262
6	Neuer Patient	268	16	Aufgabe hinzufügen	246
7	Patient hinzufügen	216	17	Aufgabe löschen	1563
8	Patient löschen	382	18	Aufgabe ansehen	254
9	Patient auswählen	727			

Tabelle 10: Aktionstypen im Workflow

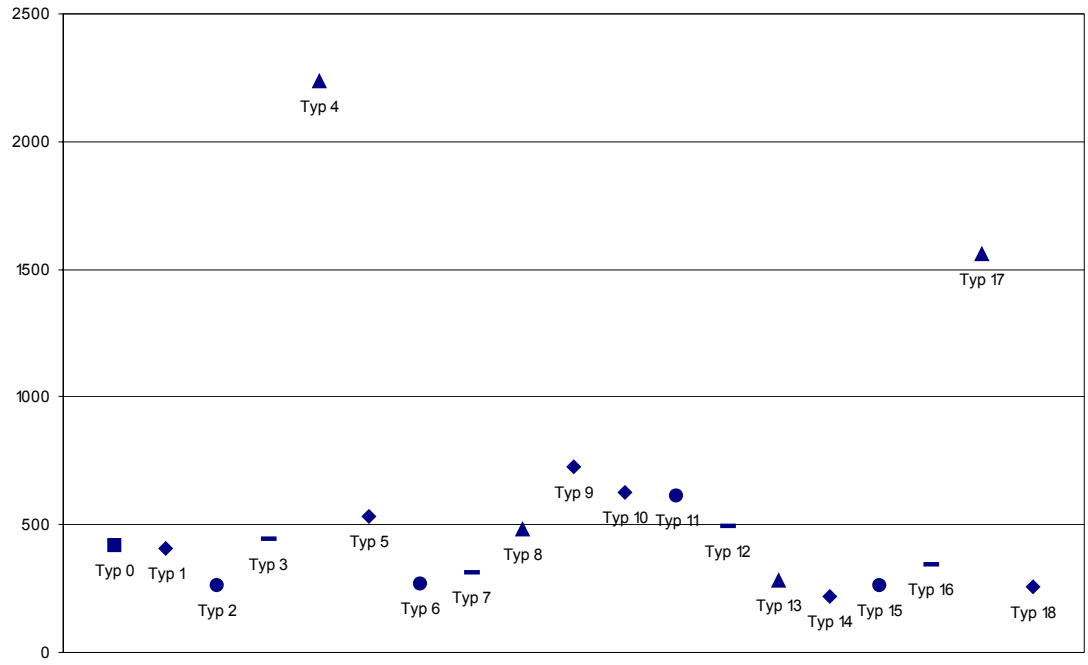


Abbildung 32: WIRT für alle Typen (in ms)

Wie aus dem Diagramm ersichtlich ist, hängt die Antwortzeit stark vom Typ der Anfrage ab. Dabei liegen statische HTML-Seiten (Typ 0) nur im Mittelfeld.

Neu (Kreis): Diese Operation bezeichnet den Aufruf eines HTML-Formulars mit Eingabefeldern. Sie rufen keine Session-Bean-Methoden auf. Eine Ausnahme ist „Termin hinzufügen“ (Typ 12). Bei dieser Aktion werden in einer Auswahlliste alle Aufgaben angezeigt, die

als Termin hinzugefügt werden können. Daher ist dieser Typ in seiner Klasse am langsamsten.

Hinzufügen (Strich): Dabei wird ein neuer Datensatz in die gewünschte Tabelle eingefügt. Die Zeiten unterscheiden sich nicht stark.

Löschen (Dreieck): Das Entfernen von Datensätzen ist am teuersten. Dies gilt vor allem für die Objekte, deren Löschen weitere Objekte entfernt (siehe auch „Kaskadierendes Löschen“ im Anhang), z.B. bei Betreuern und Aufgaben.

Ansehen (Karo): Beim Ansehen gibt es 2 Gruppierungen, die erste beinhaltet die Ausgabe einer Liste (Typen 1, 5, 9 und 10) und benötigt deshalb mehr Zeit als das Ausgeben eines einzelnen Datensatzes (Typen 14 und 18).

In einer zweiten Meßreihe wurde der Einfluß einer steigenden Clientzahl auf die Abarbeitungsgeschwindigkeit von 100 Anfragen untersucht. Es wurden die Datenbanken A, B und C benutzt. Die Ruhezeit der Clients betrug 100ms. Es ergab sich folgendes Bild:

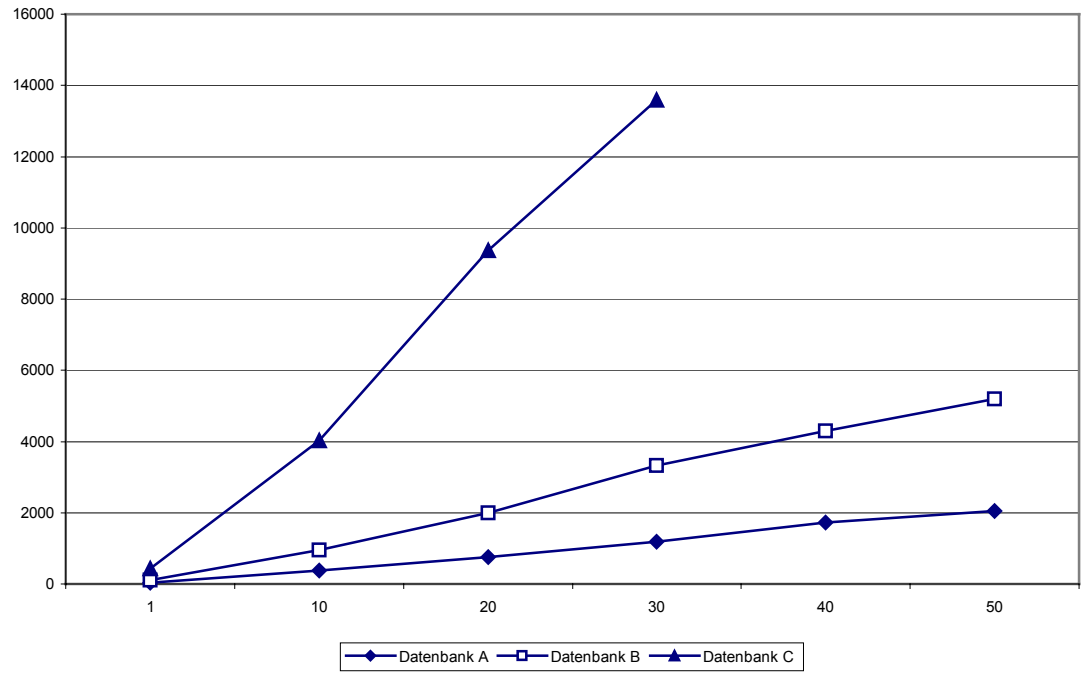


Abbildung 33: Abarbeitungszeit für 100 Anfragen (in s)

Erwartungsgemäß steigt die benötigte Abarbeitungszeit mit zunehmender Clientzahl stark an. Zumindest bis zu der hier verwendeten Zahl von 50 parallelen Clients erfolgt dieser Anstieg jedoch beinahe linear, d.h. die Ausführungszeit von n Clients entspricht in etwa der

n-fachen Ausführungsdauer eines Clients. Ob es aber bei noch größerer Clientzahl nicht doch zu einem exponentiellen Anstieg kommt, kann daraus nicht geschlossen werden.

Clients	1	10	20	30	40	50
Datenbank A	28	372	751	1179	1720	2055
Datenbank B	104	945	2000	3325	4305	5204
Datenbank C	431	4021	9360	13583		

Tabelle 11: Antwortzeit (in ms)

Einen viel größeren Einfluß auf die Antwortzeit hat aber die gewählte Größe der Datenbank. In der Tabelle läßt sich ein Faktor ablesen, der nur gering unter dem Faktor liegt, um den sich der Füllgrad der Datenbank voneinander unterscheidet. Die WIPS für die einzelnen Datenbanken sind in folgender Abbildung dargestellt.

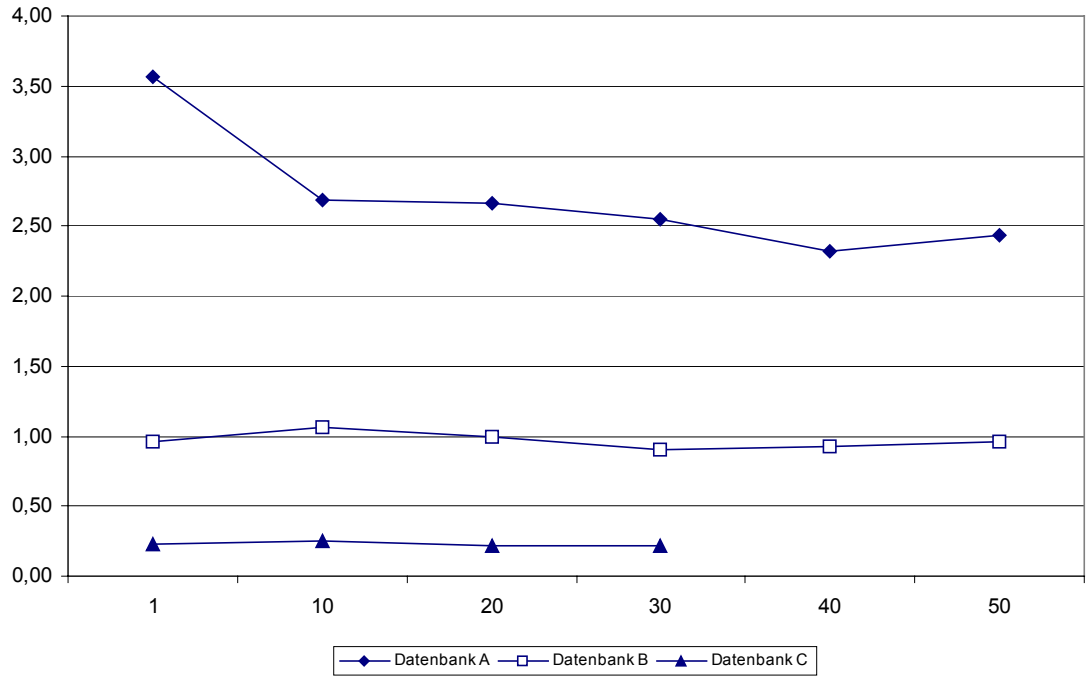


Abbildung 34: WIPS für die Datenbanken A, B und C

Abschließend wurden noch untersucht, in wieweit eine Änderung des Isolationslevels beim Zugriff auf die Datenbank die Ausführungsgeschwindigkeit beeinflußt. Dazu wurden die unterstützten Transaktions-Modi NONE, READ_UNCOMMITED, READ_COMMITED, REPEATABLE_READ und NONE_SERIALIZABLE je mit 20 und 50 Clients bei einer Wartezeit von 100ms und 100 gleichzeitigen Clients getestet. Erstaunlicherweise lagen die Unterschiede allerdings im Rahmen der Meßungenauigkeit.

6.9. Diskussion der Meßresultate

Jede Messung wurde dreimal durchgeführt und daraus der Mittelwert berechnet. Die Meßergebnisse zeigten in einigen Fällen eine starke Varianz der benötigten Zeit bei gleichen Ausgangsdaten. Dies läßt sich zum Teil darauf zurückführen, daß der Typ der Abfragen zufallsbestimmt wurde, so daß eventuell ein Testlauf mehr Löschoperationen ausgeführt hat als ein anderer. Interessant wäre es, die Wartezeiten auf eine Verteilungsfunktion mit zeitlichen Intervallen abzubilden. Dann ließen sich Extremwerte erkennen, die das arithmetische Mittel beeinflußt haben.

Für eine hinreichend genaue Bestimmung der Ausführungszeit ist die Zahl der durchgeführten Testläufe zu gering. Eine Meßreihe benötigt jedoch gerade im Fall der größeren Datenbanken mehrere Stunden und es war daher auch aus Zeitgründen nicht möglich, noch umfangreichere Tests vorzunehmen. Die Ergebnisse sind aber gut für eine ungefähre Abschätzung geeignet, zumal in Betracht zu ziehen ist, daß in realen Einsatzumgebungen einflußnehmende Faktoren wie der Netzwerkdurchsatz und die Prozessorauslastung aufgrund anderer Programme oder Benutzer nicht über einen längeren Zeitraum konstant gehalten werden können.

Auf den ersten Blick scheint es bei einer WIPS zwischen 0,3 und 2,5 nicht ratsam, größere Projekte mit der vorgestellten Beispielkonfiguration verwirklichen zu wollen. Dabei muß man jedoch bedenken, daß eine Wartezeit von 100ms zwischen zwei Anfragen bei menschlichen Benutzern unrealistisch niedrig ist. Erhöht man dagegen den Wert auf 10 Sekunden, kann man auch eine ca. 100 Mal größere Zahl von Clients bedienen.

Der geringe Einfluß der Isolationsstufe mag an der geringen Verknüpfung der Objekte auf Ebene der Datenbank liegen (siehe Tabelle Datenbankschema). Des weiteren sind die im Beispiel vorkommenden Transaktionen kurz und beziehen nur wenige Beans ein.

Für die Implementierung von J2EE-Applikationen läßt sich der Schluß ziehen, daß die Ergebnismengen von Suchoperationen auf Enterprise-Beans nicht zu groß gewählt werden dürfen. In der EJB-Spezifikation gibt es keine Möglichkeit festzustellen, ob die Attribute eines Beans in einer Transaktion tatsächlich verändert wurden. Viele EJB-Container führen daher auch am Ende reiner Lesetransaktionen ein UPDATE auf der Datenbank aus, was unnötig zu Lasten der Performanz geht.

Rückblickend läßt sich sagen, daß die Hardwareressourcen einen entscheidenden Einfluß auf die Messungen ausübten. Damit relativieren sich die Aussagen über das Verhalten des Application Servers dahingehend, als daß auch die Performanz des Datenbankrechners und des Webservers das Ergebnis beeinflußt haben. Messungen mit mehr als 50 Clients waren beispielsweise nicht möglich, da dabei der Client zu stark ausgelastet wurde, um die Anfra-

gen schnell genug abzusetzen und die Ergebnisse zu parsen. Auch die Zuweisung von mehr Hauptspeicher für den EJB-Container könnte den Objektpool von EJBs vergrößern und dadurch den Antwortdurchsatz drastisch steigern.

Die Stabilität des Servers war im Test äußert befriedigend. Es gab während des gesamten Verlaufs keinen echten Absturz. Nur in einem einzigen Fall blieb der Server mit einer Bugmeldung der virtuellen Maschine hängen.

Die EJB-Architektur konnte die an sie gestellten Anforderungen erfüllen. Durch die inzwischen ausgereifte Integration mit Entwicklungstools wie JBuilder entfällt der langwierige Vorgang des Anpassens der Deployment Deskriptoren u.ä.

Die größten Vorteile (verglichen mit einer ohne die J2EE-APIs programmierten Version) zog des Projekt aus folgenden Punkten:

1. Der implizit vorhandenen *Verteilbarkeit*. Außer dem Erhalt einer Referenz auf die Bean-Interfaces unterscheidet sich der Code nicht vom Aufruf lokaler Objekte.
2. Der automatischen *Persistenzabbildung*. Der Vorteil einer deklarativen Abbildung von Objekten auf Tabellen wird spätestens bei Änderungen am Objektmodell oder bei Änderungen der Datenbank deutlich.
3. Dem *Ressourcenmanagement*. Die Bereitstellung von Datenbank- oder Netzwerkverbindungen erfolgt transparent.
4. Der integrierten *Transaktionalität*. Eine eigene Transaktionssteuerung ist nicht nötig.

Das Beispiel besteht aus 33 Klassen und ca. 6400 Codezeilen (mit Kommentaren, aber ohne den generierten IIOP-Code), die sich etwa zur Hälfte auf die Enterprise-Beans und das Servlet, zur anderen Hälfte auf die Testclients verteilen. Es konnte in ca. 40 Personentagen implementiert werden. Dabei sind die Leistungstests des Application Servers noch nicht enthalten. Der größere Teil der Zeit war allerdings für die Entwicklung der Testclients nötig. Ein ähnliches Beispiel auf Basis von CORBA oder COM hätte erfahrungsgemäß um einiges länger gedauert.

Zusammenfassung Kapitel 6

Die hier verwendete Konfiguration ermöglicht auch mit einer eingeschränkt leistungsfähigen Hardware eine performante Programmausführung. Steigende Clientzahlen führen zumindest im untersuchten Bereich nicht zu einem überproportionalen Anstieg der Abarbeitungszeit. Der Einfluß der Datenbankgröße ist dagegen deutlich meßbar. Obwohl die Tests nur rudimentärer Art sind, kann man doch davon ausgehen, daß für Mobtel kein völlig abweichendes Verhalten zu erwarten ist. Die Ausführungszeit für zusätzlichen Java-Code in den verwendeten Methoden ist im Vergleich zur Zeit, die für die Durchdringen der Verhüllschichten des Containers und für Instantiierung der EJB benötigt wird, gering.

7. Ergebnisse und Ausblick

Die vorliegende Arbeit besaß als eine der Zielstellungen, eine *Middlewareplattform* als Basis für das Betreuungssystem Mobtel auszuwählen. Dazu wurden drei verschiedene Möglichkeiten evaluiert: Der Einsatz von Enterprise JavaBeans, CORBA und COM basierten Application Servern. Die Entscheidung viel zu Gunsten von EJB, da diese aufgrund der Verwendung der Programmiersprache Java plattformunabhängig sind. Ferner ist ihr Einsatz zukunftssicher, da EJB von einer überwältigenden Zahl von Herstellern unterstützt werden. Die große Anzahl standardisierter Schnittstellen für den Zugriff auf Dienste wie Datenbanken, Transaktionsverwaltung, Sicherheit und Verteilbarkeit erbringt eine konzeptionelle Trennung von einer ausgewählten Implementierung, so daß Mobtel nicht auf ein bestimmtes Basisprodukt festgelegt ist. Der Einsatz des Borland Application Servers führte zu einer ersten Umsetzung der Aufgabe.

Der zweite Schwerpunkt lag in der Untersuchung von *Persistenzmöglichkeiten* für die von Mobtel verwendeten Datenobjekte. Dabei hat sich gezeigt, daß der ursprünglich avisierte Einsatz eines objektorientierten Datenbankmanagementsystems (POET) für die vorliegende Architektur ungeeignet war. Die Hauptprobleme lagen dabei in der fehlenden Anbindung an vorhandene Application Server bzw. der konzeptionellen Ausrichtung von POET auf lokalen Einbenutzerbetrieb. Des weiteren birgt der Einsatz einer relationalen Datenbank nicht die Gefahr, aufgrund des proprietären Datenformats zukünftig von einem Produkt abhängig zu sein. Der Zugriff mit Hilfe von SQL-Klauseln und JDBC stellt begründeterweise den Standard für Datenbankoperationen und Persistenz in Java dar.

Die bei der Abbildung von Java-Objekten auf Datenbanktabellen auftretenden Probleme wurden erläutert und Lösungen aufgezeigt. Außer dem höheren Aufwand bei dem Design der Datenbank existieren keine konzeptbedingten Nachteile für den Einsatz von relationalen Datenbankensystemen. Die hohe Abarbeitungsgeschwindigkeit von Anfragen, die große Zahl von Verwaltungswerkzeugen und die Zukunftssicherheit des SQL-Standards sprechen für ihre Verwendung.

Die Arbeit ergab ferner eine enge Korrelation zwischen *Skalierbarkeit* und *Verfügbarkeit* bei EJB Application Servern. Eine J2EE-Applikation kann auf mehrere Serverinstanzen verteilt werden. Viele Hersteller bieten Mechanismen für einen automatischen Lastausgleich und für die Umleitung von Anfragen im Fall des Versagens einer Instanz. Für den Client ist diese Vorgehensweise transparent.

Letztlich wurde an einem *praktischen Beispiel* ansatzweise gezeigt, daß sich die Projektvorgaben von Mobtel in einer EJB-Infrastruktur umsetzen lassen und genügend Leistungsreserven für den Einsatz in einer realen Anwendungsumgebung bieten.

Ausblick

In einer weiterführenden Arbeit könnte das praktische Skalierungsverhalten und das Clustering des EJB-Servers nach der theoretischen Betrachtung und die damit verbundenen Auswirkungen an konkreten Beispielen untersucht werden. Ebenfalls wäre es interessant zu testen, welche Probleme bei der Portierung einer J2EE-Anwendung auf einen der in Kapitel 4.6.2 vorgestellten freien Application Server auftreten. Die Verwendung einer vollständig als Open Source verfügbaren Plattform inklusive Betriebssystem und Datenbank könnte eine leistungsfähige Ausgangsbasis für weitere Projekte darstellen.

Unabhängig vom Inhalt dieser Arbeit wären auch eine Reihe von Zusatzfunktionen denkbar, die den potentiellen Anwenderkreis für Mobtel erhöhen bzw. den PMA stärker in die Rehabilitation integrieren. Beispielsweise wäre es von Vorteil, auf Seiten der mobilen Betreuungsregion nicht auf einen einzelnen Gerätetyp für den PMA festgelegt zu sein. Zukünftig könnten leistungsfähigere Geräte, die ein gewisses Maß an Programmierbarkeit besitzen, eine Alternative darstellen. Das Betreuungssystem könnte ferner aus den gesammelten Log-Dateien ein Verhaltensmuster für einen Patienten aufbauen und durch einen adaptiven Lernprozeß gezielter auf Alarme reagieren. Dazu wäre es allerdings notwendig, auch andere an der Behandlung beteiligte Personen wie z.B. die Ärzte oder die den Patienten privat betreuenden Personen einzubinden, weiterhin auch Institutionen wie Krankenkassen und Apotheken. Für diese müßten eigene Bedienoberflächen entworfen werden. Durch die Einbeziehung von Öffnungszeiten dieser Institutionen und Fahrplänen öffentlicher Verkehrsmittel im Stadtgebiet könnte eine effiziente Wegplanung für den Patienten entworfen werden, die eine ihn ständig begleitende Person ersetzen kann, da bei Notfällen alternative Routen errechnet und vorgeschlagen werden. Auf lange Sicht wird ein solches Informationsportal für eine große Zahl älterer Bürger interessant sein und ihnen bei der Tagesplanung helfen.

Danksagung

Ich möchte meinem Betreuer Hendrik Schulze für seine Geduld bei der Anfertigung dieser Arbeit danken.

Literaturverzeichnis

[BER] Berg, C. „*Advanced Java2 Development for Enterprise Applications*“
2. Auflage SUN Press 2000

[BR 00] Brock, D. *„A Recommendation for High-Availability"* TPC Org.
http://www.tpc.org/information/other/articles/ha.asp

[DOM 99] Domajnko, T. et al *„How to Store Java Objects"*
Java Report April 1999

[FOW 98] Fowler, M.; Scott, K. *"UML konzentriert"*
1. Auflage Addison-Wesley 1998

[GH 99] Gebauer, J.; Hartmann, H.; Seguin, M. *„Clustering mit Windows NT"*
1. Auflage Addison-Wesley, Bonn 1999

[IRM 99] Irmscher, K. et al *„Mobile Einsatzszenarien von Telemedizin bei der logischen Therapie hirngeschädigter Patienten mit Gedächtnis- und Funktionsstörungen"*
Universität Leipzig, Dezember 1999

[IRM 01] Irmscher, K. et al „*Verbundprojekt Mobtel - Abschlußbericht*“
Az.: 4-7533-70-0361-98/16 Universität Leipzig, Juni 2001

[JNET] JnetDirect Inc.: JSQLConnect V2.24
http://www.j-netdirect.com/Downloads.htm

[MOB 98] Projekt Telemedizin *„Anforderungen für Mobtel-Funktionen"*
Universität Leipzig, Juni 1998

[MOU 99] Mougin, Ph. *„EJBs from a critical perspective"*
TrendMarkers Dezember 1999

[PER 00] Perrone, P.; Chaganti, V. *"Building Java Enterprise Systems with J2EE"*
1. Auflage SAMS 2000

[POE 99] POET Java SDK Programmer's Guide Version 5.1

[REI 00] Reiners, W. *"IT-Projekte in der Medizin"*
Java Spektrum 4/ 2000

[ROM 99] Roman, E. *"Mastering EJB"*
1. Auflage Wiley&Sons 1999

[ROS 99] Roßbach, P.; Schreiber, H. "*Java Server and Servlets"*
1.Auflage Addison 1999

[SAA 00] Saake, G.; Sattler, K.-U. *"Datenbanken & Java: JDBC, SQLJ und ODMG"*
1. Auflage dpunkt.verlag 2000

[SCH 00] Schulze, H.; Irmscher K.: *"A Mobile Distributed Telemedical System for Application in the Neuropsychological Therapy"*
USM 2000 in "Trends in Distributed Systems: Towards a Universal Service Market" LNCS Nr. 1890, Springer pp. 176-187

[STA 00] Stal, M.: "*Die Komponententechnologien COM+, EJB und CORBA Components*"
Objektspektrum SIGS 2000

[SUN 00] SUN Microsystems *"Designing Enterprise Applikations with the Java 2 Plattform EE 1.0.1",* Oktober 2000

[SUN 01] SUN Microsystems *"PJama: A Prototype Implementation of Orthogonal Persistence for the Java platform"* September 2000
http://www.sun.com/research/forest/

[TPCW] TPC-W Benchmark
http://www.tpc.org/tpcw/default.asp

[WC 98] Wunderlich, J.; McCarthy, C.: *"Making Application Highly Available"* Richardson System Software Lab
http://www.interrex.org/conference/IWorks98/sessions/sn036/paper.html

Anhang A – Anlagen zu Compperf

A.1. Installation und Kompilierung der Beispiele

Alle Beispieldateien wurden mit Borland JBuilder 4 EE erstellt. Zum Kompilieren ist dieser jedoch nicht nötig, es müssen lediglich die Visigenic-Dateien (vbj, vbjc) verfügbar sein, die auch mit dem BAS installiert werden. Im Hauptverzeichnis *compperf* befindet sich ein Make-file. Bevor diese genutzt werden kann, ist die Datei *setEnv.bat* anzupassen. In dieser sind die Pfade für das JDK und den Application Server einzutragen.

Nach dem Aufruf von *make.bat* werden alle Java-Dateien kompiliert, der IIOP-Code erzeugt, die Dateien zur JAR-Archiven zusammengefaßt und das Programm auf den Server aufgespielt. Die Datei *clear.bat* löscht alle generierten Klassen und den IIOP-Code.

A.1.1. Voraussetzungen

Zur Installation des Beispielprogramms müssen folgende Voraussetzungen erfüllt sein:

1. Ein J2EE-kompatibler Application Server, bevorzugt der für die Tests eingesetzte Borland Application Server 4.5.
2. Eine SQL-Datenbank. Leider arbeitet das Beispiel nicht korrekt mit der integrierten JDataStore-Datenbank zusammen, da es ab einer gewissen Anzahl von Zugriffen zu Lock-Timeouts kommt.
3. Ein JDBC-Treiber für den Zugriff des Application Servers auf die Datenbank.

Für alle Tests wurde folgende Konfiguration verwendet:

1. Borland Application Server 4.5
2. SQL Server 2000
3. JSQLConnect von j-netdirect [JNET], ein Typ4-Treiber. Speziell für den Zugriff auf Datenbanken mit TDS-Protokoll entwickelt, ist keine besondere Konfiguration nötig.

A.1.2. Konfiguration

Für die Installation des Borland Application Servers sind keine speziellen Einstellungen nötig.

Auf dem Datenbankrechner muß eine neue Datenbank angelegt werden. Das Compperf-Beispiel benötigt ein einfaches Schema mit 5 Tabellen. Zu deren Erzeugung existiert das Skript *createTables.sql*.

Nach dem Start des BAS wird der JDBC-Treiber aufgespielt. Dies kann mittels der Application Server Console geschehen. Im allgemeinen handelt es sich dabei clientseitig um eine JAR-Datei. Man wählt *Tools->J2EE Deployment Wizard* und dann die Treiberdatei aus. Danach muß in Schritt 4 noch ein Application Server im lokalen Subnetz angegeben werden. Wenn der Vorgang erfolgreich war, kann man die Bibliothek unter dem Punkt *Installed Libraries* sehen.

Die Beispielanwendung liegt fertig in Form der Datei *compperf.ear* vor. Sie enthält sowohl das Archiv mit den Enterprise JavaBeans (*compperf_ejb.jar*) für den EJB-Container als auch die Web-Applikation (*compperf_web.war*). Beim Aufspielen ist auf jeden Fall eine Anpassung des Deployment Deskriptors nötig. Dazu muß in Schritt 3 des J2EE Deployment Wizards der Button *DD Editor* gedrückt werden. Hier können im Menüpunkt *JDBC 1 DataSources* die Verbindungseigenschaften für *datasources/CompPerfDataSource* geändert werden. Dabei bleibt der JNDI Name unverändert, die URL der Form:

jdbc:<subprotokoll>://<hostname>:<port>/<datenbankoptionen>

muß auf die gewünschte Datenbank zeigen, wobei das Subprotokoll vom verwendeten Datenbanktreiber und die Optionen vom verwendeten Datenbanktyp abhängen. Weiterhin muß man über einen gültigen Benutzernamen verfügen, sowohl zum Zugriff auf den Datenbankrechner als auch für die eigentliche Datenbank. Der Name der Datenbanktreiberklasse ist spezifisch für jeden Treiberhersteller. Sie befindet sich aber in dem zuvor aufgespielten JAR-Archiv und endet im allgemeinen auf *Driver*. Der Rest des Deployvorgangs bedarf keines Eingriffes mehr. Nach Abschluß sollte unter

http://localhost:8080/compperf/index.html

die Startseite der Mobtel-Demo erscheinen.

A.1.3. Füllen der Datenbank

Zu diesem Zeitpunkt existieren in der Datenbank noch keine Werte. Um die Tabelle zu füllen, existiert das Programm *PopulateDB*. Es befindet sich im Verzeichnis *classes/compperf/client* und wird von der Shell aus gestartet. Man kann auch direkt die Batchdatei *runPopulateDB.bat* ausführen. Wenn sich der Application Server nicht auf dem lokalen Rechner befindet, muß zuvor osagent.exe gestartet werden. PopulateDB kann über die Datei */classes/compperf/client/populateDB.properties* konfiguriert werden. Dabei gibt es folgende Einstellungen:

- *betreuerFile* : Gibt die absolute Position der Datei an, die die Betreuernamen enthält. Es werden Beispieldateien im Verzeichnis */dbvalues* mitgeliefert.
- *patientenFile*: Selbiges für Patienten. Das Format einer Zeile in der Datei besteht aus: <Name>@<Straße Nr.>@Stadt@Postleitzahl.
- *aufgabenFile*: Selbiges für Aufgaben.
- *debug*: Wenn dieser Wert auf *true* gesetzt ist, wird das Ergebnis der Parse- und Einfügeprozesse ausgegeben.
- *sleep*: Bezeichnet eine Wartezeit zwischen 2 Einfügeoperationen in die DB. Da das Programm die Termine der Patienten fortlaufend eingibt, sollte sie mindestens 1000 ms betragen, um zu vermeiden, daß es zu einer Primärschlüsselverletzung kommt.

Des weiteren gibt es das Programm PopulateBIGDB. Mit ihm lassen sich Datenbanken beliebiger Größe erzeugen. Es erwartet als Kommandozeilenparameter die Anzahl der zu erzeugenden Betreuer. Es werden ebenso viele Aufgaben wie Betreuer erzeugt, weiterhin für jeden Betreuer 10 Patienten und für jeden Patienten ein Termin mit jeder Aufgabe. Alle Objekte haben das Format <objektname><nummer>, also z.B. *Betreuer1*. Eine Wartezeit zwischen dem Einfügen ist hier nicht nötig.

A.2. Funktionsweise der Testclients

LoginBeanClient ist ein einfacher J2EE-Client, der alle Betreuer ausgibt, die sich in der Datenbank befinden. Er erwartet keine Parameter. LoginBeanClient greift über eine Referenz auf LoginBean direkt auf Enterprise-Beans zu.

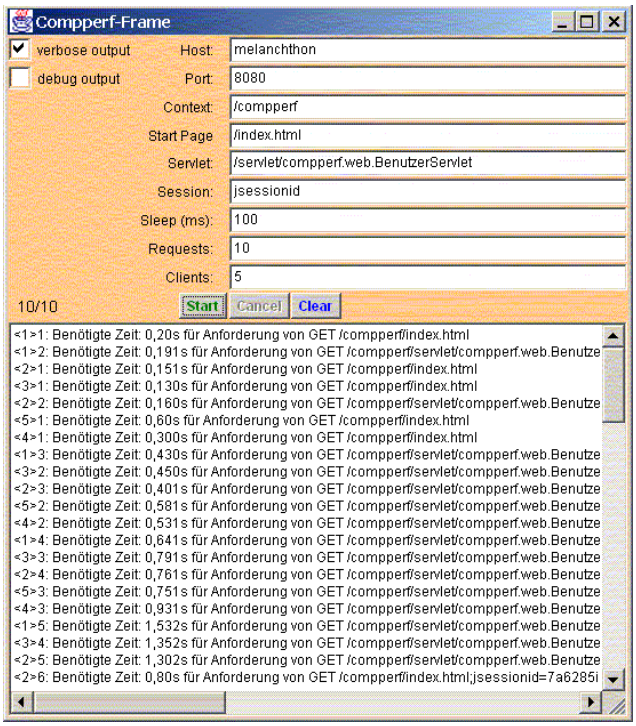
WorkflowValidator ist ebenfalls eine Kommandozeilenapplikation, die alle Wege im Navigationsworkflow einmal durchläuft. Im Gegensatz zu LoginBeanClient kommuniziert WorkflowValidator mit dem BenutzerServlet. Tritt während der Abarbeitung ein Fehler auf, wird dieser angezeigt. Wenn das Programm fehlerfrei ausgeführt wird, ist die Funktionstüchtigkeit der Web-Applikation gezeigt. WorkflowValidator setzt keine Werte in der Datenbank voraus und fügt auch über seine Ausführung hinaus keine hinzu.

SimulatedWebClient ist das Herzstück der TestClients. Bei Ausführung wird zuerst eine Konfigurationsdatei geladen. Deren Parameter sind folgende:

- *verbose*: Steht für eine detaillierte Ausgabe, welche URL angefordert wurde und wieviel Zeit dazu nötig war. Im Fehlerfall wird eine Meldung ausgegeben.
- *debug*: Gibt sehr ausführliche Informationen über die gelesene Antwort, die erkannten Eingabefelder, Links oder Buttons, Verbindungsparameter und ähnliches aus. Aufgrund der Vielzahl der Meldung ist der Modus nur zur Fehlersuche gedacht.
- *host*: Der Rechner, auf welchem der Webserver läuft.

- *port*: Bezeichnet den Port, an dem der HTTP-Dienst ausgeführt wird.
- *context*: Nach der J2EE-Spezifikation müssen alle Web-Applikationen innerhalb eines Kontextes laufen. Ein Kontext ist dabei mehr als ein Verzeichnis. Er definiert auch die Zugriffsmöglichkeit von Servlets untereinander. Man kann ihn als eine Art virtuellen Applikationscontainer verstehen.
- *startPage*: Die Startseite, mit der die Bearbeitung beginnen soll. Dieser Wert wird benötigt, da sich ein automatischer Client während seiner Abarbeitung auch vom Compperf-System abmelden kann und dann diese Datei zum Neueinloggen anfordert.
- *file*: Bezeichnet das Servlet, mit dem die Testclients interagieren. Aus Gründen der Einfachheit wurden aber alle Funktionen in einer Klasse zusammengefaßt. Die Zeichenkette muß mit „/servlet“ beginnen, um die Servlet-Engine aufzurufen.
- *sessionKey*: Variablenname, den die verwendete Servlet-Engine zur Kodierung der Session-ID nutzt. Dieser Parameter wird für das Sitzungsmanagement benötigt.
- *sleep:* Ist die Zeit in Millisekunden, die der Client zwischen dem Erhalt einer Antwort und einer neuen Anfrage deaktiviert ist, um ein menschenähnliches Verhalten zu simulieren
- *requests:* Gibt die Anzahl von Abfragen an, die der Client an den Server senden soll.

SimulatedWebApplet ist eine Java-AWT-Anwendung, die alle Eigenschaften von SimulatedWebClient steuern kann. Mit ihr lassen sich komfortabel auch größere Testreihen vornehmen. Die Eigenschaftsfelder sind dieselben wie bei SimulatedTestClient. Zusätzlich können auch mehrere Clients parallel als Threads gestartet werden.



Die Statusanzeige im Applet zeigt den Abarbeitungszustand des aktuellen Threads an. Bei mehreren Threads kann er voneinander abweichen und die Anzeige scheint rückwärts zu zählen. Das ist aber kein Fehler. Mit Druck auf den *Start*-Button wird die Abarbeitung gestartet, mit *Cancel* unterbrochen. *Clear* löscht die Ausgabe im Textfeld in der unteren Hälfte. Nach Beendigung aller Threads wird der Startbutton wieder aktiv.

Jede beantwortete Anfrage wird in einer Zeile ausgegeben. Dabei ist die Threadnummer in „<“ bzw. „>“-Zeichen gefaßt. Darauf folgt die Nummer der abgearbeiteten Anfrage, die benötigte Zeit und die angeforderte URL. Am Ende gibt jeder Thread die Gesamtzeit aus, die er mit der Abarbeitung beschäftigt war, sowie die Zeit, die er auf Antwort vom Server gewartet hat. Letztendlich wird eine Statistik ausgegeben. Sie beinhaltet unter anderem die für den gesamten Ablauf aller Threads benötigte Zeit, die Anzahl jedes Anfragetyp mit der durchschnittlichen Zeit zur Ausführung dieses Typs und die Zeit, die alle Clients zusammen warten mußten.

A.2.1. Sitzungsmanagement

Da HTTP ein zustandloses Protokoll ist, können aufeinanderfolgende Anfragen eines Clients diesem nicht zugeordnet werden, da der Server den Client nicht zweifelsfrei identifizieren kann. Um diese Einschränkung zu umgehen, wird eine eindeutige Kennung bei jeder Anfrage und Antwort gesetzt: die Sitzungsnummer. Dazu muß der Client Cookies akzeptieren, das sind kleine Datenpakete, die im temporären Speicher des Browser aufbewahrt werden. Diese werden bei jeder Anfrage mit zum Webserver gesendet. SimulatedWebClient speichert keine Cookies, nutzt aber ein ähnliches Verfahren, das sogenannte URL-Rewriting. Dabei wird die Sitzungsnummer als Parameterwert jeder Anfrage angefügt.

A.2.2. Wahrscheinlichkeiten

Um ein realistisches Nutzerverhalten nachzubilden, wurde für den Navigationsworkflow eine Wahrscheinlichkeitstabelle aufgestellt. So wird nicht jede Aktion gleich häufig ausgeführt. „Ansehen“ ist für alle Objekte die am zahlreichsten ausgeführte Aktion, „Löschen“ hängt dagegen stark vom Zielobjekt ab. Die Wahrscheinlichkeiten im einzelnen sind:

	Aufgaben	Betreuer	Patienten	Termine
Einfügen	5%	1%	5%	50%
Löschen	0,2%	0,5%	3%	25%
Ansehen	94,8%	98,5%	92%	25%

Compperf verfolgt das Prinzip des kaskadierendes Löschens. Dabei werden abhängige Datensätze entfernt, um Referenzen auf nicht mehr vorhandene Objekte zu vermeiden. Ter-

mine können ohne Auswirkung auf andere Datensätze gelöscht werden. Das Löschen eines Patienten löscht dagegen alle seine Termine. Wird ein Betreuers gelöscht, so werden auch alle Patienten gelöscht. Das Löschen einer Aufgabe bewirkt ebenfalls das Löschen aller Termine. Dies Einschränkung ließe sich aber prinzipiell umgehen, wenn man z.B. einen Anwendungsfall „Patient übernehmen“ realisiert.

A.2.3. Navigation im Workflow

Das Workflowsystem des Servlets ist sehr einfach aufgebaut. Es ordnet alle Anfragen zuerst anhand des immer vorhandenen versteckten Eingabefeld-Parameters *command* in grobe Kategorien ein, je nach Art des zu manipulierenden Objekts. Dort wird die Kennzeichnung des gedrückten Buttons als Unterkommando ausgewertet und in die entsprechende Aktion verzweigt.

A.3. Anlagen

A.3.1. Dokumentation

Die ausführliche Online-Dokumentation aller Klassen und Methoden befindet sich im Verzeichnis */doc*. Sie wurde mit Javadoc generiert und kann mit *index.html* gestartet werden. Eine PDF-Version ist im Hauptverzeichnis vorhanden.

A.3.2. Quellcode

Der Quellcode aller Programmdateien befindet sich im Verzeichnis */src*.

A.3.3. Restriktionen

1. „Diese Operation auf ... wird nicht unterstützt.“: Bei Benutzung des Web-Clients müssen zur Bestätigung einzelner Aktionen die zugehörigen Buttons mit der Maus angeklickt werden, ein Drücken der ENTER-Taste ergibt einen Fehler.
2. „Out-of-Memory“: Der EJB-Container im BAS 4.5 erhält eine feste Speicherzuteilung bei Programmstart. Dadurch kann es zu Speicherknappheit kommen, auch wenn eigentlich noch genügend RAM verfügbar wäre. Man kann die Einstellungen in *ias.config* im Verzeichnis *<appserver>/var/servers/<name>/admin/properties* ändern, z.B. in `vmparam -Xmx64m`.
3. „Leeren“ der Datenbank. Während des automatischen Testablaufs werden Datensätze hinzugefügt und gelöscht. Während beim Anlegen eines neuen Betreuers aber nur ein Datensatz angelegt wird, werden beim Löschen eines Betreuers alle Patienten und deren Termine gelöscht. Obwohl versucht wurde, dieses Verhalten durch die Wahrscheinlichkeitstabelle zu verringern, läßt sich dieser

die Wahrscheinlichkeitstabelle zu verringern, läßt sich dieser Effekt nicht vollständig vermeiden. Das Einfügen vieler neuer Betreuer hat nämlich den Nachteil, daß viele Betreuer mit wenigen oder keinen Patienten in die Datenbank aufgenommen werden. Das kann aber die Meßergebnisse verfälschen, da wie gezeigt die Operationszeit von Abfragen auch von der Menge der übertragenen Datensätze abhängt.

4. Bei der Verwendung mehrerer simultaner Clients kann es zu Fehlern kommen, die auf die gleichzeitige Abarbeitung zurückzuführen sind. Dazu gehört, daß sich mehrere Threads in der Rolle desselben Benutzers anmelden können und einer von ihnen einen Patienten löscht, während ein anderer für diesen gerade einen Termin hinzufügt. In diesem Fall wird eine ObjectNotFoundException aufgeworfen und der Clientthread neu gestartet. Ein anderer Fall ist das gleichzeitige Anlegen eines neuen Patienten. Da der automatisch generierte Name durch die Systemzeit bestimmt wird, kann es zu Überschneidungen kommen, da der Zeitzähler nur eine Genauigkeit von ca. 50ms besitzt. Hier kommt es zu einer DuplikateKeyException und ebenfalls zum Neustart. Fehler sind um so wahrscheinlicher, je ungünstiger das Verhältnis von Betreuern zu simulierten Clients ist. Gibt es mehr Clients als Betreuer, melden sich mindestens 2 Clients unter demselben Betreuernamen an.

Anhang B – Abbildungs- und Tabellenverzeichnis

B.1. Abbildungen

B.2. Tabellen

Anhang C – Abkürzungen

2PC	Two Phase Commit	Sind in einer Transaktion mehrere Quellen (z.B. Datenbanken) involviert, muß ein Commit mindestens 2 Phasen umfassen.
AWT	Abstract Windowing Toolkit	Grafikbibliothek unter Java1.1. Bietet rudimentäre Zeichenfunktionen und Formularelemente.
BMP	Bean Managed Persistence	Die Persistenzabbildung eines EB wird von ihm selbst gesteuert, meistens durch Java-JDBC-Code. (siehe auch CMP)
CMP	Container Managed Persistence	Die Persistenzabbildung wird nicht von dem EB direkt vorgenommen. Der EJB-Container ruft statt dessen eine spezielle Mapping-Engine auf, die die Attribute und Assoziationen der Klasse unter Benutzung der im DD erfolgten Deklaration auf die Datenquelle abbildet.
DBMS	Datenbankmanagementsystem	Datenbankprogramm, wird auch häufig synonym mit Datenbank gebraucht
DD	Deploment Descriptor	Ein oder mehrere XML-Dateien, die zur Konfiguration von EJBs dienen, ohne daß diese neu kompiliert werden müssen. Enthält von SUN standardisierte Elemente, die die Interfaces, Referenzen usw. beschreiben sowie Herstellerabhängige Elemente, die bspw. den Zugriff auf Datenbanken und die CMP steuern.
EAR	J2EE Application Archive	Ähnlich zu JAR. Enthält alle zu einer Anwendung gehörenden EJB-JARs, Web-WARs und EJB-Client-JARs sowie einen weiteren DD.
EB	Entity Bean	Eine Komponente, die ein Geschäftobjekt repräsentiert und über das Ende der Anwendung hinaus existiert. (siehe auch SB)
EJB	Enterprise JavaBeans	Eine Spezifikation von SUN, die eine komponentenbasierte Programmierschnittstelle zur Entwicklung von Mehrschichtanwendungen beschreibt. Ist eine Untermenge von J2EE.
IIOP	Internet Inter ORB Protokoll	Kommunikationsprotokoll für Verteilte Objekte

J2EE	Java 2 Enterprise Edition	Eine auf der J2SE aufbauende Erweiterung von SUN. Enthält hauptsächlich APIs, die bei der Entwicklung betrieblicher Software benötigt werden, z.B. EJB, JSP, JDBC, JNDI, JTA, JavaMail, RMI-IIOP.
J2EE-RI J2EE-SDK	J2EE Referenzimplementation/ Software Development Kit	Von SUN zur Verfügung gestellte Softwarerealisierungen der J2EE-APIs. Dienen zu Einstiegs-, Test- und Kompatibilitätszwecken.
J2SE	Java 2 Standard Edition	Das «eigentliche» Java. Enthält die grundlegenden Sprachpakete sowie eine ganze Reihe von APIs, die i.a. Standardfunktionen abdecken, z.B. Applets, AWT, Swing, Collections, I/O, Netzwerk, JavaBeans, Mehrsprachigkeit usw.
JAR	Java Archive	Ein ZIP-kompatibles Komprimierungsformat, welches häufig für die Zusammenfassung einer großen Zahl für Java-Klassen verwendet wird. EJB werden i.a. mit ihren Hilfsklassen zu Jars gebündelt.
JDBC	Java Database Connectivity	Zugriffs-API für Datenbanken unter Java
JDK	Java Development Kit	Basis-Entwicklungssoftware für Java-Programme
JNDI	Java Naming and Directory Interface	Abstrakte API für einem Verzeichnisdienst
JVM	Java Virtual Machine	Eine Ablaufumgebung für Java-Programme. Da der Java-Compiler keinen prozessorabhängigen Maschinencode, sondern universellen Bytecode erzeugt, kann dasselbe Javaprogramm auf unterschiedlichen Hardwareplattformen und JVMs ausgeführt werden.
MML	Mobtel Markup Language	Eine Beschreibungssprache für Seiten des Mobtel-Pagers
MTBF	Mean Time Between Failure	Durchschnittliche Zeit zwischen 2 Fehlern
MTP	Mobtel Transport Protokoll	Eine Transportprotokoll für Seiten des Mobtel-Pagers
MTTR	Mean Time To Repair	Durchschnittliche Zeit für die Reparatur eines Systems
O2R	Object to Relational Mapping	Beschreibt den Vorgang des Abbildens von Java-Objekten in Tabellen Relationaler Datenbanksysteme. Für diesen nichttrivialen Vorgang gibt es eine Reihe eigener Programme.
ORB	Object Request Broker	Zentrale Instanz für die Verwaltung von Verteilten Objekten

RMI	Remote Method Invocation	Auf RPC aufsetzendes Protokoll zur Kommunikation entfernter Objekte mittels lokaler Stellvertreter. Nur für Java verfügbar.
RPC	Remote Procedure Call	Entfernter Prozeduraufruf, speziell über Rechnergrenzen
SB	Session Bean	Eine Komponente, die einen Client oder Anwendungsfall repräsentiert. Es gibt zustandslose und zustandsbehaftete, also zuordenbare SB.
SPOF	Single Point of Failure	Durch den Wegfall nur einer Komponente kann das Gesamtsystem ausfallen.
SSL	Secure Socket Layer	Verschlüsslungsmöglichkeit für HTTP-Verbindungen
UML	Unified Modeling Language	Abstrakte Beschreibungssprache für Objektmodelle auf Basis grafischer Piktogramme
WAP	Wireless Application Protocol	Transportprotokoll zum Übertragen von Hypertext auf Mobiltelefone u.ä.
WAR	Web Component Archive	Ähnlich zu JAR. Enthält alle Dateien, die zur Web-Schicht einer Anwendung gehören, also .html, .jsp oder .class von Servlets/ JavaBeans.
WIPS	Web Interactions Per Second	Anzahl von Webseiten, die ein Server je Sekunde generieren und zurücksenden kann
WIRT	Web Interaction Response Time	Zeitspanne zwischen Senden und Empfangen einer HTTP-Anfrage
WML	Wireless Markup Language	Auszeichnungssprache für WAP-fähige Mobilgeräte

Erklärung

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

Leipzig, 18. Juli 2001