EJB 2.0 Anwendungen

Die Reihe Programmer's Choice

Von Profis für Profis
Folgende Titel sind bereits erschienen:

Bjarne Stroustrup
Die C++-Programmiersprache
1072 Seiten, ISBN 3-8273-1660-X

Elmar Warken
Kylix — Delphi für Linux
1018 Seiten, ISBN 3-8273-1686-3

Don Box, Aaron Skonnard, John Lam
Essential XML
320 Seiten, ISBN 3-8273-1769-X

Elmar Warken
Delphi 6
1334 Seiten, ISBN 3-8273-1773-8

Bruno Schienmann
Kontinuierliches Anforderungsmanagement
392 Seiten, ISBN 3-8273-1787-8

Damian Conway
Objektorientiert Programmieren mit Perl
632 Seiten, ISBN 3-8273-1812-2

Ken Arnold, James Gosling, David Holmes
Die Programmiersprache Java
628 Seiten, ISBN 3-8273-1821-1

Kent Beck, Martin Fowler
Extreme Programming planen
152 Seiten, ISBN 3-8273-1832-7

Jens Hartwig
PostgreSQL — professionell und praxisnah
456 Seiten, ISBN 3-8273-1860-2

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster
480 Seiten, ISBN 3-8273-1862-9

Heinz-Gerd Raymans
MySQL im Einsatz
618 Seiten, ISBN 3-8273-1887-4

Dušan Petković, Markus Brüderl
Java in Datenbanksystemen
424 Seiten, ISBN 3-8273-1889-0

Joshua Bloch
Effektiv Java programmieren
250 Seiten, ISBN 3-8273-1933-1

PEARSON Education

ADDISON-WESLEY

Volker Gruhn
Manfred Schneider

EJB 2.0 Anwendungen

Entwurf leistungsfähiger Java-Komponenten

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek — CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

10 9 8 7 6 5 4 3 2 1

04 03 02

ISBN 3-8273-1977-3

Lektorat: Martin Asbach, masbach@pearson.de
Korrektorat: Sonja Fischer, München
Produktion: Monika Weiher, mweiher@pearson.de
Satz: Hilmar Schlegel, Berlin — gesetzt in Linotype Aldus/Palatino, Monotype Gill Sans
Einbandgestaltung: Christine Rechl, München
Titelbild: Carex grayi, Riedgras. © Karl Blossfeldt Archiv —
Ann und Jürgen Wilde, Zülpich/VG Bild-Kunst Bonn, 2002
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer

Printed in Germany

Inhaltsverzeichnis

1 Einleitung

Neue Softwaretechnologien werden in die industrielle Softwareentwicklung eingebracht, bevor ihre Leistungsfähigkeit demonstriert ist. Die Hersteller von Entwicklungswerkzeugen und Betriebsplattformen versprechen ein ums andere Mal Produktivitätssteigerungen, die ins Reich der Fabel gehören und mit gutem Willen gerade noch als absurd bezeichnet werden können. Und wenn eine neue Technologie dann überhaupt mal funktioniert, dann hat man ewig damit zu tun, sie robust zu machen und in existierende Welten einzubinden.

Stimmt alles. Aber es nützt nichts. Trotzdem führt kein Weg an neuen Software- und Informationstechnologien vorbei. Und diese Aussage stimmt allein deshalb, weil de facto keine PL1- und Cobol-Entwickler mehr ausgebildet werden und weil es immer schwieriger wird, Berufsanfänger dazu zu bewegen, in uraltem Code herumzuwühlen. Existierende, de facto unwartbare Softwaresysteme werden folglich — wenn schon nicht aus Einsicht in die Notwendigkeit von Innovationen — biologisch abgelöst. Spätestens dann werden auch in den letzten Assembler-Bastionen neue Technologien Einzug halten.

Wahrscheinlich wird es dann erst recht fürchterlich. Diejenigen Unternehmen, die sich einigermaßen kontinuierlich mit neuen Technologien auseinander gesetzt haben, wissen, welches Chaos mit dem Wechsel eines Technologieparadigmas verbunden ist, welche Ankündigungen der Hersteller und Berater mit hoher Wahrscheinlichkeit glatt gelogen sind und was getan werden muss, um eine neue Technologie in einem existierenden Kontext einzusetzen. Die dringendsten Probleme reichen von Robustheit von Entwicklungswerkzeugen und neuen Anwendungen über Integration in die Arbeitsvorbereitung, die Gewährleistung eines Rechenzentrumsbetriebs und über Unterstützung eines planbaren und nachvollziehbaren Konfigurationsmanagements bis hin zur Perfomanz von Dialog- und Batchbetrieb. Wer in den 80er Jahren mit großem Enthusiasmus Client/Server-Systeme entwickelt hat und/oder wer in den frühern 90er Jahren seine ersten Gehversuche mit objektorientierter Software gemacht hat, der weiß, wie unendlich elegant die Berechnungen der Total Cost of Ownership (wir nutzen ja endlich die PCs, die ansonsten nur unnütz auf den Schreibtischen rumstehen!) und der Wiederverwendungspotenziale waren (eine Klasse wie »Anschrift« kann ja in quasi allen Anwendungen wieder verwendet werden!).

Leider weiß er auch, wie gnadenlos falsch diese Berechnungen waren. Und — hoffentlich — kann er sich auch noch daran erinnern, wie schwierig es war und teilweise noch immer ist, objektorientierte Client/Server-Systeme performant zu machen. So performant, dass die Anwender sie auch benutzen wollen. So performant, dass Batchläufe in den dafür vorgesehenen Zeitfenstern abgewickelt werden können. Und so skalierbar, dass nicht immer wieder substantielle Teile der Weltchip-Produktion aufgekauft werden müssen.

Typischerweise gelingt es mit neuen Technologien so gerade eben, die benötigte Anwendungsfunktionalität auf die Beine zu stellen. Viel weniger gelingt es in der Regel, die gängigen nicht-funktionalen Anforderungen wie Performanz, Skalierbarkeit, Robust-

heit, Fehleranfälligkeit, Benutzbarkeit im Sinne von Ergonomie und Ähnliche zu befriedigen. Als eine Gesetzmäßigkeit der Technologieunreife scheint sich mit jedem neuen Technologieschub zu bewahrheiten, dass nicht-funktionale Anforderungen umso weniger befriedigt werden können, je unreifer die eingesetzte Entwicklungstechnologie ist.

Im Rahmen dieses Buches interessiert uns eine nicht-funktionale Anforderung an Softwaresysteme in besonderem Maße. Es ist die Anforderung der Performanz. Softwareperformanz bezeichnet die Leistungsfähigkeit von Softwaresystemen. Präziser ausgedrückt handelt es sich bei der Performanz eines Softwaresystems um die Fähigkeit des Systems, eine gegebene Menge wohldefinierter Aufgaben in einem definierten Zeitintervall abzuarbeiten. Von besonderer — weil für den einzelnen Anwender unmittelbar erkennbarer — Bedeutung ist in diesem Kontext das Antwortzeitverhalten. Aber auch das Laufzeitverhalten von so genannten Batches — Bearbeitungsschritten, die keine menschliche Interaktion erfordern und die in der Regel auf große Datenbestände angewendet werden — ist von enormer Brisanz. Traditionell werden für Batches so genannte Batchfenster reserviert. Hierbei handelt es sich um Zeiten, in denen kein Dialogbetrieb stattfinden kann. Die Notwendigkeit zumindest von Montag bis Freitag von 07:00 Uhr bis 20:00 auskunftsfähig zu sein, definiert die ersten Grenzen für Batchfenster. Call Center-Services, die über die normalen Geschäftszeiten hinausragen, schränken die prinzipiell möglichen Zeiten für Batchfenster weiter ein. Denkt man dann noch an Anwendungen, die nicht mehr vom Sachbearbeiter bedient werden, sondern an solche, die sich direkt an Geschäftspartner oder gar an den Endkunden wenden, gerät man leicht in die Nähe von 7×24-Stunden-Verfügbarkeitsanforderungen. Selbst wenn diese gemildert werden können, ist klar, dass die Zeiten für Batchfenster enger werden. Die Performanz von Batchläufen ist damit ein ebenso brisantes Thema wie das von Dialoganwendungen.

Performanz ist eine nicht-funktionale Anforderung an Software. Als solche wird sie während des gesamten Entwicklungsprozesses häufig vernachlässigt und nur ungenau beschrieben. Nicht-funktionale Anforderungen gehören zu den Anforderungen, die langwierigen Streitereien zwischen Anwender und Hersteller zugrunde liegen. Sie erfordern sehr häufig umfängliche Nachbesserungs- und Umbauarbeiten. Softwareperformanz lässt sich genauso wenig wie Softwarequalität im Nachhinein einbauen. Sie darf nicht auf die nächstfolgende Version verschoben werden. Dies alles scheint auf den ersten Blick intuitiv einleuchtend und klar. Und trotzdem findet man in der Praxis immer wieder windelweiche Formulierungen der folgenden Art:

Softwaresysteme sollen performant sein, sie sollen einen reibungslosen Geschäftsbetrieb ermöglichen, sie sollen mindestens so schnell wie ihre Vorgängersysteme sein, sie sollen dem Stand der Technik entsprechen und eine dazu passende Bearbeitungsgeschwindigkeit ermöglichen.

So weit sind sich Hersteller und Anwender meist einig. Was bedeutet das aber im Einzelnen? Was ist unter einer passenden Bearbeitungsgeschwindigkeit zu verstehen? Geht es darum, dass der Sachbearbeiter mithilfe der Software in der Lage ist, eine bestimmte Menge von Geschäftsvorfällen pro Zeiteinheit zu erledigen oder geht es darum, dass er nie länger als eine Sekunde auf einen Dialogwechsel warten muss? Muss gefordert werden, dass auch komplexe Batchläufe von Freitagabend bis Montagmorgen durchge-

führt werden können oder entspricht es dem Stand der Technik, dass das zu bearbeitende Datenvolumen in Portionen zerlegt wird, die dann ihrerseits auf alle zur Verfügung stehenden Batchfenster verteilt werden? Ist es überhaupt noch zeitgemäß, Batchfenster auszuzeichnen, oder bedeuten die Anforderungen des e-Business nicht eine 7×24-Stunden-Verfügbarkeit? All diese Fragen haben etwas mit Softwareperformanz zu tun.

Während Lösungen für Performanzprobleme für etablierte Softwaretechnologien in der Regel bekannt sind (diejenigen Technologien, für die es solche Lösungen nicht gibt, verlieren schnell und grundlegend an Bedeutung), stellen Performanzanforderungen für neue Technologien oft eine erhebliche Hürde dar. Vereinfachend kann man diese Erfahrung in der folgenden Regel über den Zusammenhang zwischen Technologiereife und Performanz zusammenfassen (hierbei ist der Begriff der Anwendung so zu verstehen, dass es sich um eine ernsthafte, in aller Regel unternehmensweite Anwendung handelt, die tatsächlich gebraucht wird, der Speiseplan im Intranet soll nicht als eine solche Anwendung gelten):

Regel zur Technologiereife: Je unreifer eine Entwicklungstechnologie, desto mehr Aufwand muss getrieben werden, um mit ihr eine performante Anwendung zu erstellen.

Software Performance Engineering [73] verfolgt das Ziel, die Performanz von Software während des gesamten Entwicklungsprozesses zu verfolgen. Dieser prozesszentrierte Ansatz wird in fast allen wichtigen Ansätzen des Software Performance Engineering verfolgt [10]. In Anlehnung an das Capability Maturity Model (CMM) ist das Performance Engineering Maturity Model (PEMM) entstanden [56]. Hier werden die einzelnen Aktivitäten des Software Performance Engineering so genannten Subprozessen zugeordnet und diese Subprozesse hinsichtlich ihres Reifegrads klassifiziert.

Während der Anforderungsanalyse bedeutet das, dass die Performanzanforderungen quantifiziert und messbar spezifiziert werden. Zu den typischen performanz-bezogenen Anforderungen gehören die Antworten auf die folgenden Fragen:

- Wie viele Anwender sollen parallel arbeiten können?
- Welche Reaktionszeiten müssen gewährleistet werden (in wie viel Prozent der Fälle)?
- Welche maximalen Reaktionszeiten sind für welche Arten von Transaktionen akzeptabel?
- Welches Datenvolumen muss in welchen Batchläufen innerhalb welcher Zeit bearbeitet werden können?

Viele Ansätze des Software Performance Engineering fordern, dass nicht nur die Anforderungen in eindeutiger Weise angegeben werden, sondern dass auch die Messverfahren, mithilfe derer später beurteilt werden soll, ob die Anforderungen eingehalten worden sind, angegeben werden [62].

Auch in anderen Aktivitäten der Softwareentwicklung lassen sich performanz-relevante Eigenschaften überprüfen. In der fachlich geprägten Architektur und im Ergebnis der objektorientierten Analyse lassen sich in aller Regel die potenziellen Performanzschwach-

punkte identifizieren. Dies gelingt meist jedoch nur, wenn die Prüfung vor dem Hintergrund einer soliden Kenntnis der Anwendungsdomäne erfolgt. Die erkannten Schwachstellen lassen sich auf dieser Ebene meist nicht oder nur unvollständig ausräumen, da das fachliche Modell (als Abstraktion der Realität) von Entwurfsentscheidungen abstrahieren soll. Dennoch können die Hinweise auf der Grundlage der fachlichen Modelle benutzt werden, um während des Entwurfs an bestimmten Stellen von vornherein eine gewisse Performanzsorgfalt walten zu lassen.

Williams und Smith verweisen in [72] darauf, dass »Our experience is that most performance failures are due to a lack of consideration of performance issues early in the development process, in the architectural design phase.« Clements [11] geht darüber hinaus und legt nahe, auf der Grundlage von Architekturbeschreibungen erste Performanzvorhersagen abzuleiten, um so das potenzielle Ausmaß des Schreckens (also einer völlig unzureichenden Performanz) möglichst früh klar zu machen. Für manche, nachgerade Performanz verhindernde Architekturentscheidungen wie den Einsatz von selbst geschriebenen Frameworks, Proxies und Proxy-Generatoren (oder ähnlichen Ergüssen, die sich darauf zurückführen lassen, dass ein Anwender (sic!) sich nicht damit abfinden kann, dass er Software nur anwenden, aber nicht herstellen soll), braucht man allerdings nicht einmal Berechnungs- und Vorhersagemodelle. Hier reicht — wie für viele andere Performanzkiller auch — Erfahrungen mit den Auswirkungen von Architekturentscheidungen auf die Implementierung von Software.

Die Realisierung der Software selbst ist sicherlich eine weitere Hauptquelle für mangelhafte Performanz. Mangelnde Kenntnis effizienter Algorithmen, sorglose Wahl konkreter Algorithmen, Abstraktion von Laufzeit- und Speicherplatzverhalten sind in Zeiten augenscheinlich unbegrenzter Hauptspeicher und immer schnellerer Prozessoren auf den ersten Blick lässliche Defizite. Für ernsthafte Anwendungen (viele Anwender, große Datenvolumen) rächt sich jedoch jedes einzelne dieser Defizite bitter. Die Erfahrung lässt unglücklicherweise die folgende Regel zum Zusammenhang zwischen Beherrschung neuer Technologien und der Kenntnis von Performanzproblemen vermuten:

Regel zur Entwicklerunreife: Ein Entwickler muss erst am eigenen Leib (soll heißen in einem eigenen Projekt) gespürt haben, was es bedeutet, grundlegende Entwurfs- und Realisierungsentscheidungen im Hinblick auf ihre Performanzrelevanz zu prüfen. Hat er das getan, weiß er, was Performance Engineering bedeutet. Leider dauert das aber so lange, dass er nicht mehr die wirklich aktuellen Entwicklungstechnologien kennt.

Auch andere Aktivitäten der Softwareentwicklung (wie Testentwurf, Test, Überdeckungsanalysen, Inbetriebnahme, Monitoring) werden von Performanzaspekten durchdrungen. Generell lässt sich fordern, dass bei allen Entwicklungsaktivitäten geprüft werden muss, welche Auswirkungen auf die Performanz durch das jeweilige Handeln verursacht werden. Das Ergebnis solcher Prüfungen lässt sich durch den Einsatz nachvollziehbarer Vorhersagemodelle belastbar machen. Typischerweise lassen sich die quantifizierten Aussagen solcher Vorhersagen schlechter verdrängen und wegreden als Performanzprobleme, die lediglich qualitativ bezeichnet werden können.

Software Performance Engineering ist das Gebiet an der Nahtstelle zwischen Software Engineering und Performance Management [74]. Obwohl Methoden und Verfahren des

Software Engineering zunehmend auch die industrielle Praxis der Softwareentwicklung prägen und obwohl in fast allen großen Projekten die Sorge um die Performanz der neuen Anwendung eine erhebliche ist, kann von systematischem Software Performance Engineering in der Praxis selten die Rede sein.

Wenn überhaupt, dann findet Software Performance Engineering in Gebieten statt, in denen die noch drängenderen Probleme als Performanz gelöst sind. Das heißt, es geht um bekannte Anwendungsfelder und die einzusetzenden Technologien werden gut beherrscht, sie sind stabil und robust. In solchen Situationen haben Softwareentwickler oftmals ein verlässliches Gefühl für das, was an Performanz möglich und den Anwendern zumutbar ist. Kommt es zu Problemen, werden schon frühzeitig Performanzexperimente angestellt, Algorithmen werden verbessert und zuweilen werden sogar frühe Artefakte des Entwicklungsprozesses (fachliche Modelle, Entwürfe) im Hinblick auf potenzielle Performanzprobleme untersucht.

Gerade beim Einsatz neuer Entwicklungstechnologien findet sich demgegenüber ein weit verbreitetes Paradox:

Die Technologien sind noch so unreif, dass sie von vornherein mit höherer Wahrscheinlichkeit zu inperformanten Anwendungen führen. Gerade weil sie unreif sind, wird Performanzanforderungen zu wenig Beachtung gewidmet.

Dieses Paradox führt zum einen dazu, dass eine Vielzahl von Entwicklungsprojekten, in denen neue Softwaretechnologien eingesetzt werden, scheitern. Dies alleine ist oftmals ein Desaster ohnegleichen, das in keiner anderen als der Softwareindustrie in der gegebenen Häufigkeit denkbar ist! Zum anderen führt das genannte Paradox aber auch dazu, dass neue Softwaretechnologien manchmal nicht den Weg in die Praxis finden, einfach deshalb, weil die ersten mit großem Interesse bestaunten Projekte scheiterten und die entsprechenden Technologien allzu schnell als Irrwege abgetan werden.

Diesem Paradox lässt sich nur entgegnen, wenn die Gründe im Einzelnen verstanden werden und wenn entsprechende korrektive Maßnahmen eingeleitet werden. Zu den Gründen zählen:

1. Mangelnde Verankerung des Software Performance Engineering im Softwareprozess: Softwareprozesse sind in gewissen Teilen technologieunabhängig. In fast allen Softwarehäusern und bei fast allen großen Entwicklern von Software gibt es mindestens Fragmente von Softwareprozessmodellen (also Beschreibungen, wie die einzelnen Aktivitäten der Entwicklung gestaltet werden sollen, welche Reihenfolgen und Abhängigkeiten zu beachten sind und welche Rolleninhaber für welche Aktivitäten und Ergebnisse verantwortlich sind). Diese Modelle überleben meistens auch den Wechsel zu einer anderen Technologie. Änderungen und Anpassungen sind notwendig, werfen die grundlegende Struktur der Modelle aber nicht um. Wenn in einem solchen Modell die Aktivitäten des Software Performance Engineering nicht verankert sind, dann wird der Wechsel zu einer neuen Technologie nicht dazu führen, dass sie eingeführt werden. Stattdessen sind die Entwickler und Prozessverantwortlichen mit den dringenden Anpassungen an die neue Technologie beschäftigt. Strukturelle Erweiterungen — und die Ergänzung um Aktivitäten des Software Performance Engineering

wäre sicherlich eine — bleiben unberücksichtigt. Der immense Schaden basiert darauf, dass für die etablierte Technologie Performance Engineering nicht mehr nötig war, erfahrene Entwickler wussten, was in welcher Weise getan werden musste, um performante Ergebnisse zu erzielen. Das Fehlen im Modell wurde durch Erfahrung kompensiert. In Bezug auf die neue Technologie fehlen dann plötzlich Erfahrungen und das anleitende Prozessmodell.

2. Geringe Erfahrung mit Performanzeigenschaften einzelner Ergebnisse des Softwareprozesses: Selbst wenn Entwickler ahnen (oder gar wissen), dass sie sich frühzeitig um die Performanz der zu entwickelnden Software kümmern sollten, fehlt ihnen oft die konkrete Erfahrung mit den Ergebnissen der Entwicklungsaktivitäten. Ein erfahrener Cobol-Entwickler kann für das Klassendiagramm, das als Ergebnis des objektorientierten Entwurfs entsteht, halt nicht notwendigerweise entscheiden, ob es die Grundlage einer performanten Realisierung sein kann. Seine Performanzerfahrungen gehören zu einer Technologie, der Transfer auf eine andere Technologie ist mit Unsicherheiten und Verunsicherungen behaftet.

3. Unklare Wechselwirkungen zwischen den Bestandteilen integrierter, oftmals verteilter Hardware-/Softwaresysteme: Je verteilter Softwaresysteme sind und je verteilter die Hardwaresysteme sind, auf denen sie laufen, je abstrakter die Services werden, die von Middleware-Systemen, Telekommunikationseinrichtungen und Netzwerken erbracht werden, umso schwieriger ist es, Performanz verlässlich vorherzusagen. Die Menge der potenziell beeinflussenden Parameter nimmt einfach zu stark zu. Als Konsequenz werden solche verteilten Anwendungen entwickelt (mit großem, in der Regel abnehmendem Vertrauen und mit kleinem, in der Regel zunehmendem Unbehagen). Die Verteiltheit und das komplexe Zusammenspiel verschiedener Teilsysteme macht dann auch noch die Diagnose schwierig. Ist die Anwendung als Ganzes nicht so performant wie nötig, ist es extrem kompliziert die Stellschrauben für die nötigen Verbesserungen zu identifizieren. Experten, die wissen, dass es am Netzwerk, am zu geringen Hauptspeicher und an fehlenden Datenbank-Indizes liegt, sind schnell gefunden (meistens stimmt alles auch ein bisschen). Aber die wesentlichen Ursachen liegen dann meistens doch tiefer: Anwendungsstruktur, Algorithmen, unkoordiniertes Lesen und Schreiben von Daten. Die Verbesserungen in diesen Bereichen sind oft unvermeidbar und immer teuer. Das lässt sich zuweilen ertragen, es sei denn der Termin für den Produktivbetrieb ist nahe und unverschiebbar. Denn eines ist sicher: Bei Performanzverbesserungen, die die interne Struktur der Anwendungen betreffen, schlägt der mythische Personenmonat voll zu oder — mit anderen Worten — es gibt nur ganz wenige, die diese Aufgabe wahrnehmen können, es nützt nichts, ihnen Assistenten zur Seite zu stellen. Oder mit noch anderen Worten: Anwendungsstrukturelle Performanzverbesserungsmaßnahmen brauchen ihre Zeit und können durch zusätzlichen Personaleinsatz nicht verkürzt werden. Nein, auch dann nicht, wenn die Entwicklung ansonsten abgebrochen werden muss.

4. Anforderungsdruck und schwach ausgeprägtes Anforderungsmanagement: Entwickler überschätzen sich. Immer wieder. Gerade am Anfang kommen ihnen die Anforderungen des Anwenders ja doch eher lapidar vor. Es geht halt nur um die Verwaltung von Versicherungsverträgen oder Wohnungen oder Telefonkunden oder von sonst

irgend was einfachem. Und da das ja alles so einfach ist, lohnt es sich auch nicht im Detail über die ein oder andere Anforderung zu diskutieren. Dumm nur, wenn die Entwickler, die ein Bestandsführungssystem bauen, erst noch lernen müssen, was Staffelzu- und -abschläge sind, was versicherte Risiken und Deckungsausschlussklauseln sind, wenn die Entwickler, die ein Informationssystem für die Wohnungsverwaltung bauen, erst noch lernen müssen, was Betriebskosten und Staffelmieten sind und wenn diejenigen, die Telefonkunden verwalten sollen, lernen müssen, was Call Detail Records sind und welche Tarifvielfalt bewältigt werden muss. Jetzt beginnt der Prozess der Diskussion: muss das denn wirklich alles in der ersten Version zur Verfügung stehen, reicht denn nicht auch Version 127? Und wenn der Anwender nun zu eindeutig gewinnt, hat er auch schon verloren. Dann sind die Entwickler, denen der Anwender jetzt ja gerne zeigt, dass sie beim nächsten Mal doch etwas mehr Respekt vor detailliertem Anwendungswissen haben sollten, bis zum letzten Tag vor der Inbetriebnahme der Anwendung damit beschäftigt, funktionale Anforderungen zu erfüllen. Manchmal haben alle Glück und das klappt gerade so. Und dann könnte eine katastrophal inperformante Anwendung in Betrieb gehen. Gewinnen können Anwender und Entwickler also nur, wenn sie Anforderungen frühzeitig klären, klassifizieren und priorisieren und zwar so, dass der Code Freeze (also das Ende der Entwicklung weiterer Funktionalitäten) nach maximal 80 % der Entwicklungszeit erfolgt.

5. Herausforderungen beim Einsatz neuer Technologien lenken von Perfomanzproblemen ab: Der Einsatz neuer Softwaretechnologien ist davon begleitet, dass viele Integrationsprobleme gelöst werden müssen. Es muss geklärt werden, wie existierende und neue Anwendungen integriert werden können, wie existierende Rechner eingebunden werden können und wie aktuelle Datenbestände verfügbar gemacht werden können. Bevor diese Integrationsprobleme nicht gelöst sind, kann über die Performanz einer neuen Anwendung kaum eine Aussage gemacht werden. Allein diese zeitliche Abhängigkeit bedeutet, dass Performanzprobleme in kurzer Zeit gelöst werden müssen und dass eine kontinuierliche, den Softwareprozess begleitende Untersuchung der erzielbaren Performanz nur selten stattfindet.

6. Technologiemängel und Erweiterungsbedarf: Neue Technologien werden durch Werkzeuge ermöglicht, die selbst unter sehr stark technologie-orientierten Aspekten entwickelt wurden. Handhabbarkeit, Administrierbarkeit und Unterstützung für die Erfüllung nicht-funktionaler Anforderungen spielen für die frühen Versionen dieser Werkzeuge oft nur eine untergeordnete Rolle. Erweiterungen und Ergänzungen von technologischen Konzepten und Werkzeugen sind meist nötig, um überhaupt in die Lage zu kommen, produktiv einsetzbare Anwendungen entwickeln zu können. Mit solchen Erweiterungen und Ergänzungen sind Versionswechsel und manchmal sogar Migration nötig, sodass es in zeitlich knappen Projekten zu zusätzlichen Störimpulsen kommen kann.

Damit ist klar, dass der zu frühe Einsatz neuer Technologien fast schon zwangsläufig auf ernsthafte Performanzprobleme hinausläuft. Auf der anderen Seite ist der zu späte Einsatz von Technologien oft damit verbunden, dass zu lange in alte Technologien investiert wird, dass Wettbewerbsnachteile in Kauf genommen werden müssen und dass

technologische Führerschaft, die ja in immer mehr Branchen auch mit der inhaltlichen und fachlichen Führerschaft verknüpft ist, aufgegeben werden muss, respektive gar nicht erst erreicht werden kann.

Was ist also zu früh und was ist zu spät? Eine allgemeine Antwort auf diese Frage trauen wir uns nicht zu. Aber für die komponentenbasierte Entwicklung ist der richtige Zeitpunkt gekommen!

Die komponentenbasierte Softwareentwicklung setzt zwei jahrzehntelange Trends des Software Engineering fort: den Trend zur Abstraktion und den Trend zur Verteilung. Wiederzuverwendende Bestandteile von Softwaresystemen sollen endlich auf das Anwendungsniveau gehoben werden. Ein Entwickler eines Bestandsführungssystems soll endlich in die Lage versetzt werden, so etwas wie einen Bestandteil »Partner« wiederzufinden (nicht etwa eine parametrisierbare Klasse zur Verwaltung doppeltverketteter Listen). Ein Softwaresystem, das aus solchen vorgefertigten Bestandteilen zusammengesetzt ist, soll flexibel auf verschiedenen Plattformen und Rechnern zum Einsatz kommen können, ohne dass programmiertechnische Anpassungen erforderlich werden. Der Charme einer solchen Idee ist offensichtlich: Komponenten können von verschiedenen Herstellern bezogen werden, manche können selbst entwickelt werden, alle können zusammengestöpselt werden und das entstehende Softwaresystem kann plattformunabhängig eingesetzt werden. Für den Rest dieses Buches beschränken wir uns auf eine Teilmenge der komponentenbasierten Softwarewelt, nämlich auf die Welt der Enterprise JavaBeans. Für eine ausführliche Diskussion zu Stärken und Schwächen dieser Teilwelt und der Teilwelt der COM-Komponenten sei auf [30] verwiesen.

Inwieweit die Versprechen der komponentenbasierten Softwareentwicklung umgesetzt werden können oder dann doch an den Ecken und Kanten der realen Welt scheitern, werden wir uns in den folgenden Kapiteln anschauen. An dieser Stelle interessiert uns zunächst nur eine Frage: Ist es nicht vielleicht noch zu früh, für den Einsatz der komponentenbasierten Softwareentwicklung?

Diese Frage beantworten wir mit einem entschiedenen Nein und führen dafür folgende Gründe an:

- Die allerschönsten Versprechen sind gekippt, der Rest bleibt attraktiv genug und kann realisiert werden.
- Technologische Irrwege im Kontext der komponentenbasierten Entwicklung sind inzwischen klar und deutlich identifiziert worden (Beispiel: Frameworks, Generatoren, formale Schnittstellenbeschreibungssprachen).
- Es gibt mittlerweile viele, produktive Anwendungen. Einige dieser Anwendungen sind groß genug, sodass sie auch als Muster für den Einsatz relevanter und ernsthafter Anwendungen dienen können.
- Die Kerntechnologien und die wesentlichen unterstützenden Werkzeuge (Komponentenmodelle, Applikationsserver, Entwicklungsumgebungen) gibt es seit ein paar Jahren. Die Änderungen und Erweiterungen, die — folgend auf die initialen, eher

technologisch geprägte Versionen — vorgenommen wurden, um auch typische nichtfunktionale Anforderungen erfüllen zu können, haben sich bewährt und sind seit längerem verfügbar. Mit der Spezifikation »Enterprise JavaBeans 2.0« gibt es ein Komponentenmodell (und dazu wiederum gibt es passende Werkzeuge), das explizit auf die Verbesserung der Performanz entsprechender Anwendungen abzielt.

Bleibt die Frage, ob es vielleicht schon zu spät ist für den Einstieg in die komponentenbasierte Softwareentwicklung. Auch diese Frage scheint mit einem Nein beantwortet werden zu können. Zwar gibt es erste große produktive Anwendungen und zwar haben viele der großen Anwender von Informationssystemen ihre ersten Gehversuche auf dem Weg der Migration in komponentenbasierte Softwarelandschaften hinter sich. Dennoch gibt es bisher kaum durchgängig komponentenbasierte Softwarelandschaften. Und erst mit denen erschließen sich die vollen Vorteile der Komponentenwelt. Andere, auf das Komponentenparadigma folgende Technologieinnovationen sind nicht erkennbar. Keine andere Softwaretechnologie der letzten 15 Jahre hat die dauerhaften Trends des Software Engineering so konsequent fortgesetzt wie die komponentenbasierte Entwicklung. Keine andere Technologie hat so radikal an der Hauptursache der Softwarekrise angesetzt, nämlich dem Anwendungstau (also der Unfähigkeit, dringend benötigte Anwendungen unter effizienter Ausnutzung des vorhandenen Anwendungswissens produktiv einsetzbar zu machen). Und selten ist eine Technologie so schnell zum Mainstream der Innovation in der industriellen Softwareentwicklung geworden. Nimmt man dann noch an, dass die Vorteile der komponentenbasierten Entwicklung erst mit dem Einsetzen von Netzwerkeffekten vollständig nutzbar werden (also erst dann, wenn hinreichend viele Hersteller und Anwender auf Komponenten setzen), ist es noch nicht zu spät. Besonders gute Gründe weiterhin abzuwarten gibt es allerdings auch nicht.

Das Ziel dieses Buches ist es, die Erfahrungen, die wir bei der Entwicklung performanter EJB-basierter Anwendungen gemacht haben, zu dokumentieren. Dies geschieht in der Hoffnung, dass der ein oder andere Leser nicht alle Erfahrungen selbst machen muss. Die meisten Erfahrungen — und auch die schmerzvollsten — haben wir mit der Entwicklung von Anwendungen gemacht, die auf frühen Versionen des EJB-Standards basieren. Diese Anwendungen performant — vielleicht eher einsetzbar — zu machen, hat viel Mühe gekostet. Die beteiligten Entwickler haben während verschiedener Inbetriebnahmen Blut und Wasser geschwitzt. EJB 2.0 ist die Version des EJB-Standards, die einige der Merkmale, die man für die Entwicklung performanter Anwendungen schon seit längerem gebraucht hätte, endlich bietet.

Mit EJB 2.0 lassen sich performante Anwendungen planbar entwickeln, daran haben wir nur noch wenig Zweifel. Natürlich lassen sich nicht alle Arten von Anwendungen mit Enterprise JavaBeans 2.0 in der nötigen Performanz entwickeln. Realzeitkritische Anwendungen sind wohl nicht das am allerbesten geeignete Anwendungsfeld. Und hochsicherheitskritische Anwendungen sollten wohl auch nicht von Korrektheit und Robustheit eines EJB-Applikationsservers abhängen. Für das, was die große Mehrheit der Anwendungsentwickler in ihrer täglichen Praxis beschäftigt, nämlich die Entwicklung von Informationssystemen, bietet EJB 2.0 nicht nur eine gute Basis, sondern zusammen mit den Services der marktgängigen Applikationsserver sogar eine hochproduktive Plattform.

2 Überblick

Das vorliegende Buch bildet verschiedene Design-Varianten eines Geschäftsprozesses und untersucht sie auf ihre Performanz hin. Aus diesen Varianten werden allgemeine Design-Regeln für performante EJB-Anwendungen ermittelt. Als Beispiel dient ein Geschäftsprozess aus der Versicherungswirtschaft. Das folgende Kapitel gibt einen Überblick über die Struktur des Buches.

Kapitel 3 gibt eine kurze Einführung in den Bereich der *komponentenbasierten Softwareentwicklung*. Enterprise JavaBeans werden als ein konkretes Komponentenmodell vorgestellt.

Kapitel 4 stellt die *Neuerungen des EJB 2.0-Standards* vor. Zu jeder der Neuerungen werden ausführliche Quelltextbeispiele angegeben und Zeile für Zeile erläutert.

Kapitel 5 entfaltet den Performanz bezogenen Hintergrund des Buches. Abschnitt 5.1 legt den benutzten Performanzbegriff dar. Aus den vorgestellten Maßen ergibt sich die processing time (Dauer der Abarbeitung) als geeignetes Maß für die Performanz der Implementierungsvarianten. Abschnitt 5.2 bestimmt die Grenze zwischen Performance Tuning und Software Performance Engineering. Abschnitt 5.3 skizziert die Bedeutung von Massendatenverarbeitungsprozessen in der Versicherungswirtschaft. Es wird geschildert, warum gerade Massendatenverarbeitungsprozesse für Performanzmessungen relevant sind. Abschnitt 5.4 beschreibt die technischen Grundlagen und gibt einen Ausblick auf die möglichen Verbesserungsansätze zur Performanzsteigerung.

Kapitel 6 vertieft die fachliche Problemstellung, indem es das *Anwendungsbeispiel* einer objektorientierten Analyse unterzieht:

»Ein Bestand von KFZ-Versicherungsverträgen wird bearbeitet, sodass den Verträgen, die keine Leistungen der Versicherung in Anspruch genommen haben, ein günstigerer Schadensfreiheitsrabatt (`SfKlasse`) auf die Prämie gewährt wird.«

Die objektorientierte Zerlegung des Geschäftsobjekts »KFZ-Versichungsvertrag« in Teilobjekte wird in Abschnitt 6.1 durchgeführt. Abschnitt 6.2 erläutert den Ablauf des Geschäftsprozesses mithilfe von Aktivitätsdiagrammen. Die Tauglichkeit des Geschäftsobjekts und des Geschäftsprozesses für die Ermittlung der Design-Regeln wird diskutiert. Das Kapitel schließt mit einer Abgrenzung des Beispiels zu einem produktiven System.

Kapitel 7 erläutert die *Durchführung* der Performanzmessung der einzelnen Varianten. Abschnitt 7.3 stellt ein Konzept zur Umsetzung von Massendatenverarbeitungsprozessen vor. Die Klassen zur Messung der Performanz werden dargelegt und ihr Zusammenspiel mit den Klassen der Implementierungsvarianten beschrieben. Der Umfang und die Zusammensetzung der Testdaten, die durch das Anwendungsbeispiel bearbeitet werden, werden motiviert. Diese Testdaten bilden die sog. workload (siehe Abschnitt 5.1). Ein einheitliches Ergebnisprotokoll gibt die Beschreibung der Implementierungsvarianten und die Interpretation der Messergebnisse vor. Abschnitt 7.4 skizziert seinen Aufbau.

Kapitel 8 beginnt mit der Beschreibung des objektorientierten Entwurfs der Basisvariante. In den folgenden Abschnitten werden nach dem in Unterkapitel 7.4 vorgestellten Ergebnisprotokoll die wesentlichen Entwurfentscheidungen der *Varianten* wiedergegeben. Es wird genannt, worin sie sich zu der ausführlich beschriebenen Basisvariante unterscheiden und wieso eine Verbesserung zu erwarten ist. Das *Ergebnis des Messlaufs*, die Dauer der Abarbeitung des Geschäftsprozesses (*processing time*), wird dokumentiert und interpretiert. Das Kapitel schließt mit einer Variante, die alle zuvor beschriebenen Verbesserungen in sich vereinigt.

Kapitel 9 fasst die ermittelten Ergebnisse als Design-Regeln zusammen. Das Buch schließt mit einem Ausblick in **Kapitel 10**.

Anhang A enthält die Auflistung der eingesetzten Hard- und Software. Dem Buch liegt eine CD mit dem Quelltext der Implementierungsvarianten bei. **Anhang B** beschreibt, wie die darauf befindlichen Prototypen übersetzt und gestartet werden können.

3 Komponentenbasierte Softwareentwicklung mit Enterprise JavaBeans

In diesem Kapitel wird ein Überblick über die komponentenbasierte Softwareentwicklung gegeben. Die Grundidee der komponentenbasierten Entwicklung ist zwar weitgehend technologieunabhängig, aber dennoch kommt es bei konkreten Ausprägungen in Vorgehen und Basistechnologien zu konkreten Festlegungen, die substanziellen Einfluss auf konkrete komponentenbasierte Softwareprozesse haben. In diesem Kapitel werden zunächst die grundlegenden Ideen der komponentenbasierten Entwicklung erörtert (Abschnitt 3.1). Danach wird der Begriff der Softwarekomponente in Abschnitt 3.2 diskutiert. Abschnitt 3.3 gibt einen ganz kurzen Überblick über die Bedeutung des Komponentenmodells Enterprise JavaBeans. Weitere Details — insbesondere der Version 2.0 der EJB-Spezifikation — werden in Kapitel 4 erörtert. In Abschnitt 3.4 werden wesentliche Services von EJB-Applikationsservern vorgestellt.

3.1 Komponentenbasierte Softwareentwicklung

Der komponentenbasierten Softwareentwicklung liegt die Idee zugrunde, dass Anwendungen zukünftig aus Bausteinen (den so genannten Komponenten) zusammengesetzt werden können, die auf dem Abstraktionsniveau der Anwendungswelt angesiedelt sind. Mit anderen Worten: ein Anwendungsentwickler in der Versicherungswirtschaft bekommt solche Komponenten zur Verfügung gestellt, die im Kontext der Versicherungswelt eine fachliche Bedeutung haben, ein Anwendungsentwickler in der Logistik arbeitet mit ganz anderen Komponenten. Beide nutzen die Komponenten ihres Kontexts, um daraus Anwendungen aus ihrer Welt zusammenzusetzen.

Vereinfachend wird die komponentenbasierte Entwicklung häufig mit der Legostein-Metapher erörtert. Die Bausteine sollen so präpariert sein, dass sie einfach zusammengesetzt werden können. Sie sollen wiederverwendbar sein und es wird eine ganze Reihe von Bausteinen geben, die nur für bestimmte Anwendungsdomänen geeignet sind. Das Zusammensetzen der Komponenten erfolgt dadurch, dass Komponenten Services anbieten, die von anderen Komponenten benutzt werden. Ein typischer Grundsatz der komponentenbasierten Entwicklung lautet beispielsweise:

Anwendungen werden aus Komponenten zusammengebaut. Komponenten bieten als Service eine Menge von Funktionen an, die in einem bestimmten Gebiet logisch zusammengehören. Andere Komponenten können diesen Service einer Komponente nutzen. Als Vertragsgrundlage zwischen Komponenten dienen dabei Schnittstellen, die Teilmengen der angebotenen Methoden zusammenfassen.

Der Grundsatz besagt, dass es vorgefertigte Bausteine gibt, in denen auf einer höheren Abstraktionsstufe all die Programmteile zusammengefügt sind, die für die Erbringung eines Service in einem bestimmten Bereich zusammenarbeiten. Die Komponente kapselt die Implementierung des Service und stellt diesen Service extern durch eine Menge von Schnittstellen zur Verfügung, die von anderen Komponenten angesprochen werden können. Komponenten können Dienstleistungen aus der Anwendungswelt bereitstellen (Beispiel wäre hier eine Komponente »Partnerverwaltung«), aber auch Infrastrukturkomponenten (z.B. zentral zur Verfügung gestellte, allgemein genutzte Dienste wie Workflowmanagement und Dokumentenmanagement) sind möglich.

Die etwas zu starke Vereinfachung der genannten Metapher besteht darin,

- dass Komponenten nur ganz selten »zusammengestöpselt« werden können, ohne dass Anpassungen notwendig sind,
- dass von der internen Struktur einer Komponente in der Regel nicht so einfach abstrahiert werden kann wie bei einem Legostein,
- dass die Funktionalität der bereitgestellten Services beschrieben werden muss — und zwar möglichst eindeutig und unmissverständlich, bevor sie von anderen Komponenten sinnvoll benutzt werden können.

Die Metapher macht aber auch ein Kernproblem der komponentenbasierten Entwicklung deutlich: Genau wie bei Legosteinen ist es eine ganz wesentliche Entwurfsentscheidung, welche Komponenten bereitgestellt werden. Sind sie zu nah an einer konkreten Anwendung, können sie für diese Anwendung zwar wunderbar eingesetzt werden, die Chance auf ihre erneute Verwendung im Rahmen anderer Anwendungen ist aber eher gering. Sind sie zu allgemein, können sie prinzipiell sehr oft verwendet werden, allerdings muss in jedem Einzelfall mit Anpassungen gerechnet werden.

Die Nutzung von Services anderer Komponenten über Schnittstellen ist in Abbildung 3.1 erläutert. Die obere Komponente bietet einen Service an, der über drei Schnittstellen in Anspruch genommen werden kann (dargestellt am oberen Rand des grafischen Symbols). Jede dieser Schnittstellen realisiert einen Teil des Service. Die obere Komponente nimmt zwei Service-Funktionen anderer Komponenten in Anspruch. Über die entsprechenden Schnittstellen dieser Komponenten kann die obere Komponente bereitgestellte Service-Funktionen aufrufen (dargestellt am unteren Rand des grafischen Symbols für die obere Komponente). Eine Aufrufbeziehung wird mithilfe eines Pfeiles zwischen aufrufender und aufgerufener Komponente veranschaulicht.

Die letzten Jahrzehnte in der Softwareentwicklung waren von zunehmender Verteilung von Softwaresystemen geprägt. Aus monolithischen und intern nicht strukturierten Systemen wurden Two-tier- und Multi-tier-Systeme. Die Aufteilung in Schichten wurde weiter aufgeweicht, als Ergebnis entstanden Systeme, die aus einer Reihe von Bausteinen bestehen, die flexibel verteilt werden können und die über vordefinierte Arten von Beziehungen zusammengesetzt werden können. Abbildung 3.2 veranschaulicht diese Tendenz. Ausgehend von monolithisch strukturierter Software aus den Anfängen der Softwareentwicklung wurden die Systeme immer feingranularer strukturiert. Die Aufteilung

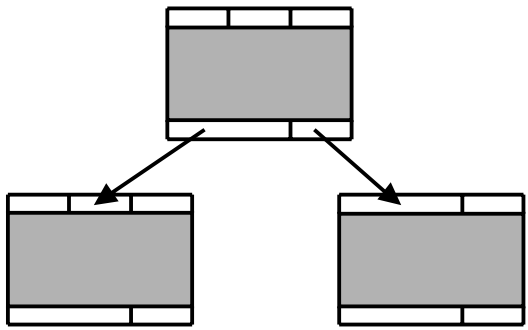


Abbildung 3.1: Nutzung von Services anderer Komponenten über Schnittstellen

in die drei klassischen Schichten Oberfläche, Anwendung/Applikation und Datenbank (DB) und der damit verbundene Übergang zu Client/Server-Systemen stellt einen ersten wichtigen Meilenstein dar. Die Aufteilung in weitere Schichten (oberflächennahe Anwendung, DB-Zugriffsschicht und ähnliche) und die horizontale Aufteilung (also die Aufteilung von Schichten in bestimmte logisch zusammengehörende Teile) setzen den Trend fort. Bei genügend kleinteiliger Strukturierung wird es dann fast unvermeidlich, die fixe Zuordnung von Bestandteilen zu Rechnern aufzugeben und damit bei Geflechten von verteilten und flexibel verteilbaren Komponenten anzukommen.

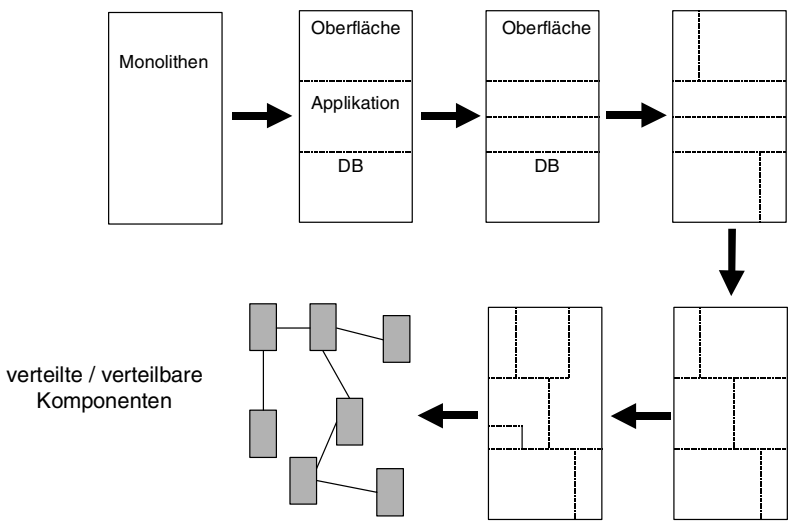


Abbildung 3.2: Verteilung und Strukturierung von Softwaresystemen

Vergleicht man diesen Aufbau mit herkömmlichen Architekturen, wie z.B. Client/Server-Architekturen, so kann man durch die flexible Kombination von Komponenten den Softwareaufbau nicht mehr einer der üblichen Softwarearchitekturen zuordnen. Es

ist somit eine neue Sicht und Beschreibung erforderlich. Abbildung 3.3 zeigt, dass in typischen komponentenbasierten Anwendungen die Anwendungslogik konzentriert in einer Schicht (zumindest aber in einer Reihe von Komponenten) angesiedelt wird, die von der Präsentationsschicht und von der Persistenzschicht getrennt ist. Diese Komponenten werden typischerweise von einem Applikationsserver unterstützt, der mittels der genannten Services eine Konzentration der Anwendungsentwickler auf die Anwendungslogik unterstützt. Auf diese Weise wird es überflüssig, die Anwendungslogik auf Client- und Persistenzschicht zu verteilen und so zu »fetten« Clients oder nicht mehr erkennbaren Persistenzmechanismen beizutragen.

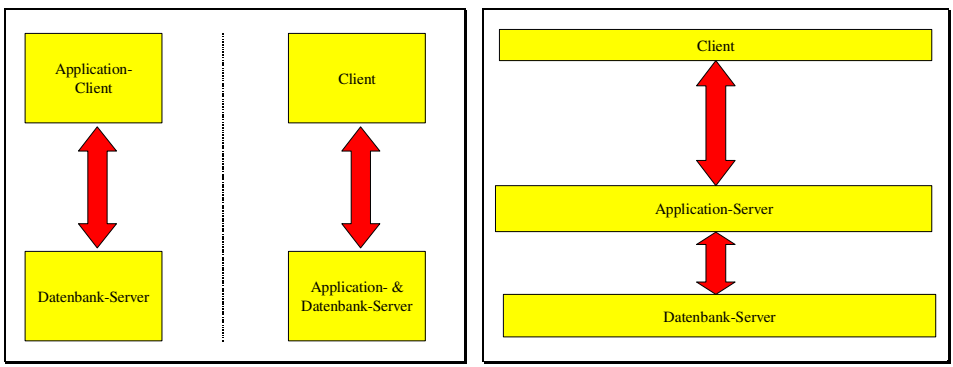


Abbildung 3.3: Zusammenfassung der Anwendungslogik in Komponenten/Schichten

3.2 Softwarekomponenten

Obwohl die grundlegenden Technologien rund um die komponentenbasierte Softwareentwicklung mittlerweile bewährt sind und obwohl die unterstützenden Werkzeuge in konsolidierter Version vorliegen, sind wichtige Begriffe nach wie vor nicht präzise definiert. Zwei der gängigen Erklärungen zum Begriff der Komponente lauten beispielsweise:

Eine Softwarekomponente ist ein Baustein mit vertraglich spezifizierten Schnittstellen und nur ausdrücklichen Kontextabhängigkeiten. Eine Softwarekomponente kann unabhängig verwendet werden und leicht mit anderen Komponenten integriert werden. [65]

Eine Komponente ist ein Stück Software, das klein genug ist, um es in einem Stück zu erzeugen und pflegen zu können, groß genug ist, um eine sinnvolle Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen sowie mit standardisierten Schnittstellen ausgestattet ist, um mit anderen Komponenten zusammenzuarbeiten. [29]

Aus Sicht einer komponentenbasierten Softwarelandschaft ist das wesentliche Merkmal einer Komponente, dass sie leicht mit anderen Komponenten gekoppelt werden kann. Darüber hinaus gibt es eine Reihe weiterer Eigenschaften, die Komponenten haben sollten:

- **Wohldefinierter Zweck:** Eine Komponente soll einem bestimmten Zweck dienen, der oftmals umfassender und abstrakter ist als der Dienst eines einzelnen Objektes. Andererseits stellt eine Komponente aber auch keine eigenständige Anwendung dar, vielmehr ist in der Regel die Einbettung in einem Anwendungskontext erforderlich. Eine Komponente ist also ein Spezialist für eine bestimmte Aufgabe. Zum Beispiel kann eine Komponente »Rechtschreibprüfung« in vielen Anwendungen sinnvoll eingesetzt werden. Es ist nicht nötig, eine komplette Textverarbeitung mit fest verdrahteter Rechtschreibkontrolle einzusetzen, wenn die übrige Funktionalität nicht benötigt wird. Die Komponente allein ist aber auch nicht »lebensfähig«.
- **Kontextunabhängigkeit:** Das Zusammenspiel von verteilten Komponenten sollte möglichst unabhängig von Rahmenbedingungen funktionieren, die durch die verwendeten Programmiersprachen, Betriebssysteme, Netzwerktechnologien und Entwicklungsumgebungen gesetzt werden. Alles was man über eine Komponente wissen muss, um über ihre Benutzung zu entscheiden, sollte in der Schnittstelle der Komponente beschrieben werden.
- **Portabilität und Programmiersprachenunabhängigkeit:** Eine Komponente soll in nahezu jeder Programmiersprache entwickelt werden können. Es kann sich dabei sowohl um objektorientierte, aber auch um prozedurale, funktionale oder logische Sprachen handeln. Entscheidend ist nur, dass die fertig übersetzte Komponente universell verwendbar ist. Komponenten sollen plattformunabhängig eingesetzt werden können. Das bedeutet, dass Komponenten ohne Portierung auf einer Vielzahl von Plattformen zum Einsatz kommen können.
- **Ortstransparenz:** Die Nutzbarkeit der Dienste einer Komponente sollte unabhängig vom Ausführungskontext der Komponente sein, d.h. es sollte für den Benutzer keine Rolle spielen, ob sich die Komponente auf dem lokalen Rechner, in einer anderen Prozessumgebung oder auf einem verteilten System in einem entfernten Knoten befindet. Das Verhalten der Komponente ist immer gleich.
- **Trennung von Schnittstelle und Implementierung:** Eine Softwarekomponente und ihre Schnittstelle(n) sollten vollständig unabhängig von der Implementierung spezifiert werden können. Die von ihr angebotenen Dienste sollten wohldefiniert und dem Benutzer bekannt sein. Die Art und Weise der Implementierung ist für den Benutzer transparent. Zur Kommunikation mit der Außenwelt sollten ausschließlich diese exakt spezifizierten Schnittstellen benutzt werden. Ein direkter Zugriff auf das Innenleben einer Komponente ist ausdrücklich nicht erwünscht und in der Regel auch nicht möglich.
- **Selbstbeschreibungsfähigkeit:** Eine Komponente sollte einen Mechanismus unterstützen, der eine Selbstbeschreibung der angebotenen Dienste ermöglicht. Dabei sollten mindestens die Signaturen der Methoden und Attribute abrufbar sein. Selbstbeschreibende Schnittstellen sind eine wichtige Voraussetzung für die Laufzeitkopplung (späte Bindung) von Komponenten und dienen einer besseren Wiederverwendbarkeit.
- **Sofortige Einsatzbereitschaft (plug&play):** Sofort nach der Verfügbarkeit sollte eine Komponente installierbar und einsatzfähig sein. Wünschenswert ist in diesem

Zusammenhang die Fähigkeit zur Selbstinstallation und -registrierung in der zugrunde liegenden Infrastruktur (Betriebssystem, Namens- und Verzeichnisdienste der Middleware).

- **Integrations- und Kompositionsfähigkeit:** Mehrere Komponenten sollten zu einer neuen Komponente zusammengesetzt werden können. Generell müssen Komponenten untereinander über ihre Schnittstellen interagieren können. Die Komposition und Interaktion sollte nicht nur statisch zur »Kompositionszeit«, sondern vor allem auch dynamisch zur Laufzeit möglich sein. Dabei kann eine Komponente sowohl Client einer anderen Komponente sein, als auch gleichzeitig als Server für viele weitere Komponenten fungieren. Die benötigten Basisdienste (Nachrichtenversand, Auffinden einer anderen Komponente usw.) müssen durch die Komponentenarchitektur als zugrunde liegende Infrastruktur und Middleware zur Verfügung gestellt werden.

- **Wiederverwendbarkeit:** Komponenten sollten so gestaltet sein, dass sie in ihrem Anwendungskontext möglichst einfach wieder verwendet werden können. Mit Komponenten als anwendungsnahen Bausteinen wird erstmals das Anwendungsniveau erreicht. Entsprechend kann Wiederverwendung auf Anwendungsniveau ermöglicht werden.

- **Konfigurierbarkeit, Anpassbarkeit:** Es ist bei der Entwicklung und Freigabe einer Komponente nicht vollständig vorhersehbar, in welchen Situationen sie später einmal zum Einsatz kommt. Daher sollten Komponenten über Parameter konfigurierbar sein, um sie einfach und schnell an neue Situationen anpassen zu können. Zum Beispiel sollte eine Komponente zur Rechtschreibkontrolle an ein neues Regelwerk anpassbar sein, ohne die Implementierung ändern zu müssen. Durch eine weitgehende Konfigurierbarkeit kann eine erneute Übersetzung und ein wiederholter Vertrieb der Komponente vermieden werden.

- **Bewährtheit:** Komponenten sollten ausgiebig erprobt und sorgfältig getestet werden, um eine hohe Zuverlässigkeit zu gewährleisten. Es kann daher nicht einfach jede Menge von Objekten als Komponente bezeichnet werden.

- **Binärcode-Verfügbarkeit:** Komponenten liegen in ausführbarer, auf verschiedenen Plattformen lauffähiger Form vor. Sie können ohne Quelltext ausgeliefert werden. Der Binärcode verhält sich auf allen unterstützen Plattformen gleich.

Für keines der gängigen Komponentenmodelle ist gewährleistet, dass die Komponenten, die den Vorgaben des Komponentenmodells folgen, die genannten Eigenschaften vollständig erfüllen. Während viele Eigenschaften von der Sorgfalt und der Arbeitsweise der Komponentenentwickler abhängen, sind andere Eigenschaften durch die Verwendung einzelner Komponentenmodelle geradezu ausgeschlossen. Dies gilt insbesondere für die Eigenschaften der Programmiersprachenunabhängigkeit, der Plattformunabhängigkeit und der einfachen Portierbarkeit. Komponenten, die dem Komponentenmodell Enterprise JavaBeans folgen, liegen in Java vor — nicht in irgendeiner anderen Programmiersprache. Und Komponenten, die Microsofts Komponentenmodell COM entsprechen, laufen auf Microsoft-Plattformen. Nichtsdestrotrotz sind die genannten Kriterien ein gu-

ter Maßstab dafür, inwiefern Softwareteile den Grundideen der komponentenbasierten Entwicklung entsprechen.

Die genannten Kriterien machen keine konkrete Vorgabe bezüglich des Abstraktionsniveaus von Komponenten. Je nach Anwendungssituation kann es zu unterschiedlichen Komponenten kommen. Im Kontext der Entwicklung eines Bestandsführungssystems kann es sinnvoll sein, Komponenten »Partner« und »Vertragsbasisdaten« zu entwerfen. Im Kontext der Entwicklung eines Partnermanagement-Systems kann es zu kleinteiligeren Komponenten kommen. Generell gilt, dass eine Komponente ein Stück Software ist, das klein genug ist, um es erzeugen und verwalten zu können, groß genug ist, um es einsetzen und unterstützen zu können, und das festgelegte Interfaces zur Interoperabilität enthält.

Für die Beschreibung von Komponenten unterscheiden wir zwischen dem Aspekt der Komponentenspezifikation und dem Aspekt der Komponentenrealisierung (vgl. Abbildung 3.4). Die Komponentenspezifikation (oft auch als fachliche Komponente bezeichnet) beschreibt die Funktionalität einer Komponente und abstrahiert von den (technischen) Details der Realisierung dieser Funktionalität. Es werden der angebotene Service (die Dienstleistung) und die einzelnen (Service-)Funktionen benannt, aus denen der Service sich zusammensetzt. Jeder Service faßt mehrere Funktionen, die von der Komponente unterstützt werden, unter einer Sammelbezeichnung zusammen. Der Begriff des Service hilft, die möglicherweise große Menge der angebotenen Funktionen einer Komponente zu strukturieren und zu kleineren und überschaubaren Funktionen-Bündeln zusammenzufassen, die unmittelbar fachlich zusammengehören. Eine Komponente kann mehrere Services anbieten. Die Komponentenspezifikation ist außerdem nützlich, um in einer Komponentenbibliothek nach Komponenten zu suchen, die eine gewünschte Funktionalität bereitstellen können (mit anderen Worten: die eine bestimmte vorgegebene Spezifikation erfüllen). Es kann dabei mehrere Komponenten geben, die in einigen Service-Funktionen oder in der kompletten Spezifikation übereinstimmen.

Die folgenden Tabellen (3.1, 3.2) zeigen ein Beispiel für die Spezifikation einer Komponente, die die genannten Begriffe aufgreift und das Zusammenspiel zwischen Schnittstellen, Services und Service-Funktionen illustriert.

Die Komponentenrealisierung (oft auch als technische Komponente bezeichnet) beschreibt, auf welche Weise der von einer Komponente angebotene Service realisiert ist (das »Wie?«). Der gesamte Service wird durch eine Menge von Interfaces repräsentiert, deren Methoden auf Programmebene aufgerufen werden können. Die Komponentenrealisierung beschreibt eine Komponente durch die Interfaces und deren Methoden (inklusive ihrer Aufrufparameter). Das konkrete Aussehen einer realisierten Komponente ist abhängig vom gewählten Komponentenmodell.

Eine weitere Unterscheidung von technischen Komponenten ist die Unterscheidung in so genannte Client-Komponenten und Server-Komponenten. Client-Komponenten realisieren Benutzungsoberflächen und unmittelbar damit in Verbindung stehende Plausibilitätsprüfungen, jedoch keine weitergehenden Teile der Anwendungslogik. Server-Komponenten realisieren die Geschäftsobjekte mit ihren Daten und Methoden.

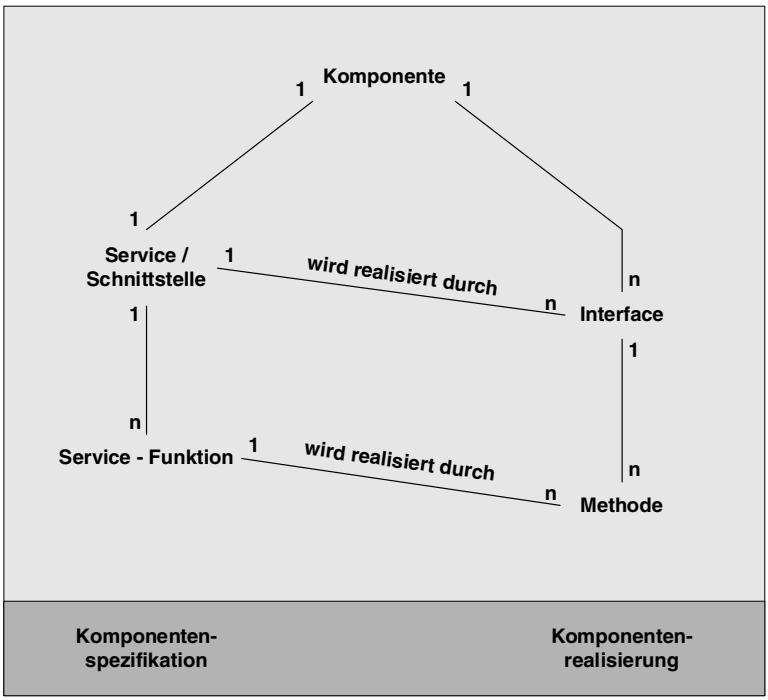


Abbildung 3.4: Komponentenspezifikation und Komponentenrealisierung

Sie repräsentieren die Anwendungslogik und sind von der Repräsentation auf bestimmten Endgeräten getrennt. Abbildung 3.5 verdeutlicht diesen Zusammenhang. Zu einem Geschäftsobjekt (zur Verfügung gestellt durch einen Applikationsserver) gibt es eine Reihe von präsentationskanalspezifischen Client-Komponenten, die sich an verschiedene Anwendergruppen wenden. Jede dieser Anwendergruppen sieht unterschiedliche Repräsentationen der Geschäftsobjekte. Die Darstellung ist auf die spezifischen Anforderungen der unterschiedlichen Anwendergruppen hin angepasst.

Der besondere Vorteil einer komponentenbasierten Software-Architektur liegt darin, dass Komponenten unterschiedlichen Ursprungs miteinander kombiniert werden können. Hierbei können existierende Softwareteile berücksichtigt werden, die als Komponenten verkleidet werden (wrapping). Genauso gut können neue, eigenentwickelte und zugekaufte Komponenten eingebunden werden. Auf diese Weise ist es möglich, dass eine komponentenbasierte Architektur als Zielarchitektur dient, der man sich schrittweise annähert. Hierbei können bestehende Komponenten mit neuen verbunden werden. Bestehende Komponenten können dann — je nach Bedarf — schrittweise gegen neue ausgetauscht werden. Solange die Interfaces unverändert bleiben, ist ein solcher Austausch transparent für die service-nutzenden Komponenten. Ein anderer Vorteil ist, dass Komponenten plattformübergreifend eingesetzt werden können. Da die gängigen Komponentenmodelle den Binärcode standardisieren, der die Implementierung von Komponenten darstellt, ist sichergestellt, dass Komponenten sich auf verschiedenen Plattformen

Spezifikation: Kundenverwaltung

Komponentenname	Kundenverwaltung
Kurzbeschreibung	Verwaltung von Kunden
Langbeschreibung	Mit dieser Komponente werden Kunden verwaltet, indem Adressen und Bankdaten in Beziehung gesetzt werden können. Der Adressenbestand und der Bankdatenbestand können getrennt gepflegt werden. Der Service »Adressdaten-Verwaltung« stellt die Funktionen »Adresse Anlegen«, »Adresse Löschen«,[...] zur Verfügung. Der Service »Bankdaten-Verwaltung« bietet die Funktionen [...]
Verantwortlicher	Herr Nehm, Abt. Kundenmanagement
Internes Objektmodell der Komponente	Nicht verfügbar
Rolle der Rolle	Server
Verwendete Komponenten	Keine
Aufrufende Komponenten	Lagerverwaltung, Finanzverwaltung
Service	
Service-Name	Adressdaten-Verwaltung
Kurzbeschreibung	Verwalten eines Adressbestandes
Langbeschreibung	Dieser Service bietet Funktionen an, um einen Adressenbestand zu bearbeiten. Die angebotenen Funktionen umfassen Einfügen, Löschen und Drucken von Adressdaten.
[Auflistung der Service-Funktionen, siehe Tabelle 3.2]	
Service	
Service-Name	Bankdaten-Verwaltung
Kurzbeschreibung	Verwalten eines Bestandes von Bankkonten und -bewegungsdaten
[...]	

Tabelle 3.1: Services einer Komponente

identisch verhalten und dass Portierungen und Mehrfachimplementierungen vermieden werden können.

Ein weiterer Vorteil komponentenbasierter Softwarearchitekturen liegt in der mehrfachen Wiederverwendung von Bausteinen und in der damit verbundenen Steigerung der Produktivität. Durch die Möglichkeit der räumlichen Verteilung von Komponenten können diese an einem einzigen Ort stationiert sein und die angebotenen Services von beliebig vielen anderen Komponenten auf beliebigen Plattformen in Anspruch genommen werden. Neben den unmittelbaren Einsparungen, die sich daraus ergeben, dass jeder Service nur genau einmal realisiert werden muss, ergeben sich qualitative Vorteile

Auflistung Service-Funktionen zu Adressdaten-Verwaltung

Service-Funktion	
Service-Funktionsname	Adresse_Anlegen
Kurzbeschreibung	Zu einem existierenden Kunden wird eine neue Adresse angelegt.
Langbeschreibung	Diese Service-Funktion legt eine neue Adresse an. Die Adresse kann von einem der folgenden Typen sein: Hauptadresse, Zusatzadresse, Sonderadresse oder Mitgliedsadresse. [. . .]
Parameterliste	
Attribut	IN
Parametername	Kundennummer
Beschreibung	Nummer des Kunden
Attribut	IN
Parametername	Adresse
Beschreibung	Neue Adresse
Vorbedingung	Es muss ein Kunde ohne Adresse bestehen.
Nachbedingung	Eine neue Adresse wurde erzeugt. Der Kunde ist mit der neuen Adresse verbunden.

Service-Funktion	
Service-Funktionsname	Adresse_Löschen
Kurzbeschreibung	Die Adresse eines Kunden wird gelöscht.
[…]	

Tabelle 3.2: Servicefunktionen verfeinert bis auf Parameterebene

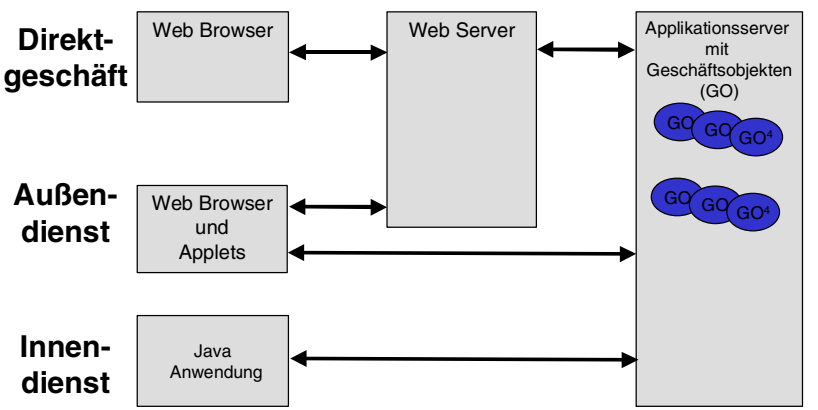


Abbildung 3.5: Client-Komponenten, Server-Komponenten und ihre Ausprägungen

dadurch, dass tatsächlich immer der gleiche Service benutzt werden kann und somit unterschiedliche Verhalten von Servicevarianten vermieden werden.

Ein Komponentenmodell legt fest, welche Anforderungen eine Komponente erfüllen muss und zwar sowohl in syntaktischer, als auch in semantischer Hinsicht. Ein Komponentenmodell macht damit Vorgaben bezüglich der Gestaltung von konkreten Komponenten. Alle Komponenten, die diesen Vorgaben folgen, können mit geringem Aufwand zusammengebaut werden. Um im Bild der Legosteine zu bleiben: Ein Komponentenmodell legt fest, wie die Steckverbindungen aussehen, über die die Komponenten zusammengebaut werden können. Die gängigen Komponentenmodelle sind JavaBeans, Enterprise JavaBeans und COM [30].

Komponenten tauschen Nachrichten miteinander aus (zum Zweck des Methodenaufrufes). Aufrufe über die Grenzen einzelner Anwendungen hinweg erfolgen gemäß eines Standards. Beispiele für solche Standards sind RPC, RMI, RMI über IIOP, und Corba Services. Mithilfe eines Vermittlers (Broker) können Komponenten miteinander kommunizieren, ohne dass sie wissen müssen, wo die aufzurufenden Objekte residieren. Unter Middleware versteht man Softwaresysteme, die benötigt werden, um verteilte Anwendungen oder Komponenten miteinander kommunizieren zu lassen. Middleware ist anwendungsunspezifisch, sie weiß nichts von dem einzelnen Anwendungskontext. In aller Regel werden Middleware-Systeme über eine Programmierschnittstelle aufgerufen. Zu den typischen Middleware-Systemen gehören Netzwerkbetriebssysteme, WWW-Browser, verteilte Datenbankmanagement-Systeme, Workflow-Managementsysteme, Groupware-Systeme, Broker und traditionelle Enterprise Application Integration-Werkzeuge. Zu den typischen Funktionalitäten von Middleware-Systemen sind die folgenden zu zählen: Datentransfer, Kommandoübermittlung (synchroner und asynchroner Aufruf), Empfang von Anfragen, Vermeidung von Verklemmungen, Etablierung und Beendigung von Kommunikationskanälen und spezifischere Dienste aus dem Workflow-/Groupware-Bereich.

3.3 Enterprise JavaBeans als Komponentenmodell

Enterprise JavaBeans (EJB) sind ein Komponentenmodell, das inbesondere die Realisierung von Komponenten unterstützt, die Anwendungslogik realisieren. Sun Microsystems definiert das Komponentenmodell EJB wie folgt:

The Enterprise JavaBeans architecture is a component architecture for the development and deployment of object-oriented distributed enterprise-level applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and deployed on any server platform that supports the Enterprise JavaBeans specification. [63]

Ziel des Komponentenmodells EJB ist es, Entwickler bei der Erstellung von Server-Komponenten zu unterstützen. Entwickler sollen die Möglichkeit bekommen, sich auf die Realisierung der eigentlichen Anwendungslogik zu konzentrieren. Querschnittsaufgaben, die bei der Realisierung eines jeden Informationssystems anfallen, sollen den

Entwicklern weitgehend abgenommen werden, indem vorgefertigte Lösungen für solche Aufgaben (Transaktionsmanagement, Persistenz, Sicherheit usw.) in Form von Services angeboten werden. Die Realisierung der einzelnen Services ist Gegenstand von so genannten EJB-Applikationsservern. EJB-Applikationsserver sind Infrastruktursysteme, die Komponenten, die der EJB-Spezifikation folgen, unterstützen.

Der Versuch, möglichst viele und möglichst mächtige Services anzubieten, verdeutlicht, inwiefern sich ein Server-Komponentenmodell von einem Client-Komponentenmodell unterscheidet. Ein Client-Komponentenmodell verfolgt zwar auch das allgemeine Ziel, kompatible Komponenten zu ermöglichen, hat dabei aber eher die Erstellung grafischer Oberflächen im Fokus. JavaBeans sind ein typisches Beispiel für ein Client-Komponentenmodell. Das Komponentenmodell JavaBeans unterstützt den Prozess der Oberflächenerstellung aus vordefinierten Elementen und das Anbinden solcher Oberflächen an die Anwendungslogik. Die Erstellung von Komponenten, die die Anwendungslogik selbst realisieren, wird jedoch nicht ausdrücklich unterstützt. Diese Unterscheidung zwischen Enterprise JavaBeans und JavaBeans wird auch durch das folgende Zitat verdeutlicht:

Enterprise JavaBeans … cannot be manipulated by a visual Java IDE in the same way that JavaBeans components can. Instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server. [20]

Die Idee von server-seitigen Komponentenmodellen im Allgemeinen und EJBs im Besonderen passt ideal zu einem Trend, der sich in der Softwaretechnologie seit Jahrzehnten beobachten lässt. Dieser Trend ist der zur Entlastung der Anwendungsentwickler von technischen Aufgaben, die von der Konzentration auf die Anwendungslogik ablenken. Anwendungsentwickler sollen sich darauf konzentrieren können, das knappe Fachwissen in Software umzusetzen. Sie sollen das tun können, ohne sich dabei über die immer wieder gleichen Transaktionsprobleme Gedanken machen zu müssen und ohne die Abbildung von Klassenstrukturen auf relationale Datenbank-Managementsysteme immer wieder neu erfinden zu müssen. Wie bei allen Lösungen, die den Anspruch erheben, allgemein gültig zu sein, sind auch die Services für EJB-Anwendungen selten ideal für die einzelne Anwendungssituation. In der Regel sind sie aber immer noch besser als die Lösungen, die Fachleute (die ja nicht notwendigerweise Experten in den eher technisch geprägten Fragestellungen sind) erfinden und vor allem reichen sie in der Regel aus.

Die Spezifikation von Enterprise JavaBeans ist von Sun Microsystems erarbeitet worden und liegt seit März 1998 in der Version 1.0 vor. In den letzten Jahren hat es eine Reihe von Folgeversionen dieser Spezifikation gegeben. Eine ganz wichtige Folgeversion ist EJB 2.0. In EJB 2.0 sind eine ganze Reihe von Ergänzungen realisiert, die insbesondere im Hinblick auf die Performanz von EJB-basierten Anwendungen grundlegend und teilweise unverzichtbar sind. An den grundlegenden Zielen von EJBs hat sich seit der Freigabe der Version 1.0 allerdings nichts Wesentliches geändert. Diese Ziele sind:

- Mit EJB soll ein Standard-Komponentenmodell für die Entwicklung von verteilten objektorientierten Anwendungen in Java vorgegeben werden.

- Dem Entwickler soll es ermöglicht werden, sich bei der Entwicklung von Anwendungen vollständig auf die Anwendungslogik zu konzentrieren.
- EJB-basierte Anwendungen folgen der »write-once, run anywhere«Philosophie von Java. Das bedeutet, ein Enterprise Bean kann einmalig entwickelt und ohne Modifikation des Programmcodes oder Rekompilation auf unterschiedlichen Plattformen eingesetzt werden.
- In der EJB-Architektur sind die Aufgaben und Verantwortlichkeiten von Client, Server und den individuellen Komponenten klar definiert.
- EJBs können flexibel zwischen verschiedenen Standorten ausgetauscht werden. Die Transformation in ein und aus einem Austauschformat erfolgt automatisch.

Das große Interesse an EJB-basierten Anwendungen basiert auch darauf, dass das Zusammenwachsen von klassischen Informationssystemen und e-Business/e-Commerce-Lösungen zu flexibel verteilbaren Softwaresystemen führt. Die folgende Einschätzung veranschaulicht diesen Mix von Gründen für die Auseinandersetzung mit EJBs:

During the early 90s, traditional enterprise information system providers began responding to customer needs by shifting from the two-tier, client-server application model to more flexible three-tier and multi-tier application models. The new models separated business logic from system services and the user interface, placing it in a middle tier between the two. The evolution of new middleware services — transaction monitors, message-oriented middleware, object request brokers, and others — gave additional impetus to this new architecture. And the growing use of the internet and intranets for enterprise applications contributed to a greater emphasis on lightweight, easy to deploy clients. [20]

Eine weitere für das Verständnis von EJBs unverzichtbare Unterscheidung ist die zwischen Entity Beans und Session Beans [46]. Entity Beans realisieren die statischen Anteile von Anwendungen, sie repräsentieren die Informationen, um die es letztlich geht. Eine Entity Bean kann von mehreren Clients genutzt werden und wird durch einen Primary Key identifziert. Entity Beans sind transaktionsorientiert und recovery-fähig. Eine Entity Bean kann selbst die Verantwortung für ihre persistenten Daten übernehmen oder die Aufgabe an den Container delegieren.

Session Beans verkapseln Teile von Geschäftsprozessen, sie fassen interaktive Teile zusammen, die in einem engen logischen Zusammenhang stehen. Session Beans repräsentieren somit eher die Dynamik einer Anwendung. Eine Session Bean repräsentiert den Zustand der Kommunikation mit einem Client. Eine Session Bean wird durch einen Client erzeugt, ist exakt nur diesem Client zugeordnet und existiert in der Regel nur während der Zeitdauer einer einzelnen Client/Server-Session. Eine Session Bean führt die Operationen für den Client aus; das können zum Beispiel Datenbank-Operationen oder Berechnungen sein. Eine Session Bean kann transaktionsorientiert sein, ist aber nicht recovery-fähig. Persistente Daten einer Session Bean sind von ihr selbst zu verwalten.

Abbildung 3.6 veranschaulicht das Zusammenspiel zwischen den verschiedenen Arten von Beans. Sie zeigt, wie auf EJBs von außen zugegriffen werden kann und welche Arten von Schnittstellen dazu benutzt werden.

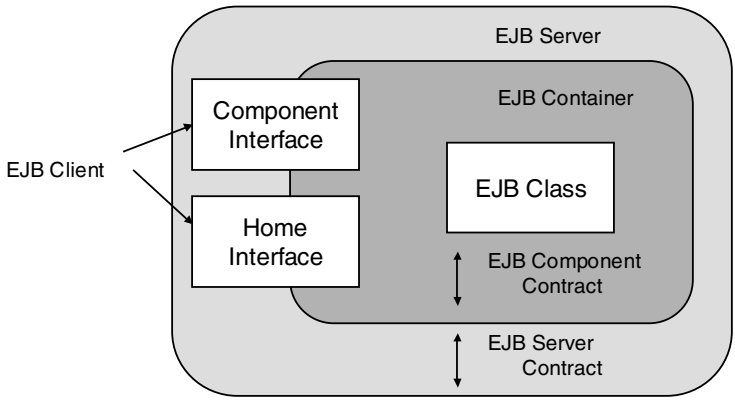


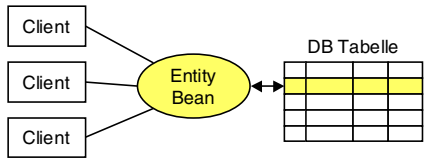
Abbildung 3.6: Schema einer EJB-basierten Architektur

Die in Abbildung 3.6 dargestellten Bestandteile einer EJB-basierten Architektur haben folgende Bedeutung:

- **Schnittstellen zu EJBs:** Ein EJB-Container stellt zwei Schnittstellentypen zur Verfügung, über die der Client mit den EJB kommunizieren kann. Über das Home Interface werden Funktionen zur Verwaltung einer EJB bereitgestellt. Dazu gehören das Erzeugen und Initialisieren einer EJB, das Löschen einer EJB, das Abfragen von Metadaten zur EJB und bei Entity Beans eine Funktion zur Suche nach EJBs. Über das Component Interface (oder in der Diktion des 1.1 Standard: Remote Interface) wird die eigentliche Anwendungslogik eines EJB implementiert.
- **EJB Clients:** Der EJB-Client nutzt die Operationen, die ein EJB anbietet. Der Client findet den EJB-Container durch das Java Naming and Directory Interface (JNDI). Der Client greift niemals direkt auf die Bean-Methoden zu, sondern immer über container-generierte Methoden, die ihrerseits die eigentlichen Bean-Methoden initiieren.

Die Prinzipien der Differenzierung zwischen Entity und Session Beans wird in Abbildung 3.7 zusammenfassend dargestellt. Es ist angedeutet, dass Entity Beans ihren direkten Niederschlag in einer persistenten Datenhaltung finden (oft in einer relationalen Datenbank) und dass Session Beans den koordinierten Zugriff auf eine Menge von Entity Beans ermöglichen, die in einem konkreten Kontext gemeinsam bearbeitet werden müssen.

- Entity Beans
  - Einfache Objekte
  - Komplexe Objekte durch Aggregate
  - Kapselung von Host-Datenbanken



- Session Beans
  - Berechnungen als Dienste
  - Kapselung von Host-Diensten
  - evtl. „lange“ Transaktionen

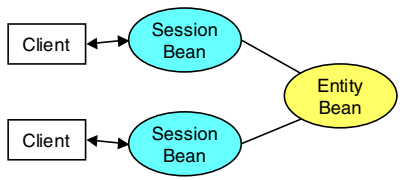


Abbildung 3.7: Abbildung Entity Beans versus Session Beans

3.4 Services der EJB-Applikationsserver

Ein EJB-Applikationsserver hat zweierlei Grundaufgaben. Zum einen soll er den neuen Anwendungen möglichst komfortable Services anbieten und zum zweiten soll er die Integration mit existierenden Anwendungen und Datenbeständen ermöglichen. In der Praxis zeigt sich immer wieder, dass die Realisierung des Applikationsserver-Zugriffs auf existierende Anwendungen besonders aufwändig und fehleranfällig ist.

Ein Applikationsserver stellt die Laufzeitumgebung für Server-Komponenten dar. Er empfängt Nachrichten von den Clients und ruft die dazu gehörenden Methoden der Server-Komponenten auf. Dazu benötigte Objekte werden vor einem Methodenaufruf instanziiert und mit ihren persistenten Daten initialisiert. Während ihrer Lebenszeit werden die Objekte in einem Cache gehalten, was wiederholte Zugriffe auf diese Objekte beschleunigt. Werden Objekte verändert, so synchronisiert der Applikationsserver die dazugehörenden Einträge in dem Persistenzspeicher. Dies geschieht in der Regel transaktionsgesichert.

Im Folgenden wird ein kurzer Überblick über die Services gegeben, die von den gängigen EJB-Applikationsservern angeboten werden. Nicht alle diese Services werden von allen EJB-Applikationsservern angeboten. Oft ist es sogar so, dass die Verfügbarkeit der Services ein Auswahlkriterium für einen konkreten Applikationsserver sein kann. Angesichts der großen preislichen Bandbreite zwischen High-End-Produkten (hier werden je nach Nutzerzahl 6-stellige Euro-Lizenzbeträge fällig) und kostenfrei einsetzbaren Produkten, differieren die Mengen der jeweils angebotenen Services erheblich. Bei einer konkreten Produktbewertung ist jedoch nicht nur die Verfügbarkeit eines Services, sondern auch dessen Qualität und seine genaue Ausgestaltung zu berücksichtigen. An dieser Stelle entsteht dann zuweilen durchaus die Situation, dass kostenfreie Produkte in einzelnen Aspekten den teuren Applikationsservern überlegen sind.

- **Naming-Service:** Der Naming-Service stellt sicher, dass aufrufende Komponenten nicht über den Aufenthaltsort von aufzurufenden Komponenten und Komponenten-

instanzen informiert sein wissen. Sie müssen nicht wissen, ob ein Aufruf innerhalb des eigenen Adressraums erfolgen muss oder ob es sich um einen entfernten Aufruf handelt. Der Naming-Service ist damit die wesentliche Unterstützung für die Gewährleistung der Ortstransparenz.

- **Lebenszyklus-Management:** Einzelne EJBs brauchen sich nicht explizit um Prozessallokation, Thread-Management, Objekterzeugung oder -vernichtung zu kümmern. Der Lebenszyklus-Service bietet die entsprechende Unterstützung.
- **Zustandsmanagement:** Wenn EJBs im Rahmen eines Geschäftsprozesses verschiedene Zustände annehmen, dann ist der EJB-Zustand zu verwalten. Dies trifft insbesondere dann zu, wenn mehrere Methoden des EJB nacheinander aufgerufen werden (und auch nur in dieser Reihenfolge aufgerufen werden sollen).
- **Sicherheit:** Einzelne EJBs müssen sich nicht um die Authentifizierung oder Autorisierung des Clients kümmern. Zur Prüfung wird der Service benutzt.
- **Persistenz:** Einzelne EJBs müssen sich nicht explizit darum kümmern, wie Daten in der DB abgelegt oder aus ihr herausgeholt werden. Stattdessen kann festgelegt werden, dass EJBs per Container-Managed Persistence persistent gemacht werden. Die Abbildung auf eine — in der Regel — relationale Datenbank erfolgt dann automatisch.
- **Transaktionsmanagement:** Einzelne EJBs müssen sich nicht um die Definition von Transaktionsgrenzen oder die Implementierung von Transaktionsmodellen kümmern. Stattdessen bietet der Service verschiedene Transaktionsmodelle an.
- **Ressourcen-Pooling:** Knappe Ressourcen wie Threads, Client-Connections, DBMS-Connections und Caches werden vom Applikationsserver gehalten. Dies geschieht so, dass diese Ressourcen einer anfordernden Komponente sehr schnell und ohne weiteren Administrations-Overhead zur Verfügung gestellt werden können.
- **Load-Balancing und Failover bei Applikationsserver-Clustern:** Applikationsserver sind skalierbar, weil sie mit geringem Aufwand zu Clustern zusammengeschaltet werden können. Im Falle eines solchen Clusters ist es wichtig, die Last so zu verteilen, dass Engpässe weitgehend vermieden werden. Im Fehlerfall muss dafür gesorgt werden, dass die Aufgaben eines ausgefallenen Applikationsservers dynamisch von einem anderen übernommen werden. Die Load-Balancing- und Failover-Services bieten genau diese Unterstützung.

Ein EJB-Server sorgt dafür, dass dieses Services von den einzelnen EJBs genutzt werden können. Er bietet eine Umgebung, in der aus EJB bestehende Anwendungen ausgeführt werden können. Er managed Mengen von EJBs. Diese werden in Containern verwaltet. Ein EJB-Container (manchmal nur als Container bezeichnet) umfasst eine Menge logisch zusammengehörender EJBs. Eine EJB-Klasse ist genau einem EJB-Container zugeordnet und ein EJB-Container verwaltet genau eine EJB-Klasse. Der Container-Managed den Lebenszyklus der Objekte, implementiert Sicherheitsmechanismen für die Objekte und koordiniert verteilte Transaktionen und die Persistenz von Objekten. Dieser Zusammenhang wird in Abbildung 3.8 veranschaulicht:

– Client
  - beliebig, muss nicht Java sein

– EJB Server
  - kann mehrere Protokolle unterstützen (RMI, IIOP, DCOM)

– Container
  - transparent für Client
  - verwaltet Beans (Lebenszyklus, Zustand, ...)

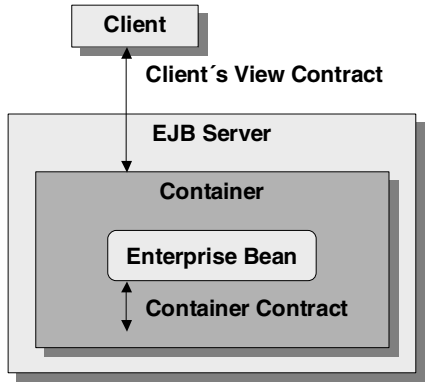


Abbildung 3.8: EJB Server, Container, EJBs und ihr Zusammenspiel

Durch den Einsatz eines Applikationsservers müssen viele Infrastrukturaufgaben nicht mehr programmiert werden. Der Applikationsserver übernimmt sie automatisch bzw. wird dazu konfiguriert (deskriptive statt programmiertechnische Realisierung).

4 Enterprise JavaBeans 2.0

Die Vorläufer des EJB 2.0-Standards sind die Versionen 1.0 (21. März 1998) und 1.1 (17. Dezember 1999). Gegenwärtig aktuell ist der 2.0-Standard als »Final Release« (22. August 2001). Er ersetzt den »Proposed Final Draft 1« (23. Oktober 2000) und den »Proposed Final Draft 2« (19. April 2001).

In dem ersten »Draft« waren vor allem die Dependent Objects strittig. Die Gründe dafür sind in Abschnitt 8.4.1 skizziert. Dependent Objects wurden daraufhin aus dem Draft entfernt. Anstelle dessen wurden `ejbSelect`-Methoden und Local Interfaces eingeführt.

In den einleitenden Kapiteln der Spezifikation [18, S. 25f] werden die folgenden Neuerungen gegenüber der Version 1.1 hervorgehoben:

- **Integration von Message Oriented Middleware:** Durch Message-Driven Beans ist ein weiterer Bean-Typus zu den bestehenden Entity und Session Beans gekommen. Durch diese Beans wird asynchrone Verarbeitung ermöglicht und der Java Message Service in die J2EE integriert. Abschnitt 4.1 gibt einen Einblick in die Konzepte des Java Message Service und der Message-Driven Beans. Parallelisierung als Nebeneffekt der asynchronen Verarbeitung wird zur Umsetzung einer Implementierungsvariante in Abschnitt 8.5 ausgenutzt.
- **Erweiterte Container-Managed Persistence (CMP):** Durch Container-Managed Persistence wird die Speicherung einer Entity Bean im Deployment Descriptor *deklarativ* bestimmt. Ein Generatorwerkzeug des Containerherstellers erzeugt aus der Deklaration Java-Klassen, die die Persistenzfunktionalität umsetzen. Demgegenüber muss der Bean Provider (Entwickler der Bean) bei Bean Managed Persistence selbst für die Implementierung der Persistenzfunktionalität sorgen. Die Container-Managed Persistence nach dem EJB 2.0-Standard ist wesentlich erweitert worden, wodurch vor allem die Entwicklungszeit von Entity Beans verkürzt wird. Zu den wichtigsten Erweiterungen zählen:
  - Container-Managed Relations (CMR)
  - Cascade-Delete
  - Dependent Value Classes

  Abschnitt 4.3 grenzt Bean Managed Persistence und Container-Managed Persistence voneinander ab und stellt die Container-Managed Persistence sowie ihre neuen Erweiterungen ausführlich vor.
- **Local Interfaces:** Local Interfaces reduzieren die Kommunikationskosten zwischen einer Bean und ihrem Aufrufer zu Lasten der Ortsunabhängigkeit. Mit ihnen ist eine veränderte Bezeichnung der Interfaces zu einer Bean eingeführt worden. Das Konzept hinter den Local Interfaces und die neue Bezeichnung der Bestandteile einer Bean wird in Abschnitt 4.2 dargelegt.
- **Enterprise JavaBeans Query Language (EJB QL):** Die EJB QL ist eine von der *Persistenzschicht unabhängige* Anfragesprache. Sie kann nur für Beans genutzt werden, die per Container-Managed Persistence gespeichert werden. Insbesondere dient

sie nur dem Auffinden von Daten und kann nicht zur Datenmanipulation benutzt werden. Abschnitt 4.4 zeigt die Funktionsweise der EJB QL auf.

- **Select Methods:** Select Methods sind den Finder Methods sehr ähnlich, die im Home Interface einer Bean deklariert werden. Sie dürfen im Gegensatz zu Finder Methods nicht durch die Interfaces nach außen sichtbar gemacht werden und stehen nur zur Nutzung *innerhalb* einer Entity Bean zur Verfügung. Ihr Rückgabewert kann auf assoziierte Beans oder einzelne CMP-Felder getypt sein. Abschnitt 4.4.2 vertieft die Abgrenzung zu Finder Methods.
- **Home Methods:** Neben den Methoden für den Lebenszyklus einer Entity Bean (create, finde, remove) können seit EJB 2.0 beliebige weitere Methoden im Home Interface deklariert werden. Diese Home Methods beziehen sich ähnlich den Klassenmethoden im objektorientierten Entwurf stets auf die Gesamtheit aller Instanzen zu einer Entity Bean. Abschnitt 4.5 erläutert ihre Verwendung.
- **Run-As Security:** Der Applikationsserver verwendet ein Rollenschema, um die Berechtigung zur Ausführung einer Methode zu überprüfen. Dazu wird im Deployment Descriptor deklariert, welchen Rollen die Ausführung einer Methode erlaubt ist.

  Die Möglichkeit, einer Bean selbst eine Identität zuzuweisen, ist neu hinzugekommen. Dadurch kann zwischen der Berechtigung, eine Methode auszuführen, und den Rechten, die eine Methode zur Ausführung benötigt, unterschieden werden. Letztere werden durch die Run-As-Identity angegeben. Abschnitt 4.6 erläutert dies anhand eines Beispiels.
- **Interoperabilität von Applikationsservern verschiedener Herkunft:** In heterogenen Umgebungen können Applikationsserver verschiedener Hersteller zusammentreffen. Seit EJB 2.0 wird ein standardisiertes Protokoll vorgeschrieben, über das Komponenten in Applikationsservern *verschiedener* Hersteller miteinander kommunizieren können. Um das Rad nicht neu zu erfinden, wurde für diesen Zweck das bestehende IIOP (Internet Inter-ORB Protocol) ausgewählt [18, S. 385–408]. Diese Neuerung wird in dem vorliegenden Buch nicht weiter vertieft, da sie für den Bean Provider (Entwickler der Bean) transparent ausgeführt sein sollte.

4.1 Enterprise Messaging

Zu den beiden bekannten Bean-Typen Entity Bean und Session Bean ist in der Version 2.0 der Spezifikation der Typ der Message-Driven Bean hinzugekommen. Durch diesen Typ wird der Java Message Service (JMS) in die Java Enterprise Edition (J2EE) integriert [47]. Im ersten Teil dieses Abschnitts wird daher auf grundlegende Konzepte des JMS eingegangen. Daran anschließend werden wichtige Eigenschaften der Message-Driven Beans vorgestellt.

4.1.1 Java Message Service

Der JMS dient im Wesentlichen nicht dazu, um Nachrichten von einem Nutzer zu einem anderen Nutzer übermitteln zu können, wie oft vermutet wird. Er dient der *asynchronen* Kommunikation zwischen verschiedenen Prozessen. Dieses Konzept der Interprozesskommunikation ist auch als Message-Oriented Middleware verbreitet. Das wohl bekannteste Produkt in diesem Bereich ist MQSeries von IBM. Es wurde erstmals im Jahr 1993 vorgestellt. Der JMS ist also keine besondere Innovation. Durch den JMS wird ein bereits lange bestehendes technisches Konzept für die Java-Welt umgesetzt, um die Möglichkeiten der J2EE als Integrationsplattform zu erhöhen. Zur Zeit unterstützen die folgenden Produkte den JMS: MQSeries (IBM), SonicMQ (Progress), FioranoMQ (Fiorano), iBus(Softwired). Nicht Java-basierte Anwendungen, die diese Produkte benutzen, können damit über die J2EE integriert werden.

Asynchrone Kommunikation hat den Vorteil, dass sie die Kopplung zwischen den kommunizierenden Prozessen vermindert. Bei *synchroner* Kommunikation, beispielsweise einem Methodenaufruf zwischen zwei Java-Klassen, muss der aufgerufene Kommunikationspartner den Aufruf zuerst abgeschlossen haben, bevor der Aufrufende fortfahren kann. Bei asynchroner Kommunikation muss der Aufrufer nicht auf den Aufgerufenen warten.

Aufrufer und Aufgerufener werden in der JMS-Terminologie auch als *Producer* (Versender) und *Consumer* (Empfänger) einer *Message* (Nachricht) bezeichnet. Der Kommunikationskanal, über den die Nachrichten zwischen Producer und Consumer übermittelt werden, wird entweder als *Topic* oder als *Queue* bezeichnet. Das ist abhängig von der Art der Nachrichtenübermittlung. Die Kommunikationskanäle und alle darin enthaltenen Nachrichten werden von einem *JMS-Server* verwaltet. Die beiden Übermittlungskonzepte sind:

- **Publish and Subscribe (pub/sub):** Wird eine Nachricht an mehrere Consumer gleichzeitig übermittelt, so wird der Kommunikationskanal als Topic bezeichnet. Ein neuer Consumer kann sich jederzeit als weiterer Empfänger registrieren lassen (subscribe). Wird über das Topic eine Nachricht veröffentlicht (publish), so erhalten alle registrieten Empfänger ihr eigenes Exemplar. Ein ähnlicher Mechanismus wird für das Event-Handling von GUI-Elementen in Java benutzt.
- **Point-to-Point (p2p):** Auch beim Point-to-Point-Übermittlungskonzept kann es mehrere mögliche Empfänger für eine Nachricht geben. Aus diesen Empfängern wird *genau einer* ausgewählt, dem die Nachricht zugestellt wird. Alle anderen Empfänger erhalten keine Nachricht. Viele Produkte übermitteln die Nachrichten nach den Regeln einer Lastverteilung, obwohl dies nicht von der Spezifikation gefordert wird. Der Kommunikationskanal, über den die Nachrichten verteilt werden, wird beim Point-to-Point Messaging als Queue bezeichnet.

Guaranteed Delivery ist eine wichtige Funktionalität, um Message-Oriented Middleware für unternehmenskritische Anwendungen einsetzen zu können. Durch sie wird sichergestellt, dass Nachrichten tatsächlich übermittelt werden. Selbst wenn Fehler auftreten

oder der Consumer einer Nachricht zeitweise nicht erreichbar ist, muss die Übertragung erfolgen. Dazu werden die Nachrichten bei ihrem Eintreffen in der Queue oder dem Topic des JMS-Servers an einen persistenten Speicher übergeben. Sie werden erst aus diesem Speicher gelöscht, sobald sie nach u.U. mehreren Versuchen übermittelt werden konnten. Das Übermitteln von Nachrichten wird in zwei Transaktionen zerlegt. Eine Transaktion umschließt das Senden der Nachricht an den Kommunikationskanal. Darauf folgt zu einem beliebigen, späteren Zeitpunkt die zweite Transaktion, die das Senden der Nachricht vom Kommunikationskanal an den eigentlichen Empfänger umfasst.

Die Nachrichten, die zwischen Producer und Consumer übermittelt werden, sind serialisierbare Java-Klassen vom Typ `javax.jms.Message`. Eine Message umfasst zwei wichtige Komponenten:

1. **Headers und Properties:** Die Headers sind ein Satz von vorgegebenen Attributen, mithilfe derer der JMS-Server die Nachrichten geeignet behandeln kann. Durch die Properties können beliebige weitere Informationen übermittelt werden, die der anwendungsspezifischen Verarbeitung dienen.

2. **Nutzlast:** Die Nutzlast (payload) oder der Rumpf der Nachricht enthält die eigentliche Information, die übermittelt werden soll. Die abgeleiteten Typen von `Message` unterscheiden sich nur durch ihre Nutzlast. `Message` selbst enthält keinen Rumpf und wird benutzt, um den Eintritt eines Ereignisses (Event) zu signalisieren.

Auf Seite des Empfängers (Consumer) kann ein *Filter* eingesetzt werden, sodass nur eine definierte Teilmenge aller empfangenen Nachrichten vom Empfänger weiterverarbeitet wird. In der JMS-Terminologie werden diese Filter als *Selector* bezeichnet. Jeder Consumer kann seinen eigenen Selector definieren. Er ist ein `String`, der beim Erzeugen des Consumer übergeben wird. Nach einer Teilmenge der SQL-92 Syntax für Where-Clauses definiert der `String` eine Abfrage der *Header* und *Properties* einer eintreffenden Message. Erfüllt die Message die Abfragekriterien, so wird sie von dem Consumer weiterverarbeitet [47, S. 43ff]. Ansonsten wird sie ignoriert. Ausführliche Beispiele und eine Einführung in die Syntax der Selectoren enthält [47, S. 204-209].

4.1.2 Message-Driven Beans

Durch die Integration des JMS in die J2EE-Plattform kann jedes ihrer Elemente zu einem Producer oder Consumer von Messages werden. Es ist beispielsweise möglich, mit einer stateless Session Bean Nachrichten an eine Queue oder ein Topic zu versenden. Das Empfangen von Nachrichten kann allerdings nur *synchron* geschehen, da Webkomponenten, Session Beans und Entity Beans durch das synchrone RMI-IIOP gesteuert werden. Das heißt so lange die Session Bean auf den Empfang einer Nachricht wartet, ist die restliche Funktionalität blockiert [47, S. 149]. Nur Elemente der J2EE wie die Client-Applikation und Message-Driven Beans sind in der Lage, Nachrichten *asynchron zu empfangen*.

Beim Einsatz von Message-Driven Beans muss zuerst das Übermittlungskonzept bestimmt werden: pub/sub oder p2p. Daraufhin wird auf einem JMS-Server entweder ein

Topic oder eine Queue eingerichtet. Der JMS-Server ist häufig Teil des EJB-Applikationsservers. Queues und Topics sind ab ihrer Erstellung über den Naming-Service verfügbar, sodass beliebige Producer an sie Nachrichten versenden können.

Eine Message-Driven Bean ist per Definition ein Consumer von Nachrichten, obwohl sie selbst auch Nachrichten versenden kann. Über einen Eintrag im Deployment Descriptor wird bestimmt, von welcher Queue oder welchem Topic sie Nachrichten bearbeitet. Die Message-Driven Bean muss zwei Interfaces implementieren:

- **`javax.ejb.MessageDrivenBean`**: Dieses Interface definiert die Callback-Methoden, mit denen der Container neue Beans erzeugt und auf die Verarbeitung von Messages vorbereitet.

```
public interface MessageDrivenBean{
    public void ejbCreate();
    public void ejbRemove();
    public void setMessageDrivenContext(
        MessageDrivenContext mdc
    );
}
```

Listing 4.1: Interface MessageDrivenBean

  Zuerst erzeugt der Container eine neue Instanz des Java-Objekts, das die Message-Driven Bean realisiert. Dieses Objekt wird über sein Umfeld in Kenntnis gesetzt, indem ihm per `setMessageDrivenContext(...)` ein Kontext zugewiesen wird. Der MessageDrivenContext basiert auf dem `EJBContext`, der auch an Session Beans und Entity Beans übergeben wird. Schließlich wird `ejbCreate()` aufgerufen und die neue Instanz dieser Message-Driven Bean einem Instanzenpool hinzugefügt. Ab jetzt kann sie Nachrichten entgegennehmen und verarbeiten.

- **`javax.jms.MessageListener`**: Dieses Interface definiert eine Methode:

```
public void onMessage(Message message);
```

Listing 4.2: Interface MessageListener

  Sie wird durch den Container benutzt, um der Bean eine Nachricht zur Bearbeitung zu übergeben. Dazu wählt der Container eine freie Instanz aus dem Pool aus. Nachdem sie die Nachricht bearbeitet hat, wird sie wieder dem Pool verfügbarer Instanzen hinzugefügt. Nach welchen Kriterien die Instanz aus dem Pool gewählt wird, ist in der Spezifikation nicht vorgegeben. Einige Applikationsserver realisieren durch die Auswahl der nächsten freien Instanz ein Load Balancing.

Message-Driven Beans sind damit den stateless Session Beans ähnlich. Beide Bean-Typen besitzen keine persistenten Attribute oder Zustände, die zwischen zwei Aufrufen aufrechterhalten werden. Einzelne Instanzen können völlig gleich eingesetzt werden und sind gegeneinander austauschbar. Dadurch eignen sie sich besonders für Instance-Pooling (vgl. [46, S. 48ff] und [47, S. 151ff]).

Bei der Erläuterung der Interfaces ist beschrieben worden, dass allein der *Container* Methoden auf der Message-Driven Bean aufruft. Dadurch wurde bereits angedeutet, dass kein Sender direkt Nachrichten an eine Message-Driven Bean übermittelt. Message-Driven Beans werden allein vom Container aufgerufen. Ein Producer hat nur die Möglichkeit, Nachrichten an die Queues oder Topics zu versenden, die durch den JMS-Server bereitgestellt werden. Damit erklärt sich, warum Message-Driven Beans im Gegensatz zu allen anderen Enterprise Beans *kein* Home oder Business Interface haben. Diese Interfaces sind nur notwendig für die Kommunikation zwischen Beans oder zwischen einer Bean und einem anderen Client.

Auch Message-Driven Beans können einen Selector benutzten, um Nachrichten zu filtern. Dieser kann aber nicht bei ihrer Erzeugung übergeben werden, da ihre Erzeugung vor dem Entwickler verborgen im Container stattfindet. Anstelle dessen kann der Selector im Deployment Descriptor angegeben werden.

Durch den Einsatz von Selectoren wird das folgende B2B-Szenario denkbar. Über eine Webkomponente können Bestellungen eingegeben werden. Die Bestellungen werden als `Message` an ein Topic versendet. Dadurch werden sie nach der »publish and subscribe«-Übermittlung an mehrere verschiedene Message-Driven Beans gleichzeitig verteilt. Die Message-Driven Beans unterscheiden sich durch die Funktionalität der `onMessage`-Methode und durch den Selector im Deployment Descriptor. In der Message werden als zusätzliche Properties der Bestellwert und die Kreditwürdigkeit des Auftraggebers vermerkt. Die Selectoren der verschiedenen Message-Driven Beans überprüfen diese Properties. Kleine Bestellungen von kreditwürdigen Auftraggebern werden durch eine Art von Message-Driven Beans direkt zur Bearbeitung an ein Lagersystem übermittelt. Ein anderer Selector lässt eine andere Message-Driven Bean aktiv werden, falls der Bestellwert sehr hoch ist oder der Auftraggeber als wenig kreditwürdig bekannt ist. Diese Bean übermittelt die Bestellung an einen Sachbearbeiter, der über die Abwicklung risikoreicher Bestellungen entscheiden kann. Schwellenwerte für die Kreditwürdigkeit und kritische Bestellwerte können im laufenden Betrieb angepasst werden, da die Selectoren in den Deployment Descriptoren angegeben werden. Durch die asynchrone Nachrichtenübermittlung und die Nutzung von Selectoren ist es im beschriebenen Beispiel möglich geworden, einfache Geschäftsabläufe nachzubilden und flexibel zur Laufzeit zu konfigurieren.

Message-Driven Beans bieten eine weitere wichtige Neuerung, die zur Steigerung von Performanz genutzt werden kann. Durch sie kann innerhalb des Servers auf parallele Verarbeitungsstränge verzweigt werden:

»In addition to providing the container infrastructure for message-driven beans, EJB 2.0 provides another important advantage: concurrent processing.[…] If several messages are delivered at the same time, the container can select a different bean instance to process each message; the messages can be processed concurrently« [47, S. 150f].

Die Möglichkeit, innerhalb des Servers von einem Verarbeitungsstrang auf mehrere Verarbeitungsstränge zu verzweigen, ist neu in der EJB 2.0-Spezifikation. Der übliche Weg über das Instanziieren von Threads ist dem Bean-Entwickler durch die Spezifikation ver-

boten. Der Grund dafür ist, dass durch das Instanziieren von Threads die Kontrolle des Containers über die Beans vermindert wird.

Das folgende Unterkapitel erläutert Schritt für Schritt die Implementierung von Parallelisierung mit Message-Driven Beans.

4.1.3 Ein Beispiel

Abschnitt 8.5 bietet ein ausführliches, praktisches Beispiel für die Nutzung von Message-Driven Beans zur Parallelisierung. Im Vorgriff auf dieses Beispiel wird an den folgenden Quelltextfragmenten gezeigt, wie eine Session Bean als Producer und eine Message-Driven Bean als Consumer zur Verzweigung auf parallele Verarbeitungsstränge genutzt werden können. Zuerst wird ein Ausschnitt aus dem Quelltext des Producers erläutert. Daran anschließend wird ein Ausschnitt aus dem Quelltext des Consumers dargestellt. Der vollständige Quelltext ist der beiliegenden CD zu entnehmen.

Die Aufgabe des Producers ist es, eine Verbindung zu einem Kommunikationskanal, d.h. zu einem Topic oder zu einer Queue, aufzubauen und an diesen Kommunikationskanal Nachrichten zu übermitteln. Zur Parallelisierung eignet sich nur eine Queue, da jede Nachricht genau einen »Arbeitsauftrag« enthält. Es werden also Nachrichten an identische Bean-Instanzen übermittelt, die daraufhin parallel arbeiten können.

Der Aufbau der Verbindung zum JMS-Server erfolgt im `ejbCreate` der Session Bean `BatchrahmenBean.java`, die als Producer fungiert. Dort wird eine Methode `private void openQueueSession()` aufgerufen. Diese Methode führt die folgenden Zeilen aus:

```
//1.
QueueConnectionFactory cf =
   (QueueConnectionFactory) (new InitialContext()).lookup(
       "de.adesso.mdb.vertrag.BatchConnectionFactory"
    );
//2.
QueueConnection queueConnection
   = cf.createQueueConnection();
//3.
QueueSession qSession = queueConnection.createQueueSession(
   true, Session.AUTO_ACKNOWLEDGE
);
//4.
queueConnection.start();
```

Listing 4.3: Aufbauen einer Verbindung zum JMS-Server

Die Nummer einer Quelltextzeile ist als Kommentarzeile vorangestellt; die einzelnen Zeilen bewirken das Folgende:

1. Über den Naming-Service wird eine Referenz auf eine `QueueConnectionFactory` ermittelt. Dieses Interface ist eine Abstraktion für eine Verbindung mit einem JMS-Server. Durch sie es möglich, die Dienste des JMS-Servers anzusprechen.

2. Die Referenz auf die `QueueConnectionFactory` wird benutzt, um eine Verbindung (`QueueConnection`) zu erstellen, über die Nachrichten übermittelt werden können. Eine solche Verbindung kann zum Senden und Empfangen benutzt werden. Für das vorliegende Beispiel wird sie nur zum Versenden genutzt.

3. Zu der Verbindung wird eine `QueueSession` instanziiert. Durch sie kann der Bean-Entwickler `Messages`, `QueueSender` und `QueueReceiver` erzeugen. `QueueReceiver` sind Subtypen von Consumern. Sie sind für das vorliegende Beispiel uninteressant, da als Consumer Message-Driven Beans benutzt werden. Der erste Parameter bei der Erzeugung der `QueueSession` gibt an, ob die Session transaktional ist. Der zweite Parameter gibt an, wie der Versender einer Message in Kenntnis gesetzt wird, ob die Nachricht korrekt übermittelt wurde. Das *Message Acknowledgement* ist zentraler Teil des Guaranteed Delivery (siehe Abschnitt 4.1.1). Erst nachdem der Sender durch das Acknowledge über die korrekte Ankunft der Nachricht im JMS-Server in Kenntnis gesetzt ist, gilt die Transaktion, innerhalb derer die Nachricht übermittelt wurde, als erfolgreich beendet.

4. Durch den Aufruf von `start()` auf der Verbindung wird der Übertragungskanal aktiviert. Durch `stop()` wird die Verbindung unterbrochen.

Durch die beschriebenen vier Schritte wird die Verbindung zu einem bestimmten JMS-Server vorbereitet. Nachdem diese Verbindung für die Übermittlung einer oder mehrerer Nachrichten genutzt worden ist, muss sie explizit beendet werden. Das geschieht durch den Aufruf von `queueConnection.close()` in der Callback-Methode der Session Bean `ejbRemove()`.

Noch ist keine Queue festgelegt, an die Nachrichten übermittelt werden. Der folgende Quelltextausschnitt entstammt einer fachlichen Methode innerhalb der SessionBean. Es wird dargestellt, wie Nachrichten an eine bestimmte Queue versendet werden.

```
//1.
// "getMessageQueueName" ist der getter zu: Sucher.MQ_NAME.
// Es ist ein Klassenattribut vom Typ String:
// "de.adesso.mdb.vertrag.SfrUmstufungQueue"
Queue batchQueue  =
    (Queue)(new InitialContext()).lookup(
        sucher.getMessageQueueName()
);
//2.
QueueSender batchQSender= qSession.createSender(batchQueue);
//...
//...
while(...){
    EJBObject vertrag = (EJBObject)alleTeilnehmer.next();
```

```
    Handle    vertragHandle = vertrag.getHandle();
    //3.
    ObjectMessage om =
        qSession.createObjectMessage(vertragHandle);
    //4.
    batchQSender.send( om );
}
```

Listing 4.4: Erzeugen und Versenden von Messages

Die nummerierten Schritte dienen dem folgenden Zweck:

1. Über den Naming-Service wird eine Referenz auf eine eindeutig bestimmbare `Queue` ermittelt.
2. Auf der `qSession`, die während des `ejbCreate(...)` erzeugt wurde, wird nun ein `QueueSender` erzeugt. Er sendet allein an die `Queue`, die zuvor per Naming-Service bestimmt wurde.
3. Mit der `qSession` werden Nachrichten erzeugt, die per `QueueSender` in die Queue eingereiht werden. Die Nachrichten sind vom Typ `ObjectMessage`, da sie als Nutzinformation ein `Handle` auf eine Bean übertragen müssen. Ein Handle ist ein langlebiges, serialisierbares Java-Objekt, das eine Referenz auf ein eindeutig bestimmbares Entity Bean repräsentiert. Im Gegensatz zu Remote Interfaces muss ein Handle auch nach einem Serverneustart noch Zugriff auf die dadurch referenzierte Bean ermöglichen. Beispielsweise kann ein Handle serialisiert gespeichert und später zum Erhalten eines Remote Interface benutzt werden.
4. Der `batchQSender` übermittelt schließlich die Nachricht an den JMS-Server. Sie enthält als Nutzinformation eine Referenz auf ein Entity Bean.

Wie in den vorangegangenen Kapiteln beschrieben, wird nirgendwo direkt auf eine Message-Driven Bean zugegriffen. Die Session Bean, die die Nachrichten erzeugt und versendet, interagiert alleine mit der Queue auf dem JMS-Server. Der JMS-Server ist zuständig, die Nachrichten an die Message-Driven Beans zu verteilen. Dadurch wird der Entwickler einer Anwendung von der Implementierung der Warteschlangenfunktionalität entbunden, die sich hinter diesem Konzept verbirgt.

Nachdem das Erzeugen und Versenden von Nachrichten dargestellt worden ist, kann die *Bearbeitung der Nachrichten durch den Consumer* beschrieben werden. Als Consumer dient eine Message-Driven Bean. Sie enthält neben den Callback-Methoden allein die Methode `onMessage(...)`. Nur über diese Methode kann fachliche Logik angesprochen werden.

Das folgende Quelltextfragment stellt die Methode `onMessage(...)` zum beschriebenen Beispiel dar. Die Callback-Methoden der Message-Driven Bean sind leer, da keine besondere Initialisierung oder Terminierung der Bean notwendig ist.

```
public void onMessage(javax.jms.Message msg) {
 try{
    //1.
    Handle vertragHandle =
        (Handle)((ObjectMessage)msg).getObject();
    //2.
    Vertrag vertrag =
        (Vertrag)PortableRemoteObject.narrow(
            vertragHandle.getEJBObject(),
            Class.forName("de.adesso.mdb.vertrag.Vertrag")
        );
    //3.
    vertrag.stufeUm(true);
  }catch (Exception e) {
    //...
  }
}
```

Listing 4.5: Empfangen und Verarbeiten von Messages mit Message-Driven Beans

1. Im ersten Schritt wird die Nutzinformation aus der übermittelten Nachricht extrahiert.
2. Sobald der `Handle` verfügbar ist, kann mit ihm ein Remote Interface angefordert werden. Per `PortableRemoteObject.narrow(...)` wird das Remote Interface auf den richtigen Typen (`de.adesso.mdb.vertrag.Vertrag`) geprägt.
3. Auf dem Remote Interface kann schließlich irgendeine beliebige Geschäftsmethode aufgerufen werden (hier: `stufeUm(boolean hochstufen)`).

Bisher ist beschrieben worden, wie Nachrichten erzeugt, versendet, empfangen und verarbeitet werden. Es wurde noch nicht deutlich, an welcher Stelle im Ablauf die Parallelisierung erfolgt. Sie erfolgt *deklarativ* im Deployment Descriptor zu der Message-Driven Bean. Dort wird angegeben, wie viele Instanzen der Pool von Message-Driven Beans enthält. Alle diese Instanzen können gleichzeitig arbeiten, indem ihnen durch Aufrufen von `onMessage(...)` eine Nachricht übergeben wird.

Das parallele Verzweigen und damit auch alle Aspekte des Load Balancing sind vor dem Bean-Entwickler verborgen. Der Container verzweigt auf die verschiedenen Threads, indem er jede Message-Driven Bean in einem eigenen Thread laufen lässt. Der Entwickler kann sich auf die Umsetzung der fachlichen Logik konzentrieren. Der Grad an Parallelität ist konfigurierbar, ohne den Quelltext neu übersetzen zu müssen. Durch mehrere Testläufe eines parallelisierten Prozesses kann eine Konfiguration gefunden werden, die optimal zur eingesetzten Hardware und zur umgesetzten Geschäftslogik passt. In Abschnitt 8.5 und 8.7 wird dies beispielhaft vorgeführt.

4.2 Local Interfaces

Zu Zeiten des EJB 1.x-Standards gab es zu einer Enterprise Bean zwei Interfaces:

- **Home Interface:** Dieses Interface definiert Methoden, die den Lebenszyklus der Bean betreffen (`create` und `remove`). Zusätzlich kann dort Funktionalität zum Auffinden von Instanzen deklariert werden.
- **Remote Interface:** Dieses Interface definierte Methoden der Geschäftslogik, die auf einer Bean aufgerufen werden konnten, sowie Zugriffsmethoden auf die Attribute einer Entity Bean. Der Zugriff auf ein Remote Interface ist immer der Zugriff auf eine bestimmbare Instanz bzw. auf eine durch Primärschlüssel bestimmbare Zeile einer Datenbanktabelle.

Nach diesem Ansatz waren Beans ortsunabhängig. Verteilungsaspekte mussten nicht explizit bei der Entwicklung berücksichtigt werden. Es wurde stets auf die Beans zugegriffen, als seien sie auf einem entfernten System, d.h. über RMI (Remote Method Invocation) mit Stubs und Skeletons. Dabei wurden immer alle Aufrufparameter serialisiert und deserialisiert — auch wenn die Beans in der gleichen JVM betrieben wurden.

Um die überflüssigen Serialisierungskosten bei Aufrufen zwischen Beans in der gleichen JVM zu vermeiden, sind Local Interfaces in die Spezifikation aufgenommen worden. Aufrufparameter können dadurch zwischen Beans in der gleichen JVM per Referenz übergeben werden. Eine Zwischenversion des 2.0-Standards sah zur Lösung dieses Problems Dependent Objects vor. Diese sind allerdings wieder aus dem Standard gestrichen worden. Abschnitt 8.4.1 enthält einen kurzen Exkurs über die Gründe der Streichung. Ein letzter Anklang von den Dependent Objects existiert noch in den Dependent Value Classes [18, S. 131].

Durch die Einführung der Local Interfaces [18, S. 51ff] hat sich eine etwas verwirrende Begrifflichkeit ergeben. Der 2.0-Standard unterscheidet zwischen zwei Gruppen von Interfaces:

- **Home Interfaces:** Sie deklarieren wie zuvor Methoden zum Erzeugen, Suchen und Löschen von Beans. Zusätzlich können sie auch so genannte Home Methods enthalten (siehe Abschnitt 4.5).
- **Component Interfaces:** Ein Component Interface ermöglicht den Zugriff auf Geschäftslogik und auf Attribute einer Enterprise JavaBean. Dieser Begriff ersetzt den Begriff des Remote Interface der alten Spezifikation.

Diese Interfaces können entweder als *Local Interface* oder als *Remote Interface* umgesetzt werden. Dabei wird in den Local Interfaces deklariert, was für einen Client einer Bean in der gleichen JVM aufrufbar ist. Die Remote Interfaces enthalten im Gegensatz dazu Funktionalität, die von entfernten Systemen abrufbar sein soll. Dadurch kann eine Bean bis zu vier verschiedene Interfaces haben: Local Component Interface, Local Home Interface, Remote Component Interface und Remote Home Interface. Es ist ausdrücklich zu betonen, dass eine Bean auch nur Local Interfaces oder nur Remote Interfaces haben kann.

Die in den Interfaces angebotene Funktionalität kann völlig unterschiedlich sein. Zugriffsmethoden für einzelne Attribute können aber auch gleichzeitig im Local und im Remote Component Interface angeboten werden. Die Implementierung der in den Interfaces angebotenen Methoden steht in allen Fällen in der Bean-Klasse.

Bei der Programmierung ist zu berücksichtigen, dass Remote Interfaces als Parameter nur Remote Interfaces benutzen dürfen. Analoges gilt für die Local Interfaces. Beispielsweise ist das Ergebnis eines `create`- oder `find`-Aufrufs auf einem Local Home Interface ein Local Component Interface. Container-Managed Relations (CMR) (siehe Abschnitt 4.3.1) können *nur* über Local Interfaces abgewickelt werden.

Durch die Unterscheidung in Local und Remote Interfaces wird die Ortsunabhängigkeit der Programmierung aufgelöst. Der Entwickler muss nun entscheiden, ob der Zugriff auf eine Bean innerhalb der gleichen JVM passiert oder eher ein Zugriff auf ein entferntes System ist. Das klingt nach überflüssigem Ballast bei der Modellierung und Entwicklung. Der EJB-Standard sollte gerade die Entwicklung von verteilten Systemen erleichtern, indem Verteilungsaspekte vor dem Entwickler verborgen bleiben und keinen Einfluss auf den Quelltext haben.

Dagegen spricht, dass es möglich sein muss, sowohl feingranularen als auch grobgranularen Zugriff auf eine Bean zu realisieren. Objekte, die in einem engen fachlichen Zusammenhang stehen oder im Lebenszyklus voneinander abhängen, rufen einander häufig auf und müssen viel übereinander wissen. Sie haben also einen feingranularen Zugriff aufeinander. Andere Objekte bieten als Verbund von Objekten komplexe fachliche Logik an, die seltener aufgerufen wird und in ihrer Ausführung unabhängig von weiterer Information ist. Auf solche fachlichen Dienste wird eher grobgranular zugegriffen. Die feingranularen Zugriffe mit großer Aufrufhäufigkeit werden durch die Verwendung von Local Interfaces wesentlich beschleunigt. Darin liegt die Erleichterung durch Local Interfaces.

Dieses Konzept kann zur Modellierung von Softwarekomponenten genutzt werden, wie sie in [30] oder [64] beschrieben sind. Der Zugriff von außen auf eine Komponente erfolgt dann grobgranular und per Remote Interface. Der starke innere Zusammenhalt der Komponente spiegelt sich in den feingranularen Aufrufen der Local Interfaces wieder.

Abschnitt 8.4 und 9.1 zeigen beispielhaft den Zuschnitt einer solchen Komponente und stellen die Auswirkungen auf die Anwendung dar.

Beispiele zu der Verwendung von Local Interfaces befinden sich auf der beiliegenden CD.

4.3 Container-Managed Persistence

Die Enterprise Java Bean-Spezifikation sieht zwei verschiedene Konzepte zur dauerhaften Speicherung (Persistenz) von Geschäftsobjekten vor: Bean Managed Persistence und Container-Managed Persistence. Der folgende Abschnitt stellt diese beiden Konzepte einander gegenüber und skizziert ihre Vor- und Nachteile.

In den folgenden Unterkapiteln werden die wichtigsten Neuerungen der Container-Managed Persistence nach dem 2.0-Standard vorgestellt. Es wird das schrittweise Vorgehen beschrieben, um eine Container-Managed Persistence Bean zu entwerfen. Der Abschnitt schließt mit einem Beispiel, dass anhand verschiedener Quelltextfragmente die Integration des Persistenzkonzepts in die Bean-Klassen darlegt.

Die beiden Persistenzkonzepte lassen sich in folgender Weise gegenüberstellen:

- **Bean Managed Persistence (BMP):** Dieses Konzept ist bereits seit der Version 1.0 der Spezifikation Teil des Standards. Es sieht vor, dass der Entwickler der Bean die Funktionalität implementiert, nach der die Attribute einer Entity Bean geladen, gespeichert, verändert und gelöscht werden. Diese Funktionlität ist auf durch den Standard vorgegebene Callback-Methoden der Beans verteilt (vgl. `ejbLoad()`, `ejbStore()`, `ejbCreate()`,...). Sie muss kodiert werden in der Anfragesprache der unter dem Applikationsserver liegenden Persistenzschicht. Das ist zumeist SQL, da zur Speicherung unternehmenskritischer Daten relationale Datenbanken sehr weit verbreitet sind.

  Bean Managed Persistence gibt dem Entwickler ein sehr hohes Maß an Freiheit bei der Gestaltung der Speicherungsfunktionalität. Das betrifft insbesondere das objektrelationale Mapping, d.h. die Frage, welches Bean-Attribut in welche Spalte einer Datenbanktabelle geschrieben wird. Falls der Typ der Persistenzschicht nicht geändert wird, beispielsweise von einer relationalen Datenbank zu einer objektorientierten, kann die entstandene Anwendung einfach portiert werden [7].

  Bean Managed Persistence ist andererseits fehleranfällig und zeitaufwendig in der Implementierung und Wartung. Stellt sich während der Entwicklung heraus, dass ein weiteres Attribut in einer Bean notwendig ist, so muss jede der Callback-Methoden angepasst und getestet werden. Hohe Aufwände fallen auch an, wenn das Namensschema der Persistenzschicht sich von der Entwicklungsumgebung zum produktiven System ändert. Jedes Vorkommen eines Bezeichners aus der Persistenzschicht, beispielsweise ein Spaltenname, muss gefunden und verändert werden. Alle Veränderungen sind zu testen.

- **Container-Managed Persistence (CMP):** Dieses Konzept gehört erst seit der Version 1.1 verbindlich zur Spezifikation. Um Entity Beans per CMP persistent zu speichern, werden alle Attribute im Deployment Descriptor *deklariert*. Ebenso ist zu deklarieren, wie die Bean-Attribute auf die Datenbankstrukturen abgebildet werden (objektrelationales Mapping). Zu dieser Deklaration erzeugt ein Generator, der mit dem Container des Applikationsservers ausgeliefert wird, die notwendigen Klassen, die die Speicherung umsetzen.

  Bis zur Version 2.0 der Spezifikation waren CMP Entity Bean schlecht zu portieren. Einerseits, da jeder Applikationsserver andere Containerschnittstellen besitzt, andererseits, da die vorangegangenen Versionen viele Detailfragen offen ließen. Das erschwerte das Portieren von Anwendungen auf andere Applikationsserver, selbst wenn sämtlicher Quelltext vorlag. Details zum Thema Portierung von EJB 1.0- und EJB1.1-Anwendungen sowie zwei beispielhafte Portierungen enthält [7].

Die Spezifikation 2.0 überlässt es weiterhin jedem Containerhersteller, wie er seine Schnittstellen für die Persistenzdienste gestaltet. Aber die Bean-Klassen und die Deklarationen für die Persistenz sind soweit detailliert, dass sich der Quelltext von CMP Entity Beans einfach portieren lässt. Durch die nachteilige Abhängigkeit von einem Generatorwerkzeug verschaffen sich CMP Entity Beans wichtige andere Vorteile:

- **Unabhängigkeit von der Persistenzschicht:** CMP Entity Beans sind unabhängig von der darunter liegenden Persistenzschicht und können durch Neugenerierung in jedem System gespeichert werden, für das der Generator ausgelegt ist.
- **Verkürzte Entwicklungszeiten:** Die Entwicklungszeiten sind kürzer als bei BMP, da die Deklarationen für die Persistenz sich direkt aus Modellierungswerkzeugen erstellen lassen. Das Casetool TogetherJ 5.5 von Togethersoft verfolgt beispielsweise einen solchen Ansatz. Es kann Bean-Klassen und Deployment Descriptoren aus UML-Klassendiagrammen generieren. Auf diesen Generaten lässt sich dann der Generator des Applikationsservers starten, sodass nach wenigen Schritten aus einem Klassendiagramm eine Anwendung entsteht, die sich im Applikationsserver installieren lässt.
- **Verkürzte Testzeiten:** Da die Persistenzfunktionalität generiert wird, muss sie nicht so intensiv wie selbst geschriebener Quelltext getestet werden. Die Tests können sich auf die implementierte Geschäftslogik der Anwendung konzentrieren.
- **Verbesserte Wartbarkeit:** Wird eine Veränderung am Namensschema vorgenommen oder ein weiteres Attribut in eine Bean eingepflegt, so wird diese Änderung konsistent durch den CMP Generator berücksichtigt. Die Veränderungen an der Deklaration für die Persistenz kann bei der Installation der Applikation in den Applikationsserver auf Richtigkeit überprüft werden.
- **Nutzung proprietärer Optimierungen:** Die meisten Applikationsserver bieten für EJB-Anwendungen Optimierungen an, die nicht durch den Standard gedeckt sind. Diese Optimierungen sind zumeist an die Nutzung von CMP gebunden.

Die Vorteile von Container-Managed Persistence legen eine breite Nutzung dieses Konzepts nahe. Da pro Bean die Art der Persistenz bestimmt werden kann, sollte erst einmal so viel wie möglich per CMP umgesetzt werden. Stehen erste Prototypen der Anwendung zur Verfügung, können an ihnen Lastmessungen durchgeführt werden. Ergibt sich dabei, dass Teile der Anwendung durch ein komplexeres objektrelationales Mapping beschleunigt werden können, so lassen sich diese Teile nachträglich per BMP umsetzen, ohne dass große Aufwände verloren sind.

Um die Wirksamkeit der beschriebenen Vorteile zu verstärken, ist das Konzept der Container-Managed Persistence im EJB 2.0-Standard um einige Fähigkeiten erweitert worden. Die folgenden Abschnitte beschreiben diese Erweiterungen.

4.3.1 Container-Managed Relations (CMR)

Relationen zwischen Beans können seit dem 2.0-Standard durch den Container verwaltet werden. Zuvor war es notwendig, dass der Entwickler der Beans mit Find-Methoden die Fremdschlüsselbeziehungen zwischen Datenbanktabellen abfragt, um zu einer Bean assoziierte Beans zu ermitteln. Seit der 2.0-Version des Standards werden die Relationen im Deployment Descriptor deklariert. Auf Ebene der Bean-Klassen beschreiben abstrakte `get`- und `set`-Methoden den Zugriff auf die assoziierten Beans. Die Implementierung für die abstrakten Methoden erzeugt der Generator, der mit dem Container ausgeliefert wird.

Die Spezifikation gibt folgende Eigenschaften für Container-Managed Relations vor [18, S. 131]:

- Nur CMP Entity Beans können durch Container-Managed Relations miteinander verbunden werden.
- Als Kardinalitäten sind alle bekannten Kardinalitäten erlaubt: 1-zu-1, 1-zu-m, m-zu-n.
- Relationen können in beide Richtungen oder nur in eine Richtung traversiert werden.
- Die `get`- und `set`-Methoden der CMR nutzen als Parameter das Local Interface der Bean (siehe Abschnitt 4.2) oder `java.util.Set` bzw. `Collection`. An dieser Stelle unterscheidet sich die Final Version der Spezifikation von dem Proposed Final Draft 2 der Spezifikation. Dort wurde für Container-Managed Relations noch das *Remote Interface* benutzt. Dieser Unterschied betrifft Anwendungen, die auf dem BEA WebLogic 6.1 implementiert worden sind und daher nicht dem endgültigen Stand der Spezifikation entsprechen.

Die zweite Eigenschaft fordert, dass auch m-zu-n-Relationen mit CMR umgesetzt werden können. Auf Ebene einer relationalen Datenbank wird dazu neben den beiden Tabellen, deren Elemente an der Relation teilnehmen, eine dritte *Verbindungstabelle* benötigt, die die Information enthält, welche Elemente der einen Tabelle welchen Elementen der anderen Tabelle assoziiert sind. Bei vorangegangenen Versionen der Spezifikation mussten die Elemente dieser Verbindungstabelle explizit als Entity Beans modelliert werden, um die Relation traversieren zu können. Seit der Version 2.0 erscheint die Verbindungstabelle allein in den Deployment Descriptoren.

Die letzte Eigenschaft, die von Container-Managed Relations gefordert wird, bedeutet vor allem, dass diese Relationen nicht durch das *Remote* Interface verfügbar gemacht werden dürfen. Anstelle dessen können z.B. Value Objects benutzt werden (vgl. [46, S. 257ff]), um die Elemente einer Assoziation nach außen sichtbar zu machen.

Bei allen Methodenaufrufen zur Verwaltung der Relation ist der Container für die Erhaltung der referenziellen Integrität zuständig. Die Spezifikation stellt ausführlich dar, wie sich die Referenzen beteiligter Objekte beim Aufruf der Methoden zur Verwaltung der CMR verändern [18, S. 137–154]. Dieses Buch verzichtet daher auf eine Wiederholung dieser Details.

Die Nutzung von CMR ist mittelbar für die Performanz relevant, da wesentliche Funktionalität der EJB QL von der Nutzung der CMR abhängen. Abschnitt 4.4 wird darauf näher eingehen.

4.3.2 Cascade Delete

Bei der Modellierung von Objektstrukturen gibt es eine besondere Relation zwischen verschiedenen Objekten: die Komposition. Sie umfasst zwei Aspekte [23]:

- **Teil-von-Beziehung:** Die eine Seite der Relation hat eine Kardinalität von 1. Die andere Seite der Relation kann mehrere Instanzen erfassen. Diese Instanzen *sind ein Teil von* der einzelnen Instanz. Beispielsweise können mehrere Fahrzeuge die Teile von einem Fuhrpark sein. Der Fuhrpark setzt sich zusammen aus den einzelnen Fahrzeugen und kann seine Funktion nur über seine einzelnen Teile anbieten.
- **Abhängigkeit im Lebenszyklus:** Die Teile des zusammengesetzten Objekts müssen bezüglich ihrer Erzeugung und Zerstörung von der Erzeugung und Zerstörung des zusammengesetzten Objekts abhängen. Damit ist der Fuhrpark *kein* Beispiel für eine Komposition. Wird der Fuhrpark aufgelöst, so sind die einzelnen Fahrzeuge weiterhin nutzbar. Hingegen kann eine Rechnung als Komposition aus einzelnen Rechnungsposten als Beispiel betrachtet werden. Wird die Rechnung gelöscht, verlieren auch die einzelnen Posten ihre Bedeutung und sollten ebenfalls gelöscht werden.

Kompositionen werden durch die EJB 2.0-Spezifikation technisch unterstützt. Die Teil-von-Beziehung lässt sich durch eine CMR abbilden. Die Lebenszyklusabhängigkeit wird umgesetzt, indem durch den Bean-Entwickler im `ejbPostCreate()` der komponierten Bean die Erzeugung der komponierenden Instanzen angewiesen wird bzw. die Relation auf die komponierenden Instanzen aufgebaut wird.

Um das *Löschen* der komponierten Instanz automatisch an ihre Einzelteile weiterzuleiten, ist im Deployment Descriptor der Block für Relationen (`ejb-relation`) um das Element `cascade-delete` erweitert worden. Ein Beispiel aus dem Deployment Descriptor ist in Abschnitt 4.3.4 enthalten. Cascade Delete kann nur eingesetzt werden, wenn das Ende der Relation, von dem das Löschen ausgeht, eine Kardinalität von 1 hat. Ein Löschen auf diesem Ende der Relation bewirkt, dass der Container automatisch `ejbRemove()` auf den Instanzen des anderen Relationsendes aufruft. Das kaskadierte Löschen bezieht sich allein auf Elemente der Relation, für die es deklariert wurde. Soll das Löschen weiter kaskadiert werden, muss es auch für die anschließenden Relationen deklariert werden. Wird eine Bean derart gelöscht, so werden alle Relationen aktualisiert, in der sie zuvor vorhanden war [18, S. 133].

Durch Cascade Delete wird die Verwaltung von Relationen vereinfacht. Lebenszyklusabhängigkeiten zwischen Instanzen werden allein deklarativ festgehalten. Quelltext zur Aktualisierung von Relationen, die durch das Löschen einer Instanz betroffen sind, wird überflüssig.

4.3.3 Dependent Value Classes

Dependent Value Classes sind instanziierbare, serialisierbare Java-Klassen [18, S. 131]. Sie können per CMP innerhalb einer Bean gespeichert werden, indem sie serialisiert in der Datenbank abgelegt werden. Dazu wird für diese Klasse ein zusätzliches CMP-Feld in der Bean mit abstrakten get- und set-Methoden angegeben. Im Deployment Descriptor wird dieses Feld auf eine Spalte einer Tabelle abgebildet, die Daten im Binärformat aufnehmen kann. Die interne Struktur einer Dependent Value Class wird *nicht* im Deployment Descriptor angegeben.

Der Aufruf der get-Methode zu der Dependent Value Class liefert stets eine *Kopie* dieser Klasse. Änderungen an der Dependent Value Class erfordern immer, dass der Inhalt der Datenbank mit der set-Methode zu der Dependent Value Class explizit ersetzt wird. Ansonsten sind die Änderungen verloren.

Durch die Speicherung im serialisierten Zustand ergibt sich der größte Nachteil, den Dependent Value Classes haben. Sie können nicht bei Suchfunktionen benutzt werden. *Beispielsweise* kann zu einer Entity Bean `PersonBean` die Adresse mit den Attributen Postleitzahl, Ort und Straße als Dependent Value Class gespeichert werden. Es ist dann weder per SQL noch per EQL möglich, eine Menge von Personen zu ermitteln, deren Postleitzahl mit 5 beginnt. Um diese Menge zusammenzustellen, muss jede `PersonBean` geladen werden, um im nächsten Schritt explizit auf ihre Dependent Value Class `Adresse` zuzugreifen. Es ist jedoch sehr inperformant, Ergebnismengen zu einer Suchanfrage durch Iterieren über Beans zu erstellen (siehe Abschnitt 8.1). Vorteil der Dependent Value Classes ist die einfache Möglichkeit, sie persistent zu speichern.

4.3.4 Ein Beispiel

Das folgende Beispiel zeigt anhand von Quelltextfragmenten, wie eine Entity Bean per CMP gespeichert wird. Eine der Anfangsschwierigkeiten bei der Entwicklung von Enterprise Java Beans besteht darin, dass Änderungen am Klassendiagramm konsistente Änderungen an *mehreren* Dateien erfordern. Ein Werkzeug, das die Interfaces, den Deployment Descriptor und die Bean-Klasse automatisch synchronisiert, ist dringend zu empfehlen. Ziel des folgenden Abschnitts ist es, auf die Stellen hinzuweisen, die synchronisiert werden müssen. Zum Zeitpunkt der Arbeit an diesem Buch lässt die Integration des EJB 2.0-Standards in die bekannten Entwicklungswerkzeuge noch viele Wünsche offen. Bei einer guten Werkzeugunterstützung ist es denkbar, dass viele der dargestellten Schritte automatisch durchgeführt werden.

Als Beispiel wird ein Ausschnitt aus dem Analysemodell gewählt, das dieses Buch begleitet (siehe Abschnitt 6.1). Abbildung 4.1 stellt den Ausschnitt dar.

Der Ausschnitt zeigt den folgenden Zusammenhang: »Eine SFKlasse wird genutzt von mehreren Verträgen. Jeder Vertrag wird durch höchstens eine SFKlasse modifiziert. Zu einer SFKlasse kann eine weitere SFKlasse als Rückstufungsklasse existieren.« Die fachliche Bedeutung der dargestellten Klassen wird in Abschnitt 6.1 beschrieben. Für das folgende Beispiel ist allein wichtig, wie die Attribute `id`, `beitragsFaktor` und

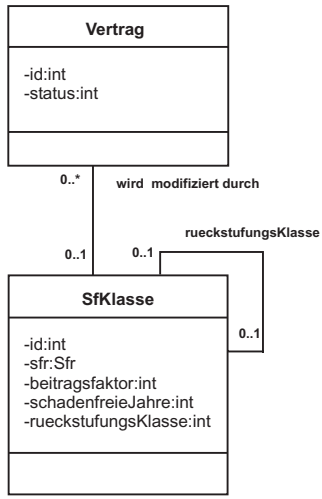


Abbildung 4.1: Klassendiagramm:Vertrag und SFKlasse

`schadenfreieJahre` in einer Bean gespeichert werden und wie die Relation zwischen `Vertrag` und `SFKlasse` realisiert wird. Die Attribute des Vertrags werden in dem Beispiel nicht berücksichtigt. Klassennamen, Attribute und Relationen sind das Ergebnis einer objektorientierten Analyse mit anschließendem Design.

Stehen als Ergebnis von Analyse und Design die fachlichen und technischen Attribute fest, kann mit dem Entwurf des Datenbankschemas begonnen werden. Das Klassendiagramm kann dazu wie ein Entity-Relationship-Diagramm gedeutet werden. Das heißt es wird angenommen, dass jede Klasse eine Tabelle repräsentiert und jedes Attribut eine Spalte in dieser Tabelle. Im nächsten Schritt muss zu jedem Typ eines Attributs im Klassendiagramm ein passender Datentyp für die Spalte gefunden werden.

Das Attribut `SFKlasse.id` ist technisch notwendig als Primärschlüssel für die Tabelle `SFKLASSE` in der Datenbank. Relationen zwischen Klassen werden auf Ebene der Datenbank als Fremdschlüsselbeziehungen ausgedrück. Für dieses Beispiel muss also der Primärschlüssel von `SFKlasse` als zusätzliche Spalte in der Tabelle von `VERTRAG` gespeichert werden. Um alle Instanzen von `Vertrag`, die zu einer gewissen `SFKlasse` gehören, finden zu können, wird diese zusätzliche Spalte analysiert.

Insgesamt ergibt sich das in Abbildung 4.2 dargestellte Schema. Der Fremdschlüssel für die Relation zwischen `Vertrag` und `SFKlasse` befindet sich in der Spalte `VERTRAG.VTG_SFK_ID`. Die selbstbezügliche Relation auf `SFKlasse` wird durch die Spalte `SFKLASSE.SFK_SFK_ID` realisiert. Die Spaltendefinitionen für Vertrag sind nur angedeutet. Eine vollständige Definition befindet sich auf der CD zu diesem Buch.

Auffällig ist vor allem, dass die Namensgebung der Spalten von den Namen der Attribute der Klassen abweicht. Indem die dreibuchstabige Abkürzung des Tabellennamens vor jeden Spaltennamen gestellt wird, erhält man globaleindeutige Spaltennamen. Das erleichert die Programmierung und Lesbarkeit von SQL-Anweisungen. Da die Namens-

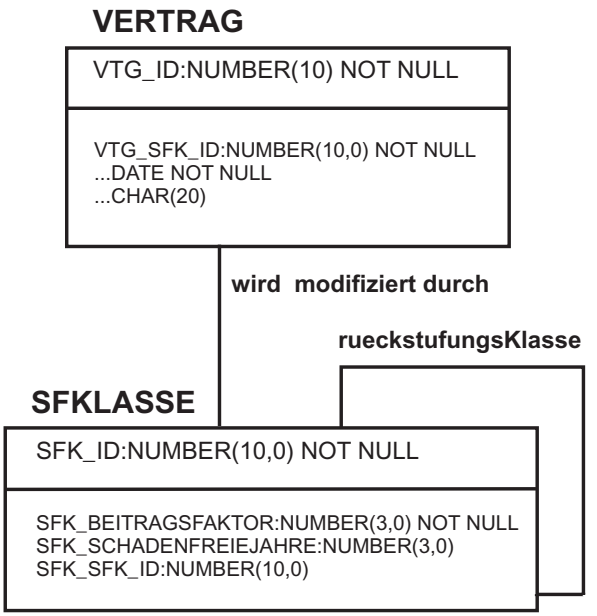


Abbildung 4.2: Datenbankschema:Vertrag und SFKlasse

gebungen auf Ebene der Beans und auf Ebene der Datenbank voneinander völlig unabhängig sind, ziehen globaleindeutige Spaltenbezeichner kein Problem nach sich. Die Gestaltung der Datenbanktabellen sei damit abgeschlossen. Enterprise JavaBeans erfordern keine besonderen Maßnahmen bei dem Entwurf von Datenbanktabellen.

Auf Ebene der Beans wird zunächst beschrieben, wie die Attribute von `SFKlasse` in `SFKlasseBean` integriert werden. Danach wird auf die Interfaces der Bean eingegangen.

Das folgende Listing stellt einen Ausschnitt aus dem Quelltext von `SFKlasseBean` dar. Die Bean-Klasse selbst ist eine abstrakte Klasse: `abstract public class SfKlasseBean`. Sie ist also nicht instanziierbar. Der Container wird eine Klasse generieren, die von ihr erbt, und aufgerufen wird, sobald auf das Remote Interface zu `SFKlasse` zugeriffen wird.

```
package de.adesso.finaleVariante.vertrag;
//import [...]
abstract public class SfKlasseBean implements EntityBean {

   abstract public double getBeitragsFaktor();
   abstract public void setBeitragsFaktor(
       double beitragsFaktor
   );

   abstract public int getSchadenfreieJahre();
   abstract public void setSchadenfreieJahre(
       int schadenfreieJahre
```

```
   );

   abstract public Integer getId();
   abstract public void setId(Integer id);
   //[...]
}
```

Listing 4.6: Getter und Setter zu Attributen in der Bean-Klasse

Zu jedem Attribut muss in der abstrakten Bean-Klasse eine *abstrakte* `get`- und `set`-Methode deklariert werden. Beispielsweise gehören diese Methoden zu dem Attribut `SFKlasse.beitragsFaktor`:

```
abstract public double getBeitragsFaktor();
abstract public void setBeitragsFaktor(
    double beitragsFaktor
);
```

Ein Attribut `private double beitragFaktor` wird in der Bean-Klasse *nicht* deklariert. Darin unterscheidet sich der 2.0-Standard von den vorangegangen Versionen. In ihnen muss zu jedem Attribut des Modells auch ein technisches Attribut als Deklaration in Java in der Bean vorhanden sein. Die `get`- und `set`-Methoden zu diesen deklarierten Attributen waren nicht abstrakt.

Die abstrakten `get`- und `set`-Methoden können innerhalb der Bean aufgerufen werden, um den Wert der Attribute zu lesen oder zu ändern. Das folgende Listing zeigt eine `ejbCreate`-Methode, in der diese Methoden genutzt werden. Zu einer Bean können mehrere `ejbCreate`-Methoden existieren, die zur Unterscheidung einen Postfix tragen dürfen (hier: NewBatchInstance). Zu jedem `ejbCreate` *muss* ein `ejbPostCreate` mit gleichem Postfix und gleicher Parameterliste existieren. Im `ejbCreate` müssen per `set[AttributName](...)` alle Attribute belegt werden, die als »not null« in der Datenbank definiert wurden. Als RückgabeTyp des `ejbCreate` dient der PrimaryKey. Innerhalb des `ejbCreate` *muss* `return null;` stehen, da der Container den PrimaryKey erzeugt und zurückliefert. Das `ejbPostCreate` dient der Initialisierung, die über die Belegung der Attribute hinausgeht. Im dargestellten Beispiel wird die selbstbezügliche Relation namens »rueckStufungsKlasse« initialisiert.

```
package de.adesso.finaleVariante.vertrag;
//import [...]
abstract public class SfKlasseBean implements EntityBean {
    //Attribute get und set[...]
    //Relationen get und set[...]
    // Callback–Methoden [...]

    public java.lang.Integer ejbCreateNewBatchInstance(
        double beitragsFaktor,
        int schadenfreiJahre,
```

```
        SfKlasse rueckStufungsKlasse
    )
        throws  CreateException, EJBException,
                RemoteException, SQLException,
                NamingException
    {
      InitialContext ic = new InitialContext();
      //setting cmp fields
      int id = DBHelper.getNextId(
                "sequenceForBean.SfKlasseBean",ic
      );
      setId( new Integer( id ) );
      setBeitragsFaktor(beitragsFaktor);
      setSchadenfreieJahre(schadenfreiJahre);

      return null;
    }

    public void ejbPostCreateNewBatchInstance(
        double beitragsFaktor,
        int schadenfreiJahre,
        SfKlasse rueckStufungsKlasse
    )
        throws  CreateException, EJBException,
                RemoteException, SQLException,
                NamingException
    {
        if(rueckStufungsKlasse != null) {
            setRueckStufungsKlasse(rueckStufungsKlasse);
        }
    }
}
```

Listing 4.7: CMP-Attribute in der Bean-Klasse

Sind innerhalb der Bean-Klasse zu allen Attributen `get`- und `set`-Methoden erstellt, wird über die Sichtbarkeit der Methoden entschieden. Das scheint paradox, da alle Methoden »public« sind. Da aber niemals mit der Bean-Klasse selbst interagiert wird, sondern der Zugriff nur über die Interfaces und die vermittelt durch den Container erfolgt, erklärt sich das Paradoxon. Nur wenn die Methoden auch im Interface deklariert werden, sind sie für einen Client der Bean sichtbar.

Das folgende Listing zeigt das Remote (Component) Interface `SFKlasse.java`. Dort ist zu sehen, dass alle dort deklarierten Methoden eine `RemoteException` werfen müssen. Weiterhin ist erkennbar, dass nicht alle Methoden der Attribute deklariert sind: `setId(int id)` fehlt und ist daher nicht außerhalb der Bean-Klasse aufrufbar.

```
package de.adesso.finaleVariante.vertrag;
//import ...
public interface SfKlasse extends EJBObject {

  double getBeitragsFaktor()throws java.rmi.RemoteException;
  void setBeitragsFaktor(double beitragsFaktor)
      throws java.rmi.RemoteException;

  int getSchadenfreieJahre()throws java.rmi.RemoteException;
  void setSchadenfreieJahre(int schadenfreieJahre)
      throws java.rmi.RemoteException;

  Integer getId()throws java.rmi.RemoteException;
}
```

Listing 4.8: CMP-Attribute im Component Interface

Seit der Version 2.0 der Spezifikation gibt es zu dem Remote (Component) Interface auch ein Local (Component) Interface. Mit dem Begriff *Component Interface* wird in der neuen Spezifikation das umschrieben, was in den alten Versionen der Spezifikation das Remote Interface gewesen ist: eine Schnittstelle, durch die die Geschäftslogik der Bean angesprochen werden kann. Ein Local Component Interface stellt einem Client, der in der gleichen JVM wie die Bean läuft, Methoden zur Verfügung. Durch das Remote Component Interface wird Funktionalität für *entfernte* Clients angeboten. In beiden Interfaces müssen nicht notwendig die gleichen Methoden zu finden sein (vgl. Abschnitt 4.2). Methoden können in keinem, einem oder in beiden Component Interfaces vorkommen.

Neben den Java-Dateien für Interfaces und die Bean-Implementierung muss auch ein Abschnitt im Deployment Descriptor für jede Bean angelegt werden. Das folgende Listing zeigt den Ausschnitt für `SFKlasse` aus dem vom Standard definierten Deployment Descriptor **ejb-jar.xml**.

```
<entity>
  <ejb-name>SfKlasseBean</ejb-name>
  <home>de.adesso.finaleVariante.vertrag.SfKlasseHome</home>
  <remote>de.adesso.finaleVariante.vertrag.SfKlasse</remote>
  <local>
    de.adesso.finaleVariante.vertrag.SfKlasseLocal
  </local>
  <local-home>
    de.adesso.finaleVariante.vertrag.SfKlasseLocalHome
  </local-home>
  <ejb-class>
    de.adesso.finaleVariante.vertrag.SfKlasseBean
  </ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
```

```
  <!-- [...] weitere Elemente -->
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>SfKlasseBean</abstract-schema-name>
  <cmp-field>
    <description>...</description>
    <field-name>beitragsFaktor</field-name></cmp-field>
  <cmp-field>
    <description>...</description>
    <field-name>schadenfreieJahre</field-name></cmp-field>
  <cmp-field>
    <description>...</description>
    <field-name>id</field-name></cmp-field>
  <primkey-field>id</primkey-field>
  <!-- weitere Informationen -->
 </entity>
```

Listing 4.9: EJB-Standard Deployment Descriptor für CMP

Die einzelnen Elemente haben folgende Bedeutung:

- **Interfaces und implementierende Klasse:** Die Elemente `home`, `remote`, `local`, `local-home` und `ejb-class` geben die Bestandteile der Bean an. Dabei bezeichnet `ejb-class` die Bean-Klasse, die die Implememntierung aller in den Interfaces deklarierten Methoden enthält. Die anderen Elemente bezeichnen die Interfaces zu der Bean.
- **Persistenzdienst:** Das Element `persistence-type` gibt an, ob die Bean per CMP oder per BMP gespeichert wird.
- **Primärschlüssel:** Das Element `prim-key-class` definiert den Typ des Primärschlüssels für die Bean. Seit EJB 2.0 können das auch Klassen vom Typ `String` oder `Integer` sein. Es können als Primary Key nur Objekte und keine atomaren Datentypen gewählt werden. Werden selbst geschriebene Objekte als Primary Key benutzt, so müssen die Methoden `equals` und `hashcode` korrekt implementiert werden. Fehler in diesen Implementierungen können auch zu Fehlern in Find-Funktionen führen.
- **Version der CMP:** EJB 2.0-Container müssen Persistenz auch nach EJB 1.1 umsetzen können. Da die beiden Versionen der CMP sehr unterschiedlich sind, muss die Version, nach der eine Bean persistent gemacht wird, explizit angegeben werden.
- **Abstraktes Schema und Bezeichner:** Das Element `abstract-schema-name` ist ein Bezeichner für das statische Schema der Bean, das im Deployment Descriptor beschrieben ist. Dieser Bezeichner wird beispielsweise in der EJB QL (siehe Abschnitt 4.4) benutzt. Auf den `abstract-schema-name` folgend muss jedes CMP-Attribut mit einem Bezeichner deklariert werden. Auch diese Bezeichner werden von der EJB QL benutzt. Die Namen der `get`- und `set`-Methoden in den Java-Klassen ergeben sich, indem diesen Bezeichnern ein »get« oder »set« vorangestellt und der erste Buchstabe

groß geschrieben wird. Abweichungen von diesem Schema führen zu Fehlern beim Generieren der Klassen, die im Container ausgeführt werden.

- **Attribut für den Primärschlüssel:** Wird kein zusammengesetzter Primärschlüssel benutzt, ist das Attribut für den Primärschlüssel mit `primkey-field` zu deklarieren. Es muss einem der zuvor deklarierten `cmp-field`-Elemente entsprechen. Bei zusammengesetzten Schlüsseln entfällt dieses Element. Anstelle dessen muss die Java-Klasse, die den zusammengesetzten Schlüssel definiert, über *public*-Attribute verfügen, die als Namen die entsprechenden Bezeichner der `cmp-field`-Elemente benutzen.

Schließlich fehlt noch das Bindeglied zwischen den Attributen auf Bean-Ebene und Spalten der Tabelle: der container-spezifische Deployment Descriptor für CMP. Die Trennung in verschiedene Descriptor-Dateien wurde vorgenommen, um Container austauschbar zu machen. So kann beispielsweise der Container des Bea WebLogic 6.1 durch einen Container aus der TopLink-Produktfamilie (Hersteller: WebGain) ersetzt werden. Durch die Aufteilung der Deployment Descriptoren wird die Portierbarkeit vereinfacht. Es werden Standard-konforme Beschreibungselemente von proprietären, nicht portierbaren Elementen separiert. Dadurch könnte es in Zukunft möglich werden, aus einem Satz Java-Dateien und den Standard-konformen Deployment Descriptoren Beans auf verschiedene Applikationsserver zu portieren — allein durch Generierung mit den vom Containerhersteller mitgelieferten Werkzeugen. Eine systematische Arbeit über die Portierbarkeit von EJB 2.0-Anwendungen steht jedoch noch aus.

Das folgende Listing zeigt einen Ausschnitt aus dem CMP-Deployment Descriptor **weblogic-cmp-rdbms-jar.xml** für den Bea WebLogic 6.1. Er beschreibt das Mapping der Bean-Attribute von `SfKlasseBean` auf die Spalten der Tabelle `SFKLASSE`.

```
<weblogic-rdbms-bean>
  <ejb-name>SfKlasseBean</ejb-name>
  <data-source-name>
      Voldagno-dataSource-BatchPool
  </data-source-name>
  <table-name>BATCHUSER.SFKLASSE</table-name>
  <field-map>
    <cmp-field>beitragsFaktor</cmp-field>
    <dbms-column>sfk_beitragsFaktor</dbms-column>
  </field-map>
  <field-map>
    <cmp-field>schadenfreieJahre</cmp-field>
    <dbms-column>sfk_schadensfreieJahre</dbms-column>
  </field-map>
  <field-map>
    <cmp-field>id</cmp-field>
    <dbms-column>sfk_id</dbms-column></field-map>
</weblogic-rdbms-bean>
```

Listing 4.10: Container-spezifischer Deployment Descriptor für CMP

Die Elemente haben die folgene Bedeutung:

- **Bean-Name:** Das Element `ejb-name` liefert die Zuordnung zu einer Bean-Definition im Standard Deployment Descriptor.
- **Datenquelle zur Speicherung:** Durch `data-source-name` wird angegeben, über welche Datenquelle (`javax.sql.DataSource`) die Bean persistent gemacht wird. Eine Datenquelle ist eine Abstraktion für die Dienste einer Datenbank.
- **Tabelle:** Mit dem Element `table-name` wird angegeben, in welche Tabelle der Datenbank die Bean gespeichert wird.
- **Attribut-Spalten-Abbildung:** Alle CMP-Attribute, die im Standard Deployment Descriptor angegeben sind, müssen in eine Spalte der Datenbank abgebildet werden. Dazu dienen die Elemente `field-map`. In ihnen werden die Bezeichner benutzt, die mit `cmp-field` im Standard Deployment Descriptor deklariert wurden. Das darauf folgende Element muss ein existierender Spaltenbezeichner der angegebenen Datenbanktabelle sein (vgl. Abbildung 4.2). Beim Einsetzen der Beans in den Applikationsserver wird überprüft, ob die verwendeten Datenbankbezeichner existieren und der richtige Datentyp damit referenziert wird.

Bisher wurden nur Attribute per Container gespeichert. Der Rest des Abschnitts widmet sich daher der Speicherung von Beziehungen zwischen Beans per CMR. Wie bereits gezeigt, müssen die Beziehungen zwischen Beans durch Fremdschlüssel beim Datenbankentwurf berücksichtigt werden (vgl. Beschreibung zu Abbildung 4.2).

Danach werden die Relationen mit `get-` und `set`-Methoden in dem *Local* Component Interface deklariert. An dieser Stelle fließt die Kardinalität der Relation ein. Bei einer Kardinalität von 1 wird als Rückgabewert und Parameter der Methoden der Local-Component-Interface-Typ gewählt. Bei Kardinalitäten größer als 1 wird als Rückgabe oder Parameter eine Menge des Local-Component-Interface-Typs benutzt (`java.util.Collection` oder `java.util.Set`).

Das folgende Listing zeigt die `get-` und `set`-Methoden auf Seite der Entity Bean `SfKlasseBean`. Da eine `SfKlasse` mehrere `Vertraege` modifizieren kann (vgl. Abbildung 4.2), wurde `java.util.Collection` als Parametertyp gewählt. Die Methoden müssen als abstrakt deklariert werden. Die Implementierung zu ihnen erzeugt der Generator, der zu dem Container ausgeliefert wird.

```
package de.adesso.finaleVariante.vertrag;
//import [...]
abstract public class SfKlasseBean implements EntityBean {
   //[...]
   // Container-Managed Relations
    abstract public Collection getVertraege();
    abstract public void setVertraege(Collection c);
   //[...]
}
```

Listing 4.11: CMR in der Bean-Klasse

CMR dürfen nur durch das Local Component Interface zugänglich gemacht werden und können nur Elemente vom Typ dieses Interface enthalten. Der »Proposed final Draft 2« der Spezifikation definierte CMR noch dem Remote Component Interface. Nach der finalen Version der Spezifikation sähe das Local Component Interface wie im folgenden Listing aus. Leider implementierte zur Arbeit an diesem Buch noch kein Applikationsserver den finalen Standard.

```
package de.adesso.finaleVariante.vertrag;
//import ...
public interface SfKlasseLocal extends EJBLocalObject {
    //[...]
    public Collection getVertraege()
        throws RemoteException;
    public void setVertraege(Collection c)
        throws RemoteException;

}
```

Listing 4.12: CMR im Local Interface

Das andere Ende der Relation würde in `VertragBean.java` über die folgenden Methoden definiert:

```
abstract public void setSfKlasse(SfKlasseLocal sfKlasse);
abstract public SfKlasseLocal getSfKlasse();
```

Listing 4.13: CMR bei 1-Kardinalitäten

Die Definition im Local Component Interface erfolgt analog. Da die Kardinalität dieses Endes der Relation 1 ist, wird als Parametertyp das Local Component Interface `SfKlasseLocal` gewählt. Der Quelltext auf der beiliegenden CD ist für den Bea WebLogic 6.1 geschrieben. Soweit es Relationen betrifft, unterstützt dieser Server die Definition der CMR auf den Local Interfaces nicht. Die Beispielquelltexte benutzen für CMR Remote Interfaces und sind allein in dieser Hinsicht nicht konform mit dem finalen Stand der Spezifikation.

Neben der Deklaration von Methoden der Bean müssen für CMR auch Blöcke im Deployment Descriptor **ejb-jar.xml** angegeben werden. Die Deklaration einer Relation besteht aus zwei Teilen, so genannte Rollen. Jede Rolle beschreibt eine Leserichtung der Relation. Das folgende Listing zeigt den Abschnitt, der die Relation zwischen `SfKlasse` und `Vertrag` definiert. Diese Relation bekommt über das Element `ejb-relation-name` den Bezeichner »Vertrag-SfKlasse« zugewiesen. Darauf folgend werden die beiden Leserichtungen der Relation beschrieben. Die erste Rolle lässt sich etwa so lesen:

»Die Beziehung namens gehoertZuVertrag (`ejb-relationship-role-name`) hat auf der Ausgangsseite eine Kardinalität von 1 (`multiplicity`) und geht von SfKlasseBean (`relationship-role-source`) aus. Innerhalb der Ausgangsklasse wird die Relation

durch das Member-Attribut (`cmr-field`) namens vertraege (`cmr-field-name`) vom Typ `java.util.Collection` (`cmr-field-type`) repräsentiert.«

```
<ejb-relation>

  <ejb-relation-name>
    Vertrag-SfKlasse
  </ejb-relation-name>

  <ejb-relationship-role>
    <ejb-relationship-role-name>
      gehoertZuVertraegen
    </ejb-relationship-role-name>
    <multiplicity>one</multiplicity>
    <relationship-role-source>
      <ejb-name>SfKlasseBean</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name>vertraege</cmr-field-name>
      <cmr-field-type>
        java.util.Collection
      </cmr-field-type>
    </cmr-field>
  </ejb-relationship-role>

  <ejb-relationship-role>
    <ejb-relationship-role-name>
      hatSfKlasse
    </ejb-relationship-role-name>
    <!-- <cascade-delete/> -->
    <multiplicity>many</multiplicity>
    <cascade-delete/>
    <relationship-role-source>
      <ejb-name>VertragBean</ejb-name>
    </relationship-role-source>
    <cmr-field>
      <cmr-field-name>sfKlasse</cmr-field-name>
    </cmr-field>
    </ejb-relationship-role>

</ejb-relation>
```

Listing 4.14: CMR im Deployment Descriptor

Das andere Ende der Relation lässt sich ebenso beschreiben. Die benutzten Bezeichner für Member-Attribute (hier: `vertraege` und `sfKlasse`) müssen sich wie bei den CMP-

Feldern in den `get`- und `set`-Methoden widerspiegeln. Innerhalb der zweiten Rolle ist das Element für kaskadiertes Löschen angegeben: `cascade-delete`. Da dieses Löschen für das dargestellte Beispiel fachlich nicht sinnvoll ist, wurde es auskommentiert.

Das Beispiel ist damit abgeschlossen und hat die Umsetzung der CMP und CMR gezeigt. Durch das kaskadierte Löschen und die Verwaltung der referenziellen Integrität bei CMR wird es dem Entwickler erlaubt, sich stärker auf die Geschäftslogik zu konzentrieren.

Leider ist die Werkzeugunterstützung für diese Neuerungen noch dürftig, sodass die vielen Abhängigkeiten zwischen den Bezeichnern häufig für Fehler sorgen. Ändert sich beispielsweise ein Attributname, so muss diese Änderung schlimmstenfalls in sieben Dateien konsistent durchgeführt werden:

- im Local und Remote Component Interface, falls dort `get`- und `set`-Methoden verfügbar sein sollen,
- im Local und Remote Home Interface als Parameter für `create`-Methoden, vielleicht als Parameter für `find`-Methoden
- in der Bean-Klasse,
- im Standard Deployment Descriptor,
- im Container-spezifischen Deployment Descriptor für CMP.

4.4 EJB QL

Die Abkürzung EJB QL steht für Enterprise JavaBean Query Language. Sie ist kein weiterer SQL-Dialekt oder SQL-Ersatz. Die Spezifikation beschreibt sie als »query specification language« [18, S. 217–242]. Das bedeutet insbesondere, dass die EJB QL allein zum Auffinden von Daten benutzt werden kann. Ein Manipulieren, Löschen oder Einfügen ist per EJB QL nicht möglich. Der folgende Abschnitt beschreibt die Konzepte der EJB QL und erläutert ihre Verwendung für Find und Select Methods.

Bei Anwendungen, die nach dem 1.0- oder 1.1-Standard implementiert sind, werden Suchanfragen in SQL formuliert und in den Java-Code eingebettet. Dadurch wird die Unanbhängigkeit gegenüber der Persistenzschicht zerstört. Alternativ besitzen einige Applikationsserver Container-Methoden, die *alle* Instanzen aus der Datenbank als Bean liefern. Es kann dann durch Iterieren auf *Ebene von Java-Quelltext* eine Ergebnismenge gebildet werden. Das ist allerdings katastrophal für die Performanz. Die EJB QL erhöht die Austauschbarkeit der Persistenzschicht, indem sie als Anfragesprache auf dem im Deployment Descriptor definierten abstrakten Schema der Bean arbeitet. Durch sie bleibt der Applikationsserver von der Persistenzschicht unabhängig, ohne zu viel Performanz einzubüßen.

Suchfunktionalität wird durch die EJB QL deklarativ in eine Bean integriert. Auf Ebene des Java-Quelltextes wird im Home Interface bei Find-Methoden und in der Bean-Klasse bei Select-Methoden die Signatur der Methode (Parameter, Rückgabetyp, Exceptions)

angegeben. Diese Signatur wird im Deployment Descriptor wiederholt und erhält zusätzlich eine in EJB QL formulierte Anfrage.

Diese Anfrage ist an SQL92 angelehnt und hat die Form:

```
select_clause from_clause [where_clause]
```

Die Bestandteile haben die folgende Aufgabe:

- `select_clause`

  Dieser Teil gibt den *Typ* der Ergebnismenge von Objekten oder Einzelwerten an. Einzelwerte können nur bei so genannten Select-Methoden als Ergebnistyp benutzt werden und dann auch nur falls sie als CMP-Attribut Teil einer CMP-Bean sind [18, S. 232ff]. Neben den CMP-Attributen können alle Beans als Rückgabetyp genutzt werden, deren Struktur als abstraktes Schema im Deployment Descriptor bekannt ist. Über eine Select-Methode ist es also möglich, die Werte für ein einzelnes Attribut sämtlicher Instanzen aus der Datenbank zu ermitteln, *ohne* stets die ganze Bean laden zu müssen. Finder, die im Remote Home Interface definiert sind, haben als Ergebnistyp stets Remote Component Interfaces. Bei Local Home Interfaces verhält es sich analog.

- `from_clause`

  Die `from_clause` beschreibt die *Herkunft* der in der `where_clause` benutzten Variablen. Die Herkunft kann entweder, wie im Beispiel, ein abstraktes Schema aus dem Deployment Descriptor sein oder eine Menge von Elementen, die durch eine Container-Managed Relation definiert wird [18, S. 223].

- `where_clause`

  Dieser Teil der EJB QL-Anfrage ist optional und beschreibt eine Bedingung, der die ausgewählten Elemente genügen müssen. Es können verschiedene Operatoren benutzt werden. Die Eingabeparameter werden beginnend bei 1 durchnummeriert. Innerhalb der Bedingung referenziert ?1 beispielsweise den ersten Parameter, der in der Methodensignatur im Quelltext angegeben ist.

Als Bezeichner werden die Bezeichner der CMP- oder CMR-Felder aus dem Deployment Descriptor benutzt. Innerhalb der Anfrage kann auch über Relationen traversiert werden, um Attribute von assoziierten Beans anzufragen.

Die Eingabeparameter können nur innerhalb der `where_clause` benutzt werden. Die `where_clause` muss nicht alle angegebenen Parameter verwenden. Als Parameter können einfache Datentypen wie `String`, `int`, `double` usw. benutzt werden. Sollen in der Anfrage auch Vergleiche auf `null` möglich sein, muss für atomare Datentypen der Objektwrapper benutzt werden, z.B. `Integer` anstelle von `int`. Komplex strukturierte Objekte können nur verwendet werden, wenn ihr abstraktes Schema im Deployment Descriptor definiert ist [18, S. 128]. Das sind vor allem CMP Entity Beans.

Eine ärgerliche Einschränkung ergibt sich bei dem Umgang mit Datumsfeldern. Für Datumsvergleiche kann bisher nur der Typ `java.sql.Date` genutzt werden, der intern

die Zeit als `long` repräsentiert. Um beispielsweise zu ermitteln, ob ein Datum mehr als ein Jahr in der Vergangenheit liegt, muss über `java.util.Calendar` das um ein Jahr verminderte Datum als `long` erzeugt werden. Ebenso ärgerlich ist, dass keine Vererbungshierarchien bei der EJB QL berücksichtigt werden [18, S. 235].

Innerhab der `where_clause` können Literale verschiedenen Typs verwendet werden:

- **Boolesche Literale:** `TRUE` und `FALSE`,
- **Numerische Literale:** beispielsweise ganzzahlige Werte wie `42`, `23`, `-88` oder Dezimalwerte in wissenschaftlicher Notation `-57.9E2` bzw. mit Dezimalpunkt `+6.2`,
- **Literale von Zeichenketten:** Literale von Zeichenketten werden umschlossen von einfachen Anführungszeichen: `'MK'`, `'Herr'`. Die Zeichen sind, wie in Java üblich, in Unicode kodiert.

Folgende Operatoren und Ausdrücke sind innerhalb der `where_clause` erlaubt:

- **Pfadoperator:** Mit Hilfe des Pfadoperators lässt sich ein Navigationspfad zu einem Container-Managed Attribut oder einer Container-Managed Relation definieren. Die Angabe `vertrag.sfKlasse.beitragsFaktor` bestimmt beispielsweise einen Verweis auf das Attribut `beitragsFaktor`, indem von einem `Vertrag` zu seiner assoziierten `SfKlasse` und von dort zu deren Attribut `beitragsFaktor` navigiert wird.
- **Arithmetische Operatoren:**
  - Vorzeichenoperatoren +, –
  - Multiplikation und Division `*`, `/`
  - Addition und Subtraktion +, –
- **Vergleichsoperatoren:** =, >, >=, <, >=, <> (ungleich)
- **Logische Operatoren:** `NOT`, `AND`, `OR`
- **BETWEEN:** Dieser Ausdruck ist auf arithmetische Ausrücke anwendbar und prüft, ob ein Wert innerhalb eines Intervalls liegt: `sfKlasse.beitragsFaktor BETWEEN 75 AND 100`.
- **IN:** Dieser Operator kann nur auf Zeichenketten angewendet werden. Er prüft, ob eine Zeichenkette Element einer Aufzählung von Literalen ist. Beipielsweise kann ein Attribut des Vertrags ein Landeskennzeichen nach ISO 3166 sein. Dann liefert der folgende Vergleich `TRUE` für Verträge aus Deutschland, Japan und dem Libanon: `vertrag.landeskennzeichen IN ('DE','JP','LB')`
- **LIKE:** Eine Zeichenkette wird mit einem Muster verglichen. Genügt die Zeichenkette dem Muster, liefert der Vergleich `TRUE`.
- **NULL-Vergleich:** Durch diesen Operator kann ein Attribut einer Bean auf `NULL` verglichen werden: `vertrag.sfKlasse IS NOT NULL` prüft beispielsweise, ob einem `Vertrag` eine `SfKlasse` zugeordnet ist (vergleiche Abbildung 4.1).

- **EMPTY-Vergleich:** Durch diesen Ausdruck kann ermittelt werden, ob einer Bean weitere Beans assoziiert sind: `sfKlasse.vertraege IS EMPTY` ermittelt, ob es Instanzen von `Vertrag` gibt, die durch diese `SfKlasse` modifiziert werden (vergleiche Abbildung 4.1).
- **MEMBER-Anfrage:** Mit diesem Ausruck kann geprüft werden, ob eine Bean Element einer Menge von Beans ist: `vertrag MEMBER OF sfKlasse.vertraege` prüft beispielsweise, ob die Instanz `vertrag` zu den Instanzen von `Vertrag` gehört, die von der Instanz `sfKlasse` modifiziert werden (vergleiche Abbildung 4.1).
- **Funktionale Ausdrücke:**
  - Funktionen auf Zeichenketten: Die EJB QL kennt vier Funktionen auf Zeichenketten: `CONCAT(String, String)` zum Zusammenfügen von Zeichenketten, `SUBSTRING(String, start, length)` zum Ermitteln von Teilzeichenfolgen und `LOCATE(String, String[, start])` zum Ermitteln des ersten Vorkommens einer Teilzeichenfolge. Zum Ermitteln der Länge einer Zeichenkette dient `LENGTH(String)`.
  - Arithmetische Funktionen: Die Funktion `ABS(number)` ermittelt den absoluten Wert einer Zahl. Die Funktion `SQRT(double)` errechnet die Wurzel der übergebenen Zahl.

Eine vollständige, formale Definition der EJB-QL-Syntax kann im Backus-Naur-Format (BNF) in [18, S. 239f] angelesen werden.

Unter Verwendung des abstrakten Schemas und des im Deployment Descriptor festgehaltenen objektrelationen Mapping wird aus der EJB-QL-Anfrage eine Anfrage für die darunter liegende Persistenzschicht generiert. Das ist zumeist SQL; denkbar sind aber auch Anfragesprachen nicht-relationaler Datenbanken. Die EJB QL kann in zwei Arten von Methoden eingesetzt werden: *Find Methods* und *Select Methods*. Diese Methoden werden im Folgenden anhand von Beispielen beschrieben.

4.4.1 Beispiele für Find Methods

Find Methods werden im Home Interface zu einer Bean deklariert. Sie sind schon aus vorangegangenen Versionen der Spezifikation bekannt. Neu ist ihre Art der Umsetzung. Während zuvor jeder Applikationsserverhersteller seine eigene Art hatte, nach der ein Bean-Entwickler die Suchanfrage formulieren musste, sind die Suchanfragen mit EJB QL vom Werkzeug unabhängig. Sie sind sowohl bezüglich der Persistenzschicht als auch bezüglich des Applikationsservers portierbar. Zur Portierung notwendige Änderungen sind begrenzt auf den Teil des Deployment Descriptors, der die Bean-Attribute auf die Strukturen und Bezeichner der Persistenzschicht abbildet.

Bei der Implementierung von Find Methods muss sich der Entwickler zuerst entscheiden, welchen Typ das Ergebnis der Anfrage hat. Suchanfragen, die im Local Home Interface deklariert sind, haben als Rückgabewert ein Local Component Interface oder eine Menge von Local Component Interfaces. Bei Suchanfragen, die im Remote Home Interface deklariert sind, verhält es sich analog. Es ist die Aufgabe des Containers, zu der Definition

der Anfrage im Deployment Descriptor den richtigen Typ als Ergebnismenge zu instanziieren. Ebenso ist es Aufgabe des Containers, aus der in EJB QL formulierten Anfrage eine Anfrage für die benutzte Persistenzschicht zu bilden.

Um beispielsweise zu einem `RisikoBuendel` alle aktuell gültigen Instanzen von `ZuAbschlagStufe` zu finden, wird die folgende Signatur im Local Home Interface angegeben:

```
public Collection findeAktuelleZuElementarRisiko(
                    int id,
                    Date maxDate
                  )
    throws FinderException;
```

Listing 4.15: Find Method im Local Home Interface

Da mehrere Elemente in der Treffermenge erwartet werden, ist der Rückgabetyp der Anfrage eine `Collection`. Die Collection enthält Local Interfaces der Bean-Klasse `ZuAbschlagStufeBean`, da die Methode im Local Home Interface zu `ZuAbschlagStufeBean` deklariert ist. Auf diesen Typ muss gecastet werden, wenn über die Ergebnismenge iteriert wird. Als Parameter dient das Attribut von `ZuAbschlagStufe`, das den Fremdschlüssel auf `RisikoBuendel` repräsentiert, und ein Gültigkeitsdatum. Die `FinderException` ist durch die Spezifikation vorgegeben. Im Deployment Descriptor wird folgender Block in die Deklaration von `ZuAbschlagStufe` eingefügt:

```
<query>
  <query-method>
    <method-name>
      findeAktuelleZuRisikoBuendel
    </method-name>
    <method-params>
      <method-param>int</method-param>
      <method-param>java.sql.Date</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <![CDATA[
    SELECT OBJECT(v) FROM ZuAbschlagStufeBean v
          WHERE v.risikoBuendelFachId = ?1
                AND v.hGueltigBis = ?2
    ]]>
  </ejb-ql>
</query>
```

Listing 4.16: EJB QL im Deployment Descriptor

Der von `query-method` umschlossene Bereich wiederholt die Signatur aus dem Interface. Schreibfehler im Methodennamen oder Fehler in der Auflistung der Parameter führen

dazu, dass zu der im Interface deklarierten Methode kein Abschnitt im Deployment Descriptor gefunden wird.

Das Element `ejb-ql` begrenzt die abstrakte Definition der Anfrage. Die `select_clause` definiert, dass ganze Objekte gesucht werden, die der Container in Component Interfaces zu wandeln hat. Die über Variable `v` bezeichneten Objekte stammen laut der `from_clause` aus dem abstrakten Schema mit Namen `ZuAbschlagStufeBean`. Alle gefundenen Objekte müssen den in der `where_clause` definierten Bedingungen genügen:

- das Attribut `risikoBuendelFachId` muss dem ersten angegebenen Parameter gleichen *und*
- das Attribut `hGueltigBis` muss dem zweiten angegebenen Parameter gleichen.

Sollte diese Anfrage nicht nur über das Local Home Interface angeboten werden, sondern auch über das Remote Home Interface, so ist die Signatur aus dem Local Home Interface in das Remote Home Interface zu kopieren. Wie alle Methoden im Remote Interface muss ihre Deklaration auch eine `RemoteException` enthalten:

```
public Collection findeAktuelleZuElementarRisiko(
                        int id,
                        Date maxDate
                    )
    throws FinderException, RemoteException;
```

Listing 4.17: Find Method im Remote Home Interface

Der Abschnitt im Deployment Descriptor dient damit *beiden* Methoden. Der Container hat die Aufgabe, im einen Fall die Collection mit Local Interfaces zu füllen und im anderen Fall mit Remote Interfaces. Ist die Ergebnismenge leer, wird eine leere Collection geliefert.

Duplikate in der Treffermenge einer Anfrage können auf zwei Arten vermieden werden [18, S. 184]:

- Anstelle einer `Collection` wird im Home Interface ein `Set` als Rückgabetyp definiert.
- Die `select_clause` wird um das optionale Schlüsselwort `DISTINCT` erweitert: `SELECT DISTINCT Object(v) FROM...`. Der Rückgabetyp darf dann `Collection` oder `Set` sein.

Ein Beispiel für eine EJB-QL-Anfrage, die Container-Managed Relations innerhalb der `where_clause` traversiert, wird in Abschnitt 8.2.2 angegeben und erläutert.

4.4.2 Ein Beispiel für Select Methods

Select Methods sind neu in der Spezifikation. Sie werden innerhalb der Bean deklariert und sind nicht außerhalb von ihr verfügbar. Rückgabewerte können Local oder Remote Interfaces sein. Außerdem können auch Typen einzelner CMP-Felder zurückgegeben

werden. Duplikate werden wie bei Find Methods auf zwei Arten aus der Ergebnismenge ausgeschlossen: über das Schlüsselwort `DISTINCT` im Deployment Descriptor oder den Rückgabetyp `Set`.

Das folgende Beispiel zeigt, wie zu einem `RisikoBuendel` alle absoluten Zu- und Abschläge ermittelt werden (siehe Attribut: `ZuAbschlageStufe.wert`). Dazu wird zuerst in `RisikoBuendelBean` eine abstrakte Methode deklariert. Sie muss mit dem Präfix `ejbSelect` beginnen:

```
public abstract class RisikoBuendelBean
    implements javax.ejb.EntityBean
{
    //...
    public abstract Collection ejbSelectZuAbschlagWerte(
                int risikoBuendelFachId,
                Date maxDate
            ) throws FinderException;
   //...
}
```

Listing 4.18: select Method in der Bean-Klasse

Im Deployment Descriptor befindet sich dazu die folgende EJB-QL-Anfrage:

```
<query>
  <query-method>
    <method-name>ejbSelectZuAbschlagWerte</method-name>
    <method-params>
      <method-param>int</method-param>
      <method-param>java.sql.Date</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    <![CDATA[
    SELECT z.wert FROM ZuAbschlagStufeBean z
          WHERE z.risikoBuendelFachId = ?1
              AND z.hGueltigBis = ?2
    ]]>
  </ejb-ql>
</query>
```

Listing 4.19: EJB QL im Deployment Descriptor

Interessant ist an diesem Beispiel die `select_clause`. Durch `SELECT z.wert` wird der Rückgabewert auf einzelne Attribute der Bean `ZuAbschlagStufeBean` beschränkt. Das Ergebnis ist eine `Collection`, die nur Elemente vom Typ `ZuAbschlagStufe.wert` enthält.

Dieses Vorgehen hat den Vorteil, dass nur einzelne Werte geladen werden. Die Logik zum Instanziieren einer Bean, Anmelden der Bean beim Transaktionsmanager usw. wird eingespart.

4.5 Home Methods

Neben den create-, remove- und find-Methoden können seit der 2.0-Spezifikation auch andere Methoden im Home Interface deklariert werden: Home Methods.

Im Gegensatz zu Methoden aus dem Component Interface (Instanzmethoden) beziehen sie sich nicht auf die einzelne Instanz, auf der sie aufgerufen werden, sondern auf eine *Menge von Instanzen*. Eine solche Methode würde bei normaler Java-Programmierung durch das Schlüsselwort »static« als Klassenmethode gekennzeichnet. Bei Entity Beans wird die Unterscheidung zwischen Klassenmethoden und Instanzmethoden durch die Deklaration im Interface bestimmt. Klassenmethoden werden im Home Interface deklariert. Instanzmethoden sind alle Methoden, die im Component Interface deklariert sind, beispielsweise `get-` und `set`-Methoden auf ein bestimmtes Attribut.

In vorangegangenen Versionen der Spezifikation sind Klassenmethoden zumeist in eine SessionBean ausgelagert worden. Der Quelltext, der fachlich zu einer Bean gehört hätte, wurde so über mehrere Beans verstreut, weil es technisch nicht anders möglich war.

Zur Umsetzung muss eine Klassenmethode zuerst im Home Interface deklariert werden. Ihr Name darf nicht mit create, remove oder find beginnen. Signaturen, die im Remote Home Interface deklariert sind, müssen eine `RemoteException` enthalten. Parameter von Methoden im Remote Home Interface müssen über RMI-IIOP übermittelt werden können. Ansonsten dürfen Home Methods beliebige anwendungsspezifische Exceptions verwenden [18, S. 114f; S. 117f].

Eine Methode, die die durchschnittliche Prämie für alle Verträge des Bestands ermittelt, könnte durch eine Home Method wie folgt umgesetzt werden. Sie wird beispielsweise im Remote Home Interface deklariert. Daher muss ihre Signatur die RemoteException enthalten:

```
public double ermittleDurchschnittsPr"amie()
    throws RemoteException;
```

Listing 4.20: Deklaration einer Home Method im Interface

Die Implementierung der deklarierten Methode muss sich in der Bean-Klasse befinden und mit dem Präfix `ejbHome` gefolgt vom Methodennamen benannt sein. Sie muss `public`, darf jedoch nicht `static` sein.

```
public double ejbHomeErmittleDurchschnittsPr"amie() {
    //per ejbSelect alle Praemien bestimmen
    try{
        Collection c = ejbSelectAllePraemien();
        int anzahl = 0;
```

```
        double summe;
        while ( c.hasNext() ) {
            anzahl ++;
            //aufsummieren
            summe = ...
        }
        return summe / anzahl;
    } catch (...) {
        //...
    }
}
```

Listing 4.21: Implementierung einer Home Method in der Bean

Wichtig ist bei dieser Implementierung, dass keine Instanzmethoden der konkreten Bean, in der sie ausgeführt wird, aufgerufen werden können. Beim Programmieren einfacher Java-Klassen entspräche ein solcher Aufruf dem Aufruf einer nicht-`static` Methode aus einer `static` Methode. Solche Fehler würden bereits durch den Compiler erkannt und gemeldet. Da Beans jedoch nie direkt miteinander kommunizieren, sondern nur über Interfaces und den Container, müssen Klassenmethoden anders als über das Schlüsselwort `static` abgewickelt werden.

Klassenmethoden werden zu einem anderen Zustand im Lebenszyklus der Bean ausgeführt als Instanzmethoden. Entity Beans können in ihrem Lebenszyklus drei verschiedene Zustände einnehmen [18, S. 168]:

- **does not exist:** Dieser Zustand ist initial. Es existiert keine Instanz. Jeder Methodenaufruf führt zu einem Fehler.
- **pooled:** Aus dem Zustand »does not exist« wird eine Bean durch Erzeugen der implementierenden Klasse und dem Aufruf von `setEntityContext` in den Zustand »pooled« gebracht. Alle Instanzen in diesem Zustand sind gleich. Jede von ihnen könnte durch Belegung der Attribute zu einer konkreten Instanz werden, die einen persistenten Datensatz repräsentiert. In diesem Zustand können beispielsweise ejbSelect-Methoden und Home Methods aufgerufen werden.
- **ready:** Die im Pool vorgehaltenen Bean-Instanzen erhalten erst über den Aufruf von `ejbActivate` oder `ejbCreate` und `ejbPostCreate` eine Belegung ihrer Attribute. Sie sind nun im Zustand »ready« und stehen in Beziehung zu einem Datensatz. Erst jetzt sind die über die Component Interface deklarierten Instanzmethoden aufrufbar.

Home Methods werden also auf Instanzen im Zustand »pooled« aufgerufen. Das heißt die ausführende Instanz hat keine Identität und keine definierte Belegung ihrer Attribute. Home Methods könnte jedoch über eine Menge von Beans iterieren und auf ihnen Instanzmethoden aufrufen.

Zusammengefasst verbessern Home Methods die objektorientierte Kapselung von Entity Beans, da sie es ermöglichen, dass eine Methode und die durch sie bearbeiteten Daten in der gleichen Bean liegen können.

4.6 Run-As Security

Das »Run-As Security«-Konzept ist eine Erweiterung des bestehenden Berechtigungskonzepts für Enterprise JavaBeans [18, S. 433ff]. Die Berechtigungsprüfung, ob eine Methode von einem Aufrufer ausgeführt werden darf, wird nicht durch Abfragen in die Methode hineinkodiert. Es werden so genannte Rollen definiert, die ein Aufrufer einer Methode innehaben kann. Eine Rolle ist gekennzeichnet als eine Menge von Berechtigungen zur Ausführung verschiedener Methoden. Die Zuordnung, welche Rolle welche Methode ausführen darf, wird im Deployment Descriptor vermerkt, indem zu jeder Methode die berechtigten Rollen aufgezählt werden. Um die Berechtigung zu prüfen, wird die Identität des Aufrufers ermittelt und welche Rollen der Aufrufer einer Methode inne hat. Sie werden gegen die Rollen geprüft, die zu einer Methode definiert sind. Danach entscheidet sich, ob der Aufruf erlaubt oder verwehrt wird.

Ruft eine Methode eine weitere Methode auf, so wird der Aufrufer an die folgende Methode übermittelt. Falls der Aufrufer eine Berechtigung für die folgenden Methoden hat wird die Bearbeitung fortgeführt. Die Identität des Aufrufers wird in einem *Security Context* von Methode zu Methode überreicht, um bei jedem weiteren Methodenaufruf die Berechtigung prüfen zu können.

Die »Run-As«-Erweiterung des Berechtigungskonzepts besteht darin, dass die Identität des Aufrufers während der Ausführung des Programms ersetzt werden kann. An die Stelle der Identität des Aufrufers tritt die Identität, die zu einer Bean definiert wird. Die Bean selbst und jede von ihr aufgerufene Methode nutzen diese Identität zur Berechtigungsprüfung [18, S. 447].

Das folgende Listing zeigt *beispielsweise*, wie der Session Bean `BatchrahmenBean` die Identität »admin« zugewiesen wird. Greift ein Nutzer auf eine Methode dieser Bean zu, wird einmalig geprüft, ob er berechtigt ist, die Methode auszuführen. Die Methode selbst und alle darin aufgerufenen Methoden prüfen gegen die Identität »admin« auf Berechtigung.

```
<enterprise-beans>
    <session>
       <ejb-name>BatchrahmenBean</ejb-name>
        ...
       <security-identity>
        <run-as>
           <role-name>admin</role-name>
        </run-as>
       </security-identity>
    </session>
</enterprise-beans>
```

Listing 4.22: Deployment Descriptor

Diese Erweiterung kann genutzt werden, um sicherheitskritische Methoden flexibel in verschiedenen Kontexten verwenden zu können. Dazu werden in einer Bean alle sicherheitsrelevaten Methoden zusammengestellt. Diese Bean erhält eine Run-As-Identität,

die alle Methoden mit den nötigen Berechtigungen ausstattet. Durch Delegation wird die Bean mit eigener Sicherheitsidentität von anderen Beans genutzt. Die nutzenden Beans tragen Sorge, dass in den Nutzungskontexten nur bestimmte Anwender berechtigt sind.

Beispielsweise kann es eine Bean geben, die Funktionalität zum Ändern von Personaldaten beinhaltet. Sie wird ausgestattet mit einer Run-As-Identität, durch die sie alle notwendigen Berechtigungen erhält. An diese Bean delegieren zwei andere Beans. Eine von ihnen kann nur von Mitarbeitern aus der Personalabteilung aufgerufen werden. Die andere Bean kann von jedermann aufgerufen werden, soweit sichergestellt ist, dass nur die eigenen Daten gelesen und verändert werden können.

Eine Umsetzung ohne Run-As-Identität hätte erzwungen, dass der Quelltext zum Lesen und Ändern von Personaldaten zweimal implementiert wird: einmal mit der Berechtigung für Mitglieder der Personalabteilung und einmal, um jedermann zu erlauben, die eigenen Daten einzusehen und zu ändern.

Die Run-As-Identität vereinfacht das Abbilden von Berechtigungen in einem System. Quelltext kann in verschiedenen Kontexten wiederverwendet werden, da unterscheidbar ist, wer in welchem Nutzungskontext eine Berechtigung hat und welche Berechtigung eine Methode zu ihrer Ausführung benötigt. Diese Vereinfachung kann bei unkritischem Gebrauch zu Sicherheitslücken führen.

5 Performanz von EJB-Anwendungen

Das vorliegende Buch gehört in den Bereich der komponentenbasierten Softwareentwicklung, wie er in [30] beschrieben ist. Um Softwarekomponenten zu entwickeln und zu betreiben, werden häufig Applikationsserver eingesetzt. Das sind Softwaresysteme, die dem Entwickler einer Anwendung viele Infrastrukturaufgaben erleichtern. Sie können die Anbindung einer Applikation an eine Persistenzschicht übernehmen, bieten ein deklaratives Transaktionsmanagement, übernehmen die Verteilungsaspekte und steuern den Lebenszyklus einzelner Geschäftsobjekte. Sie arbeiten als zusätzliche Schicht zwischen der von einem Entwickler programmierten Applikation und einer Datenbank. Häufig werden sie auch als Component Transaction Monitor (kurz: CTM) bezeichnet, um anzudeuten, dass sie sich aus den Transaktionsmonitoren entwickelt haben. Eine Einführung in Transaktionsmonitore anhand bekannter Produkte bietet [6].

Aufgrund des geringen Alters der EJB-Technologie besteht kaum Erfahrung, wie mit ihr eine performante Umsetzung von Massendatenverarbeitungsprozessen implementiert werden kann. Das vorliegende Buch schließt diese Lücke, indem es Design-Regeln ermittelt, um eine möglichst hohe Performanz zu erzielen. Der hier benutzte Performanzbegriff wird im Unterkapitel 5.1 erläutert. Die Bedeutung von Performanz für Massendatenverarbeitung wird anhand des folgenden Beispiels aus der Versicherungswirtschaft illustriert.

Viele Unternehmen aus der Versicherungswirtschaft befinden sich gegenwärtig in der Situation, nach Alternativen zu ihren bestehenden EDV-Systemen suchen zu müssen. Diese Systeme sind durch die dauernde Wartung und Erweiterung unübersichtlich geworden, es fehlt an Nachwuchskräften, die bereit sind, sich in Hostsysteme einzuarbeiten, durch Zusammenschluss von Unternehmen entsteht ein abenteuerliches Konglomerat aus Systemen verschiedener Plattformen und Sprachen, die parallel weiterzupflegen sind. Neue Vertriebsstrategien fordern das Öffnen der Geschäftsprozesse für den Kunden durch Anbindung der Bestandsführungen an das Internet, was zusätzlich zu den anderen Wartungsarbeiten zu bewältigen ist. Angesichts dieser Probleme besteht der starke Wunsch, neue Systeme zu schaffen.

Ein Bereich produktiver Systeme ist die automatische Massendatenverarbeitung, bei der alle Geschäftsobjekte eines Bestands ohne Interaktion mit einem Nutzer nach programmierten, d.h. vordefinierten, Regeln bearbeitet werden. Diese so genannten Batchläufe müssen sehr performant sein, da das große Volumen der Daten, das bei Versicherungen vorliegt, zu bearbeiten ist (vgl. [69, S. 15ff]). Auf diesen Daten arbeiten Batchprozesse für unterschiedliche geschäftliche Vorgänge wie Mahnungen, automatische Umstufungen, statistische Prozesse für Controlling und Prozesse zur Datensicherung. Dem einzelnen Prozess bleibt daher nur ein kleines Zeitfenster für die Bearbeitung großer Datenmengen.

Eine Neuentwicklung als Alternative für ein bestehendes System muss Geschäftsprozesse mit großem Datenvolumen in kurzer Zeit abwickeln können. Wenn *komponentenbasierte* Systeme bestehende Anwendungen ablösen sollen, so muss es auch möglich sein,

mit komponentenbasierten Systemen performante Batchläufe zu implementieren. Dies ist das performanzbedingte Entwicklungsrisiko, von dem in der Einleitung zu diesem Buch gesprochen wurde. Kann das komponentenbasierte System den Massendatenverarbeitungsprozess nicht in dem zur Verfügung stehenden Zeitfenster abarbeiten, so kann das System nicht eingesetzt werden. Es muss mit hohen, nicht eingeplanten Kosten für Tuning und Hardware gerechnet werden, um ein Scheitern der Neuentwicklung zu vermeiden [53].

Um das beschriebene Risiko zu mindern, muss beim Erstellen eines Systems die Auswirkung einer Designentscheidung auf die Performanz abschätzbar sein.

Ziel des Buches ist es, für das Komponentenmodell EJB 2.0 systematisch und nachvollziehbar Design-Regeln für verbesserte Performanz von Massendatenverarbeitungsprozessen zu ermitteln.

5.1 Performanz

»Performanz definiert sich als eine Aussage über die Fähigkeit eines Anwendungssystems, eine Aufgabe oder eine Aufgabenmenge in einer bestimmten Zeitspanne bewältigen zu können.« [53].

Maße für Performanz werden drei Bereichen zugeordnet [52, S. 1385]:

1. **Antwortverhalten (»responsiveness«):** ist die Zeitspanne zwischen Abschluss der Eingabe und Erhalten des letzten Zeichens der errechneten Ausgabe. Das System benötigt diese Zeit zur Bearbeitung der Eingabe. Bei interaktiven Systemen wird diese Zeit als response time bezeichnet. Für Batchverarbeitungssysteme ist das Maß turnaround time charakteristisch.

2. **Durchsatz (»throughput«):** ist der Quotient aus bewältigten Arbeitspaketen pro Zeiteinheit. Was genau ein Arbeitspaket ist, lässt sich nicht allgemein definieren. Bei Datenbanksystemen wird throughput beispielsweise durch die Anzahl an Transaktionen pro Sekunde angegeben.

3. **Kosten (»cost«):** beschreiben den monetären Aufwand, der zu leisten ist, um ein Computersystem zu kaufen oder zu leasen. Es ist stets zu bedenken, welche Kosten erzeugt werden, um ausreichend Ressourcen zur Verfügung zu haben, die einen angestrebten Durchsatz oder eine angestrebte Antwortzeit ermöglichen.

Um Performanz messen zu können, muss zuvor eine Arbeitslast (workload) bestimmt werden. Dabei handelt es sich um die Menge aller Anfragen oder Befehle, die an ein System während eines definierten Zeitraums gerichtet wird. Anschaulich beschrieben ist das das Volumen der zu bewältigenden Aufgabe. Die Performanz einer Anwendung ist nur aussagekräftig, wenn bekannt ist, bei welcher Arbeitslast (»workload«) sie ermittelt wurde. Besteht keine Möglichkeit, eine reale Arbeitslast an einem produktiven System zu ermitteln, so ist eine Arbeitslast zu simulieren, die repräsentativ für das zu messende System ist (vgl. [34, S. 60ff]). Die Arbeitslast für den Kontext des vorliegenden Buches wird ausführlich in Abschnitt 7.1 vorgestellt.

Das Maß »turnaround time« wird in [21, S. 14-20] erläutert. Es bezeichnet die Zeit, die ein System zur Bearbeitung eines Batchjobs (siehe Abschnitt 5.3) benötigt. Die gemessene Zeit startet mit dem Einreihen des Arbeitsauftrags in die Eingabewarteschlange und endet mit der Ausgabe des Ergebnisses.

Für den Kontext dieses Buchs ist der prototypische Batchlauf der einzige Bestandteil der Arbeitslast während der Messung. Da es in der Praxis üblich ist, das produktive System für die Dauer der Batchverarbeitung von Nutzereingaben abzuschirmen, entspricht das einer realistischen Arbeitslast.

Für diesen Fall wird als Maß die *processing time* benutzt. Synonym zu »processing time« wird auch der Begriff stand-alone turnaround time benutzt. Dieses Maß ergibt sich aus der turnaround time abzüglich der Wartezeit, die ein Batchjob verbringt, bis die vor ihm eingereihten Jobs abgearbeitet bzw. ausgegeben sind. Die Messung dieser Größe wird in Abschnitt 7.2 dargelegt.

Die Messung und Bewertung von Performanz dient einem der folgenden vier Ziele [21]:

1. **Beratung und Auswahl**: Systemkomponenten sollen aus einer Auswahl verfügbarer Produkte gewählt werden. Um die Produkte vergleichen zu können, müssen sie zuvor nach anerkannten Testverfahren bezüglich ihrer Performanz bewertet worden sein.

2. **Verbesserung bestehender Systeme**: Ein produktives System soll bezüglich seiner Performanz verbessert werden. Dazu wird ermittelt, wo Performanzengpässe sind, um diese zu eleminieren. Diese Aufgabe fällt in den Bereich der Wartung und Pflege eines Systems. Der überwiegende Teil an Literatur zum Thema Performanzverbesserung gehört in diesen Bereich. Laut [53] wird häufig versucht, *nachträglich* ausreichende Performanz in ein neu erstelltes Programm zu bringen. Der Performanzgewinn durch nachträgliche Verbesserung ist jedoch wesentlich teurer, als wenn Performanzanforderungen bereits anfänglich berücksichtigt würden. Wesentliche Verbesserungen sind häufig nur durch ein Redesign möglich.

3. **Kapazitätsplanung**: Die Kapazitätsplanung versucht, möglichst genaue Voraussagen zu ermitteln, wie sich die Arbeitslast und die Performanz entwickeln werden. Das ermöglicht, vorausschauend Maßnahmen zu ergreifen, um Performanzengpässe zu vermeiden. Die Voraussagen beruhen auf einem Modell des betrachteten Systems. Ein Beispiel für die verbreitete Modellierung mit Petri-Netzen findet sich in [40].

4. **Design**: Bei der *Erstellung* eines neuen Systems wird ermittelt, ob die zugesagten Anforderungen an die Performanz eingehalten werden können und die weitere Entwicklung auf die Erreichung dieser Anforderungen ausgerichtet. [53] stellt heraus, dass dieses Vorgehen wesentlich erfolgreicher als ein ex-post Verbessern der fertigen Anwendung ist, um ein performantes System zu erstellen.

In dem vorliegenden Buch wird Performanzmessung betrieben, um ein performantes Design für Massendatenverarbeitung auf EJB 2.0-Basis zu ermitteln. Anhand der *processing time* werden verschiedene Verbesserungen einer Basisvariante verglichen, um aus der Kombination der Verbesserungen ein allgemeines Design bestimmen zu können.

5.2 Software Performance Engineering

Die Anforderungen an die Performanz müssen in allen Phasen eines Entwicklungsprojekts berücksichtigt werden, da nachträglich nicht beliebige Änderungen vorgenommen und Verbesserungen erreicht werden können. Software Performance Engineering ist eine Teildisziplin der Informatik, die beschreibt, wie Performanzanforderungen im Softwarelebenszyklus geeignet berücksichtigt werden müssen. Die Vorgehensweise des Software Performance Engineering ist herstellerneutral und unabhängig vom eingesetzten System [53].

Performance Tuning bezeichnet im Gegensatz dazu den Prozess, ein fehlerfreies Anwendungsprogramm und sein Laufzeitumfeld nach Abschluss der Entwicklung soweit zu verändern, bis es den Anforderungen an die Performanz genügt. Das Vorgehen beim Performance Tuning ist abhängig vom eingesetzten System und den eingesetzten Produkten.

Um festzustellen, wie ausgereift der Softwareentwicklungsprozess bei der Berücksichtigung von Performanzanforderungen ist, hat [56] auf Basis des Capability Maturity Model (CMM) das Performance Engineering Maturity Model (PEMM) erstellt. Es liefert Kriterien für eine gelungene Integration der Performanzanforderungen in den unternehmensweiten Entwicklungsprozess.

Einzelne Ansätze des Software Performance Engineering gehen von den weit verbreiteten UML-Diagrammen aus, um möglichst früh in der Enwicklungsphase Aussagen über die Performanz treffen zu können. Da Änderungen am Design in den frühen Phasen eines Entwicklungsprojekts billiger sind als zum Ende der Entwicklung, liegt der Vorteil dieses Herangehens auf der Hand.

[13] beschreibt, wie aus Anwendungsfalldiagrammen, Sequenzdiagrammen und Einsatzdiagrammen (vgl. zu den Diagrammtypen [23]) ein auf Warteschlangen basiertes Performanzmodell abgeleitet werden kann. Das beschriebene Vorgehen ist soweit formalisiert, dass es algorithmisch umgesetzt werden kann. Dieser Ansatz erfordert keine zusätzlichen Modellierungsschritte und ist daher nicht zeitaufwendiger als ein Entwurf der Performanzanforderungen nur am Rand berücksichtigt. In der Entwurfsphase eines Projektes sind die genannten Diagramme ohnehin zu erstellen. Wenn sie hinsichtlich der Performanz des entstehenden Systems automatisiert ausgewertet werden können, lässt sich ohne Aufwand zu jedem Zeitpunkt die Auswirkung einer Design-Änderung auf die Performanz ermitteln.

[2] entwickelt aus Klassen- und Sequenzdiagrammen eine Simulation des Systemverhaltens. Diese Simulation gibt Aufschluss, ob mit dem simulierten System die Anforderungen an die Performanz eingehalten werden können. Dieser Ansatz erfordert keine zusätzlichen Entwurfsaufwände, benötigt jedoch ein Werkzeug.

[32] setzt ebenfalls auf den Ansatz, UML-Diagramme in Warteschlangenmodelle zur Performanzvorhersage zu verwandeln. Er hebt jedoch den Aspekt der schrittweisen Verfeinerung hervor. Ebenso wie UML-Diagramme von einem abstrakten, unscharfen zu

einem konkreten Design verfeinert werden, ist es möglich, die Performanzanforderungen von globalen Anforderungen zu spezifischen Anforderungen an einzelne Klassen und Methoden zu verfeinern. Mit jeder Verfeinerung des Designs werden auch die Performanzanforderung an die Bestandteile des Systems konkreter. Ein konkrets Warteschlangenmodell für Applikationsserver und andere dreischichtige Architekturen wird in [41] erarbeitet.

Die beschriebenen Ansätze können nicht ohne unterstützende Werkzeuge angewendet werden. Leider ist noch keines dieser Werkzeuge als Produkt verfügbar. Durch die mangelnde Praktikabilität und Erprobung in der Praxis fehlen unabhängige Erfahrungswerte über die Aussagefähigkeit der mit den Werkzeugen ermittelten Ergebnisse. Da das Buch unmittelbar verwendbare Design-Empfehlungen liefern soll, scheiden die genannten Ansätze aus.

Alternativ ermittelt und validiert das Buch Pattern (deutsch: Entwurfsmuster, vgl. [26]). Entwurfsmuster werden in Katalogen nach einem definierten Schema dokumentiert, sodass zu einem Problem ein erprobter Entwurfsansatz gesucht werden kann. Durch Entwurfsmuster wird die Wiederverwendung auf Ebene des Designs verstärkt. [44] schlägt die Erweiterung des Schemas für Entwurfsmuster vor, sodass Hinweise auf Performanzrelevante Eigenschaften nachgeschlagen werden können. Im Bereich des Performance Engineering gibt es Entwurfsmuster zur Steigerung der Performanz und AntiPattern, die es zu vermeiden gilt. [61] stellt einige AntiPattern vor und beschreibt, mit welchen Pattern sie vermieden werden können. Unterkapitel 8.4 legt dar, warum Kommunikation zwischen Enterprise JavaBeans nach dem Standard 1.1 teuer ist. Bei EJB 1.1-Anwendungen sollte daher beispielsweise das Pattern »God« vermieden werden. Es erzeugt übermäßig viel Kommunikation, indem Funktionalität verschiedener Klassen in einer Klasse zusammengeballt wird, die zur Ausführung dennoch der Information der anderen Klassen bedarf. Solche Klassen haben oft die Begriffe »Manager« oder »Controller« im Namen.

Dieses Buch diskutiert verschiedene Design-Entscheidungen und bündelt sie zu einem allgemeinen Entwurf für Massendatenverarbeitungsprozesse. Um die Design-Entscheidungen in ihrer Wirksamkeit zu bestätigen, werden verschiedene Prototypen gebildet. Sie unterscheiden sich in ausgewiesenen Design-Merkmalen. Im Vergleich zu einem Prototypen, der über das ausgewiesene Design-Merkmal nicht verfügt, lässt sich der Einfluss dieses Design-Merkmals ermitteln. Dieses Vorgehen hat den Vorteil, dass die Ergebnisse sehr anschaulich und konkret sind. Sie können daher direkt von Entwicklern in Projekten umgesetzt werden.

5.3 Fachliche Grundlagen

Das Buch nutzt als praktisches Beispiel einen Geschäftsprozess aus der Versicherungswirtschaft. Als Nachschlagewerk für versicherungsfachliche Begriffe wurde [25] benutzt. Die fachliche Gestaltung der implementierten Klassen ist in kritischer Auseinandersetzung mit dem Analysemodell in [68] entstanden.

Eine Anforderung an ein produktives System in der Versicherungswirtschaft ist die Historisierung , d.h. die Möglichkeit, die zeitliche Entwicklung eines Geschäftsobjekts zurückverfolgen zu können. Soweit notwendig deckt das Beispiel dieses Buchs auch Historisierungsfunktionalität ab. Die Implementierung beruht auf [33], der eine zweidimensionale Historisierung umsetzt, indem er jedem Geschäftsobjekt (bzw. jedem Teil davon) einen Gültigkeits- und einen Erfassungszeitraum zuweist.

Als Beispielgeschäftsprozess wird ein Batchverarbeitungslauf (kurz: Batchlauf) aus der Versicherungswirtschaft gewählt. Im Folgenden wird dargelegt, was unter einem Batchlauf zu verstehen ist und welche besonderen Anforderungen an ihn zu stellen sind.

Der Begriff Batchlauf (syn.: Batchverarbeitung) stammt aus der Gründerzeit der Informatik. Es wurden zwei Betriebsarten für Programme unterschieden (siehe [71, S. 31f]):

1. **Online-Operation**: Über Terminals bedienen Nutzer interaktiv Programme. Die Ergebnisse der Programmausführung werden dem Nutzer direkt mitgeteilt durch das Terminal. Der Nutzer ist online in dem Sinne, dass er eine direkte Verbindung zu dem Programm unterhält, das seine Eingaben auf dem Zentralrechner verarbeitet. Heute wird der Begriff »online« hauptsächlich für die Möglichkeit benutzt, sich ins Internet einzuwählen.
2. **Batch-Operation**: Die Anforderungen zur Abänderung von Dateien oder Erlangung der Information aus Dateien werden gesammelt und zu einem gewissen Zeitpunkt nacheinander abgearbeitet. Der Begriff Batch-Operation ist dadurch geprägt, dass die Lochkarten, die das Programm beschrieben, in der Abarbeitungsreihenfolge auf einem Stapel (engl.: Batch) abgelegt wurden. Dadurch ergibt sich die Möglichkeit, die Einzelanforderungen optimal anzuordnen, sodass die Systemressourcen eine hohe Auslastung haben. Aus Nutzersicht nachteilig ist die Wartezeit auf das Ergebnis der Anfrage. Batchverarbeitung ist also gekennzeichnet durch eine sequenzielle Abarbeitung von vordefinierten Arbeitsschritten, die eine Korrektur oder Eingabe eines Nutzers ausschließen.

Mittlerweile ist es zum Standard geworden, Computer und die darauf laufenden Programme interaktiv zu bedienen. Der Begriff Batchlauf ist aber weiterhin gebräuchlich im Umfeld großer Datenbanksysteme, wie sie bei Banken und Versicherungen im Einsatz sind. Als Batchlauf wird dort ein computerbasierter Geschäftsprozess beschrieben, der ohne Interaktion mit einem Nutzer gewisse Berechnungen durchführt.

Das Fehlen der Nutzerinteraktion macht Batchläufe als Beispiele für Performanzmessungen an Enterprise JavaBeans interessant. Applikationsserver-basierte Anwendungen haben mehrere Schichten, die sich in ihren Aufgaben unterscheiden. Abbildung 5.1 stellt die typischen Schichten einer solchen Anwendung schematisch dar.

Zuunterst sind die Datenbanken dargestellt, in denen die persistenten Informationen abgelegt werden (Persistenzschicht). Die beiden folgenden Schichten gehören zum Applikationsserver. Die Schicht, in der in Form von Enterprise JavaBeans die Geschäftslogik residiert, liegt direkt über der Persistenzschicht. Darauf folgt eine Web-Container-Schicht, in der Information aufbereitet wird für die Interaktion mit dem Client. Diese Schicht

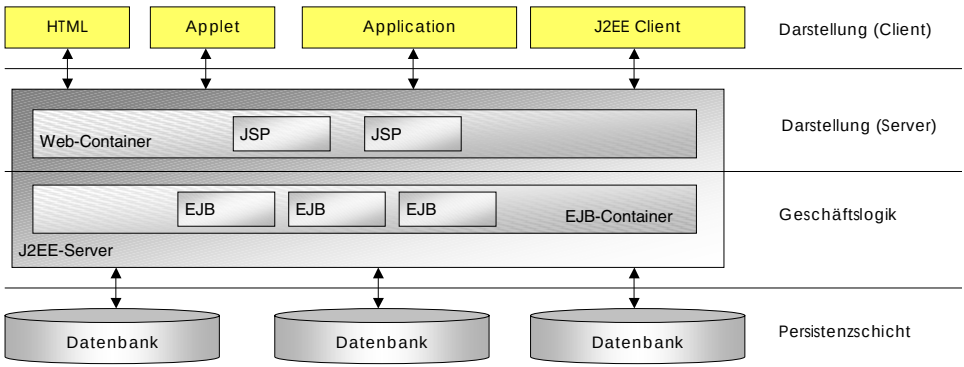


Abbildung 5.1: Schichten einer EJB-Anwendung

muss nicht immer ausgeprägt sein. Die serverseitige Darstellung kann bereits durch Java-Klassen oder EJBs erzeugt werden (vgl. Abbildung 3.5). Die oberste Schicht dient der clientseitigen Darstellung. Als Client können HTML-Seiten, Applets, Applikationen oder neuerdings J2EE-Clients benutzt werden.

Eine Aussage über die Performanz von Enterprise JavaBeans betrifft allein die Schicht der Geschäftslogik und die Persistenzschicht, auf die die Geschäftslogik zur Speicherung der Geschäftsobjekte angewiesen ist. Werden weitere Schichten in die Betrachtung einbezogen, so verfälscht ihr Einfluss die Aussage über die Performanz von EJBs. Die Performanzbetrachtung gestaltet sich einfacher und verständlicher, wenn das Laufzeitverhalten von EJBs isoliert betrachtet werden kann. Die so gewonnenen Ergebnisse können in einem zweiten Schritt auf interaktive Anwendungen übertragen werden.
Batchanwendungen sind Anwendungen, die allein die beiden untersten Schichten benötigen, da ihnen die Interaktion mit einem Client fehlt. An ihnen kann das Verhalten von EJBs isoliert betrachtet werden. Gleichzeitig sind Batchanwendungen geschäftskritisch, sodass die Einschränkung der Perspektive die Relevanz des behandelten Problems nicht vermindert.

Die technische Gestaltung und Relevanz produktiver Batchsysteme im Versicherungswesen wird in [69] beschrieben. Der Gesamtverband der Deutschen Versicherungswirtschaft, GDV, hat unter dem Projektnamen VAA ein allgemeines Konstruktionskonzept veröffentlicht (siehe: [68], [69]). Dadurch wird eine Referenz geschaffen zur schnelleren und kostengünstigeren Entwicklung von Versicherungssoftware.

Der Batchverarbeitung ist in der Referenzarchitektur ein eigenes Kapitel gewidmet. Es wird dargelegt, welche Gründe Batchläufe in einer *Referenzarchitektur von 1999* notwendig machen. Laut [68] stellt Batchverarbeitung in den folgenden Bereichen Anforderungen, die über die Anforderungen einer interaktiven Anwendung, d.h. einer Online-Anwendung, hinausgehen.

- Performanz: Das große Datenvolumen an gespeicherten Geschäftsobjekten ist komplett zu bearbeiten bis zu einem gewissen Zeitpunkt. Durch die Vorbereitung des

Drucks der Rechnungen zum Jahresinkasso darf z.B. nicht das Tagesgeschäft der Bestandsverwaltung, Sicherungsläufe oder ein Randsystem blockiert werden.

- hoher Ressourcenbedarf aufgrund großer Datenmengen

Insbesondere bezüglich der Performanz wird im VAA-Dokument darauf hingewiesen, dass »evtl. besondere Vorkehrungen bezüglich des Datenzugriffs sowie der Modellierung der entsprechenden Datenbanken zu treffen« [69, S. 15ff]) sind, um eine ausreichende Performanz zu erhalten. Die Performanz der Batchverarbeitung ist also so wichtig, dass durch sie die Datenstrukturierung für die ganze Bestandsführung bestimmt wird.

Der Wortgebrauch des Buchs folgt der Begriffsprägung von »Batchlauf«, wie sie im Großrechnerumfeld der Versicherungswirtschaft üblich ist: Ein Batchlauf ist ein Geschäftsprozess, der durch Folgendes gekennzeichnet ist:

- **automatischer Ablauf:** Der Geschäftsprozess findet in einem Computersystem ohne Interaktion mit einem Benutzer statt.
- **große Anzahl von Geschäftsobjekten:** Alle Geschäftsobjekte (z.B. Verträge, Konten, Partnerdaten) werden nach geschäftsprozessabhängigen Kriterien untersucht, ob sie an dem Prozess teilnehmen. Es wird schließlich eine Teilmenge der Geschäftsobjekte bearbeitet und damit verändert.
- **definiertes Ergebnis zu einem organisatorischen Stichtag:** Der Geschäftsprozess hat ein klar definierbares Ergebnis (etwa eine ausgedruckte Rechnung pro Kunde; neu berechnete Rabatte für jeden gültigen Vertrag), das zu einem gewissen Zeitpunkt vorliegen muss. Er ist damit oft *zeitkritisch*.
- **sequenzielle, parallelisierbare Abarbeitung:** Die Bearbeitung der Geschäftsobjekte erfolgt einzeln hintereinander. Jedoch ist das Ergebnis der Bearbeitung eines Geschäftsobjekts nicht abhängig von dem Ergebnis der Bearbeitung des vorangegangenen. Theoretisch lässt sich die Bearbeitung parallelisieren.
- **komplexe Transaktionen:** Die Bearbeitung jedes einzelnen Geschäftsobjekts muss transaktionssicher erfolgen.
- **Unterbrechbarkeit:** Nach Abschluss des Batchprozesses muss sichergestellt sein, dass jedes Geschäftsobjekt genau einmal bearbeitet worden ist. Der Geschäftsprozess muss trotz eines Systemabsturzes an einer genau bestimmbaren Stelle wieder aufgenommen werden können, ohne dass Geschäftsobjekte übersprungen oder doppelt bearbeitet werden.

Die VAA nennt als Beispiele für Geschäftsvorfälle, die vorzugsweise als Batch abzuarbeiten sind: Massendatenpflege, Statistiken, Druck, Zahlungseinzug und Mahnwesen.

5.4 Technische Grundlagen

Als konkretes Komponentenmodell wird der EJB 2.0-Standard (Final Release vom 22. August 2001) [18] eingesetzt. Als erläuternde Einführungen für die EJB-Technologie

kann [24] und [46] herangezogen werden. [42] und die Spezifikation [9] beschreiben detailliert das Transaktionsmanagement im EJB-Umfeld.

Als Applikationsserver wird »Bea WebLogic 6.1« benutzt. Er war zu Beginn der Arbeiten an diesem Buch der fortschrittlichste Applikationsserver, der den EJB 2.0-Standard als »Proposed Final Draft 2« unterstützt hat. Einziger Unterschied vom »Proposed Final Draft 2« zum »Final Release« ist ein Detail bei der Umsetzung der Container-Managed Relations (vgl. Abschnitt 4.2). Durch diesen Umstand ergeben sich jedoch keine Einschränkungen der Ergebnisse dieses Buchs.

Der Applikationsserver ist an eine relationale Datenbank (Oracle 8i) angebunden. Diese Werkzeugkombination wird auf den Webseiten der Firma Bea detailliert beschrieben und ist daher einfach nachzuvollziehen. Die Oracle-Datenbank kann aber auch durch eine andere relationale Datenbank ersetzt werden, ohne die Ergebnisse des Buchs wesentlich zu verändern.

Ein Teil des Buchs beschäftigt sich daher mit Performanz von Datenbanksystemen. Performanz von Datenbanksystemen wird in der Literatur in zweierlei Weise benutzt:

1. In der Mehrzahl der wissenschaftlichen Artikel wird damit die Verbesserung des Laufzeitverhaltens der *Implementierung* eines Datenbanksystems beschrieben. Beispielsweise geht es darum, wie eine automatische Optimierung von Suchanfragen (CBO: »Cost based Optimizer«) in einem Datenbanksystem umgesetzt werden kann. Es geht also darum, wie Performanz in eine noch zu entwicklende Datenbank integriert wird.

2. Die andere Sichtweise beschäftigt sich mit der Frage, wie *Applikationen*, die auf einer fertig implementierten Datenbank aufsetzen, beschleunigt werden können. In diesen Bereich fällt die Beschleunigung von Suchanfragen durch die *Verwendung* von Indexen und von Partitionierung. Die Datenbank steht also fertig implementiert zur Verfügung. Es muss bestimmt werden, wie die Möglichkeiten der Datenbank nutzbringend von einem *Client* der Datenbank angesprochen werden können.

Die zweite Sichtweise ist relevant für dieses Buch. Die Begründung dafür liegt darin, dass der Applikationsserver einer EJB-Architektur als Client der Datenbank fungiert. Das Buch muss Wege aufzeigen, die Performanzverbesserungsmöglichkeiten der Datenbank für EJB-Applikationen nutzen zu können. Dazu gehören folgende Ansätze:

- **Analyse der Anfragen:** Durch die Nutzung eines geeigneten Index können Anfragen, die kleine Ergebnismengen produzieren, laut [27] beschleunigt werden. Joins arbeiten nach [27] besser, wenn die Menge mit den wenigsten Elementen zuerst ermittelt wird. Die Nutzung von Indexen zur Beschleunigung von Suchanfragen wird laut [55] eingeschränkt durch erhöhte Update-Kosten.

- **Segmentierung und Partitionierung:** In [22] wird beschrieben, wie sich Performanz durch horizontale und vertikale Segmentierung gewinnen lässt. Dabei wird auf Ebene des Datenbankschemas eine Tabelle in mehrere *Tabellen* gespalten.

Bei Partitionierung wird eine Tabelle in mehrere *Wartungseinheiten* zerlegt. Nach außen werden diese Wartungseinheiten weiterhin wie *eine* Tabelle angesprochen [49]. Dadurch werden insbesondere Änderungen an dem Client der Datenbank vermieden.

- **Performanzgewinn auf Ebene des Design:** Welchen Einfluss auf die Performanz verschiedene Mappings von Objekten auf Tabellen haben können, wird in [37] und [12] beschrieben. Durch Zusammenfassen von Relationen können laut [43] Joins über Daten vorweggenommen werden. Man spricht in diesem Zusammenhang auch vom »Clustern« von Relationen. Die Joins müssen dann nicht mehr zur Laufzeit vorgenommen werden. Die falsche Verwendung mehrerer binärer Relationen anstelle einer mehrstelligen Relation wirkt sich laut [19] negativ auf das System aus.

- **Sperrsituation:** Um mehr parallele Verarbeitung erreichen zu können, sind Transaktionen darauf zu überprüfen, ob durch Umordnen und Spalten der Transaktion die Anzahl und Dauer der Sperren auf der Datenbank zu vermindern ist (vgl. [59] und [58]). Die Isolierungsgrade der Transaktionen sind zu überprüfen. Falls organisatorisch sichergestellt ist, dass keine konkurrierenden Veränderungen vorgenommen werden können, so kann der Isolierungsgrad der Transaktionen reduziert werden. Dadurch wird ein höherer Durchsatz in der Datenbank erzeugt [6, S. 214ff].

- **Stored Procedures:** Stored Procedures sind in der Datenbank gespeicherte Funktionen. Sie sind bereits durch die Datenbank bezüglich ihrer Zugriffspfade optimiert. Deshalb und wegen ihrer Nähe zu den Daten sind sie sehr performant [27]. Es ist denkbar, diese Stored Procedures aus Enterprise JavaBeans anzusprechen und so performanzkritische Teile der Geschäftslogik auszulagern [54].

  Als wesentlicher Nachteil von Stored Procedures wird die schlechte Portabilität von Datenbank zu Datenbank herausgestellt. So werden Stored Procedures für Oracle 8i in PL/SQL geschrieben, das nicht auf IBM UDB zu portieren ist. Soll die Anwendung auf verschiedenen Datenbanken lauffähig sein, so entsteht zusätzlicher Aufwand.

  Die Nutzung von Stored Procedures führt weiterhin zu einer verschlechterten Wartbarkeit der Anwendung, da Teile der Geschäftslogik sowohl auf Ebene der Beans als auch auf Ebene der Stored Procedures ausprogrammiert werden müssen. Beispielsweise muss eine Änderung des Schadenfreiheitsrabatts mit anschließender Neuberechnung der Prämie sowohl im Online-Betrieb als auch im Batchbetrieb zur Verfügung stehen. Diese verschiedenen Teile müssen identische Funktionalität abdecken und sind daher durch aufwendige Tests auf funktionale Gleichheit zu überprüfen. Verändert sich die Anforderung an die Anwendung, so ist diese Veränderung in beiden Bereichen konsistent durchzuführen.

[66] und [49] liefern für die Umsetzung der theoretischen Konzepte auf der konkret eingesetzten Datenbank praktische Hilfestellungen.

Neben dem Versuch, Performanz auf Ebene der Datenbank und Datenbankanbindung zu gewinnen, sprechen Praxisberichte wie [1] und [70] davon, mit Caching auf Ebene des Applikationsservers die Anzahl der Datenbankzugriffe zu mindern. Die Anzahl der Erfahrungsberichte ist jedoch gering und die Aufarbeitung des Themas darin unsystematisch. Insbesondere wenn noch andere Anwendungen den Datenbestand neben dem App-

likationsserver nutzen, ergeben sich Synchronisationsprobleme, die den Gewinn durch Caching in Frage stellen. Caching wird nicht von allen Applikationsservern implementiert, da es nicht zum EJB-Standard gehört. Ergebnisse, die mithilfe von Caching auf einem Server ermittelt worden sind, lassen sich daher nicht auf anderen Servern reproduzieren. Caching wird daher in diesem Buch nicht gesondert behandelt.

In Praxisberichten wird als Grund für Inperformanz stets die Erstellung des Objekts aus den Zeilen der Datenbanktabellen genannt. Um Performanz zu gewinnen, ist daher zu betrachten, an welchen Stellen die Erzeugung von Beans zur Laufzeit vermieden werden kann. Das heißt massenorientierte Verarbeitung findet in der Datenbank statt, sodass zur Bearbeitung eines Geschäftsprozesses gar nicht erst alle Informationen in Form von Beans vorliegen müssen.

Neben den genannten Aspekten sind die Neuerungen des EJB 2.0-Standards auf ihre Performanzvorteile gegenüber den Lösungen nach dem 1.0- oder 1.1-Standard zu überprüfen. Diese Neuerungen wurden bereits ausführlich in Kapitel 4 beschrieben.

6 Anwendungsbeispiel

In dem vorliegenden Buch wird ein performantes Design für Massendatenverarbeitung ermittelt. Dazu wird ein Anwendungsbeispiel in verschiedenen Varianten implementiert. Die Varianten unterscheiden sich durch gezielte Veränderungen im Design. Hat die Veränderung einen Performanzgewinn gebracht, so ist das an der verringerten stand-alone turnaround time nachzuweisen.

Die folgenden Abschnitte skizzieren das Anwendungsbeispiel und unterziehen es einer *objektorientierten Analyse*:

»Ein Bestand von KFZ-Versicherungsverträgen wird bearbeitet, sodass den Verträgen, die keine Leistungen der Versicherung in Anspruch genommen haben, ein günstigerer Schadensfreiheitsrabatt (`SfKlasse`) auf die Prämie gewährt wird.«

Unterkapitel 6.1 erläutert fachlich die statische Seite des Beispiels, d.h. die zu speichernden Daten, aus denen sich ein KFZ-Versicherungsvertrag zusammensetzt. Der Geschäftsprozess liest und verändert diese Daten. Es wird die Tauglichkeit der Struktur für das Ziel dieses Buchs begründet.

Daran anschließend wird in Unterkapitel 6.2 die dynamische Seite, d.h. *wie* der Geschäftsprozess seine Arbeit verrichtet, beschrieben. Auch die Tauglichkeit des Geschäftsprozesses wird begründet.

In dem abschließenden Unterkapitel 6.3 wird auf die Unterschiede des Beispiels zu einem produktiven System hingewiesen.

6.1 Das Geschäftsobjekt

Damit an dem Geschäftsobjekt Unterschiede verschiedener Implementierungsvarianten nachgewiesen werden können, muss es den folgenden Auswahlkriterien genügen:

1. **Aggregat aus Teilobjekten:** Das Geschäftsobjekt muss sich über mehrere technische Objekte erstrecken, um daran zeigen zu können, wie sich verschiedene Zugriffsstrategien auf die Performanz auswirken.

2. **Realistischer Anteil der Nutzinformation:** Da nicht jeder Geschäftsprozess alle Information liest oder ändert, die in einem Vertrag gespeichert ist, kann man aus der Perspektive des Geschäftsprozesses zwischen Nutzinformation und nutzloser Information unterscheiden. Beispielsweise ist die Information, ob bei einem Vertrag automatisch gemahnt werden soll, völlig unerheblich für die automatische Veränderung des Schadenfreiheitsrabatts.

   Es muss ein realistisches Verhältnis zwischen Nutzinformation und nutzloser Information bestehen. Damit ist nachweisbar, ob es einen Unterschied macht, einen Teil der Arbeit in der Datenbank erledigen zu lassen. Etwa die Selektion von »nützlicher

Information« kann in der Datenbank stattfinden, sodass nur noch diese an den Applikationsserver übermittelt wird. An solchen Fällen kann auch die Auswirkung von Partitionierung der Datenbanktabellen aufgezeigt werden.

3. **Spielraum beim objekt-relationalen Mapping:** Das objekt-relationale Mapping ist die Abbildung zwischen den Attributen eines Geschäftsobjekts und der Datenbank, in der es gespeichert ist. Durch verschiedene Arten von Beziehungen (optionale Beziehungen, obligatorische Beziehungen, Aggregationsbeziehungen, …) zwischen Teilobjekten lassen sich Auswirkungen verschiedener Mappings zeigen.

Diese Überlegungen haben dazu geführt, einen Versicherungsvertrag als Beispiel zu wählen. Versicherungsverträge sind in sich hierachisch strukturiert (vgl. Abbildung 6.1). Die oberste Stufe der Hierachie bildet das `VertragsBuendel` als umschließendes Element. Es enthält einen oder mehrere Verträge, die aus `RisikoBuendeln` bestehen, welche sich schließlich aus mehreren Instanzen von `ElementarRisiko` zusammensetzen. Ein `ElementarRisiko` repräsentiert das einzelne versicherte Ereignis.

Bei einer Bearbeitung müssen also Zugriffe auf verschiedene Objekte erfolgen. Die genannten hierarchischen Elemente können durch weitere teils obligatorische, teils optionale Bausteine modifiziert werden. Eine genauere Beschreibung folgt im nächsten Abschnitt.

Der Versicherungsvertrag enthält Information über das Abkommen zwischen Versicherungsnehmer und Versicherungsunternehmen, d.h. bis zu welcher Höhe Leistungen erbracht werden, wie lange der Vertrag dauert, ob er sich automatisch verlängert, welcher Art der Selbstbehalt ist. Es wird Information darüber gespeichert, wie verschiedene Geschäftsvorfälle mit dem Vertrag und seinen Elementen verfahren sollen, d.h. ob automatisch gemahnt wird, ob ein `ElementarRisiko` mit auf das Vertragsdokument zu drucken ist, ob für gewisse Änderungen Genehmigungen einzuholen sind. Die genannten Beispiele zeigen, dass neben den relevanten Daten für den Geschäftsprozess, der in Kapitel 6.2 beschrieben wird, mindestens ebenso viele weitere Daten verwaltet werden.

Ausgehend vom Produktmodell der VAA ist das Analysemodell (siehe: Kapitel 6.1) für das vorliegende Buch erstellt worden. Da Verträge fachlich auf Produkten basieren, lässt sich aus der Komponente Produkt [68] in kritischer Auseinandersetzung ableiten, wie mögliche Vertragsstrukturen gestaltet sind. So wird beispielsweise in einem Versicherungsprodukt definiert, welche Elementarrisiken zu einem Risikobündel zusammenfassbar sind. Im erstellten Analysemodell spiegelt sich die hierachische Struktur von der Komponente »Produkt« der VAA wider. Das dargestellte Analysemodell (Abbildung 6.1) ist detailliert auf der beiliegenden CD im HTML-Format dokumentiert. Dort sind insbesondere die Attribute beschrieben, die aus Gründen der Übersichtlichkeit in Abbildung 6.1 ausgeblendet sind.

Das dargestellte Modell dient als gemeinsame Basis der noch vorzustellenden Implementierungsvarianten. Daher wird im Folgenden ein Überblick gegeben. Dargestellt sind die einzelnen, inhaltlich abgrenzbaren Bestandteile des Geschäftsobjekts »Versicherungsvertrag«. Ein solches Geschäftsobjekt wird mit konkreten Werten versehen und gespeichert, um einem computergestützten Verwaltungsprogramm bekanntzugeben, dass ein neuer

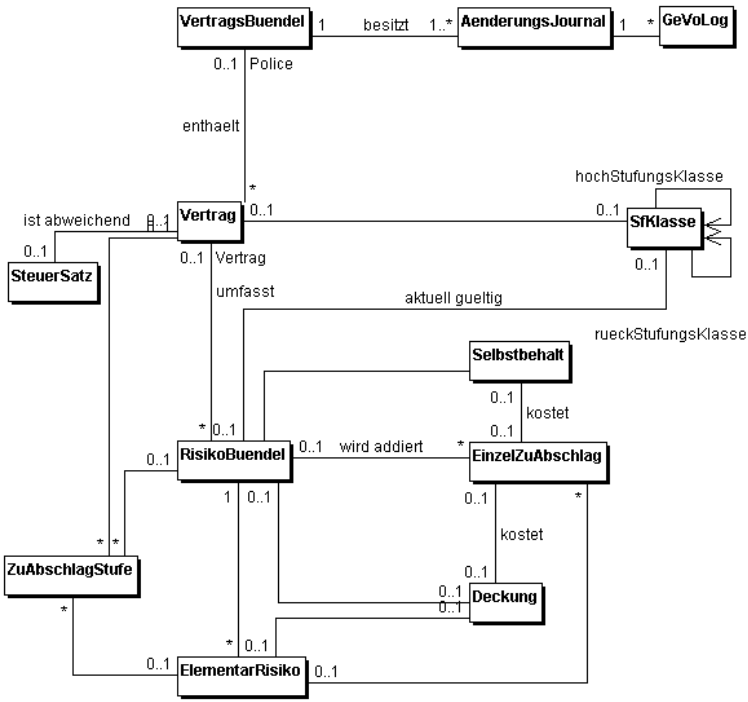


Abbildung 6.1: Klassendiagramm des Geschäftsobjekts

Versicherungsvertrag mit einem Partner geschlossen wurde. Im Einzelnen haben die in Abbildung 6.1 dargestellten Elemente die folgende Bedeutung:

- **`VertragsBuendel`** stellt eine Klammer dar, um mehrere Verträge mit einem Versicherungsnehmer gemeinsam verwalten zu können. Wenn also ein Versicherungsnehmer bei einer Gesellschaft eine KFZ-, eine Hausrat- und eine Rechtsschutzversicherung abgeschlossen hat, so soll es über das `VertragsBuendel` möglich sein, alle Verträge gemeinsam abzurechnen, sodass eine anstelle dreier Rechnungen an den Versicherungsnehmer geht. Dieses Konzept wird oft als Multilineproduct bezeichnet.

- **`AenderungsJournal`** beinhaltet Funktionalität, mit der nachgehalten werden kann, wer wann warum Änderungen an dem zugehörigen `VertragsBuendel` vorgenommen hat. Die eigentliche Information wird im `GeVoLog` (siehe dort) abgespeichert.

- **`GeVoLog`** steht abkürzend für »Geschäftsvorfalllog«. Geschäftsprozesse sind zielgerichtete, fachlich relevante Bearbeitungen von Geschäftsobjekten in Einklang mit den Unternehmenszielen. Jede dieser Bearbeitungen an einem Versicherungsvertrag wird mitprotokolliert. Die einzelnen Protokolleinträge zu einem Geschäftsprozess werden dann per `GeVoLog` abgespeichert.

- **`Vertrag`** fasst eine Struktur zusammen, die dem entspricht, »was man unterzeichnet, wenn man eine KFZ-Versicherung abschließt«. Das heißt ein Vertrag gruppiert

die großen Blöcke Haftpflicht, Unfallversicherung und Kaskoversicherung zu einer Einheit und stellt den Bezug zu dem versicherten Fahrzeug für alle drei Teile her. Weiterhin beinhaltet er Information über Vertragsbeginn und Ende, über die Art der Verlängerung, über den Umgang mit Mahnungen und die Prämiensumme der untergeordneten Bausteine. Da das Beispiel dieses Buchs eine KFZ-Versicherung ist, wird diese Gesamtprämie durch eine Rabattstaffel beeinflusst, die abhängig ist von der Anzahl schadensfreier Jahre (siehe: `SfKlasse`).

- **`SfKlasse`** beinhaltet Information darüber, welcher Schadensfreiheitsrabatt gegenwärtig auf einen Vertrag oder ein `RisikoBuendel` anwendbar ist.
- **`SteuerSatz`** enthält als Wert den aktuell gültigen Steuersatz eines Vertrags. Häufig haben Finanzdienstleister so genannte »Schlüsselverzeichnisse«, in denen zentrale Informationen wie der Steuersatz, Landeskennzeichen, Währungskennzeichen usw. gepflegt werden, um alle Systeme gleichzeitig aktuell halten zu können. Daraus resultiert die Anforderung an neue Systeme, solche Schlüsselverzeichnisse geeignet zu integrieren. Der Steuersatz ist ein Wrapper um diesen Wert. Der einfachste Weg, dies zu realisieren, ist eine einzelne Bean, die von dem Vertrag referenziert wird.
- **`RisikoBuendel`** gruppiert Elementarrisiken zu einer verkaufbaren Einheit. So kann als Kaskoversicherung z.B. Schaden durch Haarwild, Diebstahl und Brand des Fahrzeugs gebündelt werden. Das Bündel kann dadurch als Einheit modifiziert werden. Ein `Selbstbehalt` senkt die Prämie durch einen `EinzelZuAbschlag`. EinzelZuAbschläge können dem Sachbearbeiter Spielraum geben, die Prämie zu variieren. `ZuAbschlagStufen` können als tariflich zugrunde gelegte Rabattstaffeln ebenfalls die Prämie verändern. Die einem `RisikoBuendel` assoziierte `Deckung` kann als Dreiheit aus Personen-, Sach-, und Vermögensdeckung beschreiben, bis zu welchem Betrag ein Schaden durch die Versicherung in dieser Schadensart getragen wird.
- **`Selbstbehalt`** beinhaltet Information, bis zu welchem Betrag der Versicherungsnehmer selbst einen entstandenen Schaden trägt.
- **`EinzelZuAbschlag`** modifiziert die Prämie eines `RisikoBuendel` oder `ElementarRisiko`. Solche Modifikationen werden als tarifunabhängige Einzelregelung ausgehandelt.
- **`Deckung`** beinhaltet Information, bis zu welcher Schadenshöhe für Personen-, Sach- und Vermögensschäden die Versicherung die Haftung übernimmt. Diese Versicherungssummen gelten für das `RisikoBuendel` oder das `ElementarRisiko`, dem sie assoziiert sind.
- **`ElementarRisiko`** beinhaltet die Information über die Gefahr, die Gegenstand der Versicherung ist. Abhängig von gewissen Kenngrößen wird über einen Tarif ermittelt, welche Prämie für die konkrete Ausprägung des Risikos zu zahlen ist. Es kann ähnlich wie ein `RisikoBuendel` modifiziert werden, indem Bausteine wie `EinzelZuAbschlag` assoziiert werden.
- **`ZuAbschlagStufe`** entspricht einer Stufe innerhalb einer Rabattstaffel, die Anwendung findet, da sie produktseitig definiert ist. Abhängig von der Einheit, die die Rabattstaffel definiert, kann sich die Prämie eines Vertrags, der Beitrag eines `ElementarRisiko`

oder `RisikoBuendel` über die Vertragslaufzeit verändern. Gängige Rabattstaffeln richten sich beispielsweise nach Vertragslaufzeit oder Beitragshöhe.

Die genannten Elemente stellen das Ergebnis einer objektorientierten Analyse dar. Eine konkrete Instanz des Geschäftsobjekts ist in Kapitel 7.1 dargestellt und erläutert. Die Elemente ergeben in ihrem Verbund eine *fachliche Komponente* nach der Begriffsprägung in Abschnitt 3.2 (vgl. dazu [30, S. 19]). Im Einzelnen sind es die folgenden Eigenschaften, die dort für fachliche Komponenten vorgestellt werden:

- **Kapselung ganzer Geschäftsobjekte und deren Geschäftslogik:** Der Verbund von Objekten enthält als abgeschlossene *Einheit* die Funktionalität, die ein Versicherungsvertrag fachlich haben muss. Das sind u.a. Neugeschäft, Stornierung, Abrechnung und Auskunftsfunktionalität. Die Funktionalität wird über Schnittstellen angeboten. Der Nutzer einer Komponente muss kein Wissen über die interne Struktur haben.
- **Implementierung ganzer Transaktionen oder großer Teile des unterstützten Geschäftes:** Die Geschäftslogik ist möglichst vollständig in der Komponente enthalten. Dadurch wird die Austauschbarkeit der Komponente erhöht.
- **Anwendungsabhängigkeit:** Durch die ersten beiden Eigenschaften bedingt, kann die Komponente nicht in beliebigen anderen Kontexten eingesetzt werden. Sie ist nur innerhalb der Anwendungsdomäne wiederverwendbar.
- **Ablauffähig als Einzelnes:** Die Ausführung der Geschäftslogik ist von anderen Komponenten unabhängig. In manchen Fällen wird auf festgelegte Dienste anderer Komponenten zurückgegriffen.

Der Vollständigkeit halber wird hier kurz beschrieben, welche weiteren Komponenten mit dem beschriebenen Geschäftsobjekt zusammenspielen müssen, damit die Kernfunktionalität einer Bestandsführung gegeben ist. Diese Elemente sind mithilfe eines Komponentendiagramms in Abbildung 6.2 dargestellt. »Komponente« ist in diesem Zusammenhang sehr weit zu fassen und als ein Subsystem zu verstehen, das anderen Subsystemen eine gewisse Aufgabe abnimmt. Welche Aufgabe das ist, ist jeweils an den Verbindungslinien vermerkt.

Eine Partnerverwaltung (siehe Komponente »Partner«) dient der Speicherung von Adressdaten, Bankverbindungen und Ansprechpartnern, die im Kontext einer Bestandsführung vorkommen. Die Daten zu einem Versicherungsnehmer (VN) werden also nicht im `VertragsBuendel` abgelegt, sondern als eindeutiger Verweis in der Partnerverwaltung gespeichert. Ebenso können dort Geschädigte eines Schadenfalls oder Ansprechpartner anderer Versicherungen vermerkt werden.

Ein InkassoExkasso-System dient der Überwachung der Zahlungsflüsse. Hier wird kontrolliert, welche Zahlungen zu welchen Prämienforderungen eingegangen sind und welche Leistungen ausbezahlt wurden und noch auszuzahlen sind.

Ein Druck-System dient dem Ausstellen von Papierdokumenten, die zur Information, als Zahlungsaufforderung oder als schriftlicher Vertrag dem Versicherungsnehmer zugestellt werden.

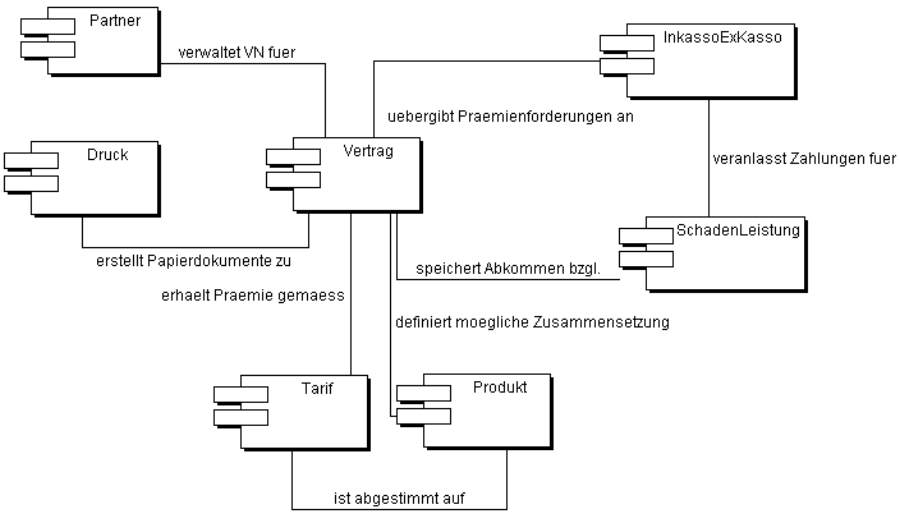


Abbildung 6.2: Komponentendiagramm: Umfeld des Geschäftsobjekts

Der Vertrag speichert und verwaltet das Abkommen zwischen Versicherungsunternehmen und Versicherungsnehmer, welche Leistungen beim Eintritt welches Risikos zu erbringen sind und welche Prämie dafür vom Versicherungsnehmer zu entrichten ist. Allein diese Komponente wird in dem vorliegenden Buch umgesetzt, so weit es das Beispiel erfordert.

Das SchadenLeistung-System dient dazu, gemeldete Schäden zu speichern, und veranlasst das Erbringen einer Leistung im Schadensfall.

Eine Tarif-Komponente dient dazu, das tabellarische Muster abzubilden, durch das einem `ElementarRisiko` eine Prämie zugeordnet wird. Allgemein orientiert sich die Prämie eines Risikos an der Eintrittswahrscheinlichkeit des Risikos und dem geschätzten Schaden, der verursacht wird. Diese beiden Größen werden anhand von signifikanten Merkmalen festgemacht, so genannten Risikomerkmalen. Die Ausprägung eines Risikomerkmals wird bei Vertragsabschluss in den `ElementarRisiko`-Bausteinen vermerkt. Eine Tariftabelle ordnet der Ausprägung von Risikomerkmalen des Vertrags schließlich eine jeweils bestimmte Prämie zu.

Eine Produkt wird benutzt, um zu definieren, wie ein Geschäftsobjekt zusammengesetzt werden darf. Im Detail wird hier festgelegt, welche ElementarRisiken zu welchem `RisikoBuendel` gehören. Es wird vordefiniert, welche ZuAbschläge anwendbar sind, welche Tarife gelten usw. Das ist der Ermessensspielraum, den der Sachbearbeiter beim Abschluss eines Vertrags hat.

Die Abbildung 6.2 lässt sich um andere Elemente, wie etwa »Provision« und »Statistik« erweitern; als Skizze einer minimalen Funktionalität reichen jedoch die dargestellten.

6.2 Der Geschäftsprozess

Damit der dargestellte Geschäftsprozess realistisch und umfangreich genug ist, um die Auswirkung verschiedener Implementierungsvarianten zeigen zu können, muss er verschiedene Auswahlkriterien erfüllen.

1. **Traversieren der Struktur:** Der Geschäftsprozess muss auf Ebene des Analysemodells die Struktur des Geschäftsobjekts traversieren. Das heißt die relevante Information muss aus verschiedenen Teilobjekten zusammengesucht werden. Dadurch lassen sich Varianten, bei denen die Traversierung auf Ebene der Datenbank stattfindet, mit Varianten vergleichen, die Beans zum Traversieren benutzen.

2. **Verschieden mächtige Treffermengen:** Die Anfragen an die Datenbank müssen verschieden große Ergebnismengen liefern, d.h. einmal nur genau ein Objekt, einmal eine große Anzahl von Objekten. Suchanfragen an die Datenbank mit kleiner Treffermenge werden technisch anders behandelt als Suchanfragen mit großer Treffermenge. Werden beide Arten von Anfragen benutzt, so sind für beide Optimierungen ermittelbar.

3. **Lesende und schreibende Transaktionen:** Es müssen sowohl lesende als auch schreibende Transaktionen vorkommen, da in produktiven Systemen auch lesende und schreibende Transaktionen vorkommen.

4. **Unabhängige Teilschritte:** Berechungsergebnisse eines Teilschritts haben keinen Einfluss auf die Berechung der anderen Teilschritte. Damit sind diese Teilschritte parallelisierbar. Es leuchtet unmittelbar ein, dass nur dann Konzepte zur Parallelisierung auf EJB-Architekturen umgesetzt und in ihrer Wirksamkeit geprüft werden können, wenn der fachliche Prozess dies zulässt.

Um all diesen Kriterien zu genügen, ist als umzusetzender Geschäftsprozess die automatische Veränderung des Schadensfreiheitsrabatts gewählt worden.

Jedem KFZ-Vertrag liegt eine Rabattstaffel zugrunde, die abhängig von der Anzahl der schadensfreien Jahre eine Vergünstigung der Prämie bewirkt. So befindet sich in Schadensfreiheitsklasse1 (SF1) mit einem Beitrag von 100 %, wer ein Jahr Beiträge zahlt, ohne die Versicherung in Anspruch zu nehmen. Bei zwei Jahren zahlt man 85 %, bei drei Jahren 75 % usw. Wer Jahr für Jahr Prämien zahlt, ohne die Versicherung in Anspruch zu nehmen, erhält automatisch eine vergünstigte Prämie. Diese automatische Vergünstigung der Prämie ist Aufgabe des Batchlaufs. Im Folgenden wird dafür der Begriff SfrUmstufung benutzt.

Wie dieser Vorgang im Detail aussehen kann, ist im Aktivitätsdiagramm (siehe Abbildung 6.3) dargestellt. Folgende Schritte sind notwendig:

1. **Ermittle hauptfällige Verträge:** Es werden zuerst alle Verträge identifiziert, deren Hauptfälligkeit erreicht ist. Ein Vertrag ist hauptfällig, wenn seit seinem Abschluss oder seiner letzten Hauptfälligkeit eine Versicherungsperiode, das ist meist ein Jahr, vergangen ist. Soll der Vertrag automatisch verlängert werden oder ist er über mehrere Jahre abgeschlossen, so ist zu prüfen, ob dem Versicherungsnehmer für die

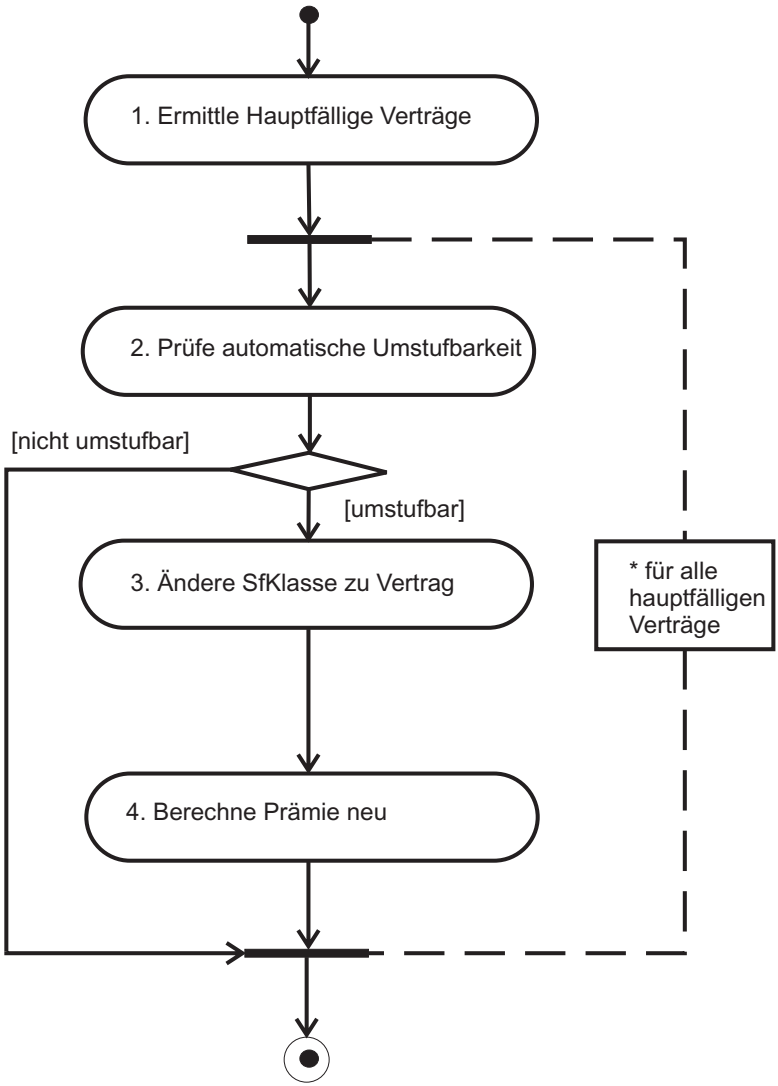


Abbildung 6.3: Aktivitätsdiagramm: Sfr-Umstufung

nächste Versicherungsperiode gemäß des Schadensfreiheitsrabatts eine günstigere Prämie zusteht. Für alle hauptfälligen Verträge werden die Aktivitäten innerhalb der Synchronisationsbalken ausgeführt (siehe Notiz an Synchronisationsbalken).

2. **Prüfe automatische Umstufbarkeit:** Prüft pro Vertrag, ob die letzte SfrUmstufung mindestens eine Versicherungsperiode zurückliegt, der automatische Wechsel in eine höhere SfKlasse für diesen Vertrag vorgesehen und die maximale `SfKlasse` noch nicht erreicht ist. Wird eine dieser Fragen negativ beantwortet, so wird der Vertrag nicht umgestuft (linker Zweig hinter der Entscheidungsraute »[nicht umstufbar]«). Ansonsten folgt Schritt 3.

3. **Ändere SfKlasse zu Vertrag:** Werden alle Fragen positiv beantwortet, so wird dem Vertrag die nächsthöhere, also für den Versicherungsnehmer günstigere, Rabattstufe (SfKlasse) zugewiesen.

4. **Berechne Prämie neu:** Auf der neuen SfKlasse basierend wird die Prämie des Vertrags neu berechnet.

Die Neuberechnung der Prämie lässt sich noch weiter verfeinern. Die Unteraktivitäten dieser Aktivität sind für einen einzelnen Vertrag in Abbildung 6.4 dargestellt. Die drei senkrecht voneinander abgetrennten Bereiche für `Vertrag`, `RisikoBuendel` und `ElementarRisiko` markieren die Zuständigkeiten für die Aktivitäten.

Durch die Synchronisationsbalken im linken und mittleren Bereich werden Schleifen dargestellt. Die Schleife im linken Bereich iteriert über alle aktuellen `RisikoBuendel`, die zu einem Vertrag gehören. Die Schleife im mittleren Bereich iteriert über alle aktuellen ElementarRisiken, die zu einem RisikoBuendel gehören. Bei der Prämienberechnung werden auf jeder Ebene des Vertrags (`Vertrag`, `RisikoBuendel`, `ElementarRisiko`) Mengen von Bausteinen zusammengestellt, die die Prämie modifizieren. Modifizierend wirken die Bausteine `SfKlasse`, `ZuAbschlagStufe` und `EinzelZuAbschlag`. Für sie wird im Folgenden der Oberbegriff Modifikatoren benutzt. Auf Ebene des `ElementarBausteins` wird ein Beitrag und ein maximaler Mindestbeitrag aus dem Tarif ermittelt. Die modifizierenden Bausteine verändern diese aus dem Tarif ermittelten Werte. Die Aktivitäten führen im Detail folgende Aufgaben aus:

4.1 **Historisiere Vertrag:** Vor der Veränderung des Vertrags wird der gegenwärtige Stand abgespeichert und vermerkt, wann er ersetzt wurde.

4.2 **Ermittle Rabatt der SfKlasse:** Die Auswirkung der neuen `SfKlasse` auf die Prämie wird als Faktor bestimmt.

4.3 **Ermittle ZuAbschlagStufen:** Alle dem `Vertrag` assoziierten `ZuAbschlagStufen` werden zusammengestellt. Diese Zusammenstellung wird später an die `Elementar–Risiken` übermittelt und dort angewendet.

4.4 **Historisiere RisikoBuendel:** Für jedes `RisikoBuendel` wird sein bisheriger Stand abgespeichert und vermerkt, wann er ersetzt wurde.

4.5 **Ermittle Rabatt der SfKlasse:** Soweit dem `RisikoBuendel` eine `SfKlasse` assoziiert ist, wird ihr Einfluss auf den Beitrag dieses `RisikoBuendels` als Faktor ermittelt.

4.6 **Ermittle ZuAbschlagStufen:** Alle dem `RisikoBuendel` assoziierten `ZuAbschlag–Stufen` werden zusammengestellt, um später an die `ElementarRisiken` übermittelt zu werden. Das sind beispielsweise Stufen einer Rabattstaffel, die sich an der Vertragslaufzeit orientieren.

4.7 **Ermittle EinzelZuAbschlaege:** Alle dem `RisikoBuendel` assoziierten `EinzelZuAb–schlaege` werden zusammengestellt, um später an die `ElementarRisiken` übermittelt zu werden.

4.8 **Historisiere ElementarRisiko:**Für jedes `ElementarRisiko` wird sein bisheriger Stand abgespeichert und vermerkt, wann er ersetzt wurde.

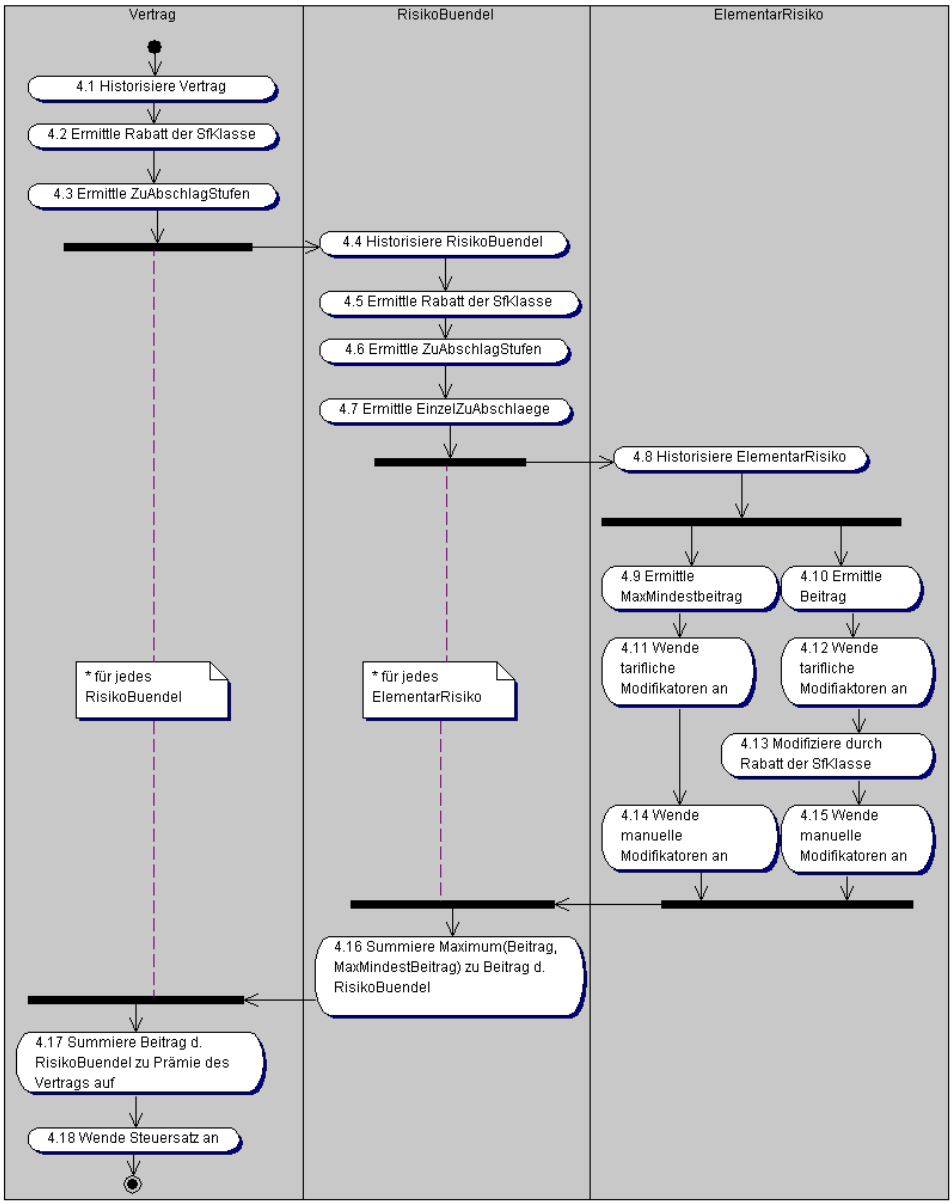


Abbildung 6.4: Aktivitätsdiagramm: Verfeinerung der Aktivität »Berechne Prämie neu«

4.9 **Ermittle MaxMindestBeitrag:** Der Tarif berechnet zu dem `ElementarRisiko` einen maximalen Mindestbeitrag. Darauf werden später alle Modifikatoren angewendet.

4.10 **Ermittle Beitrag:** Bei der Ermittlung des Beitrags wird stets so vorgegangen, dass ein Grundwert aus dem Tarif bestimmt wird. Der Grundwert des `ElementarRisikos`

wird abhängig von Risikomerkmalen mithilfe des Tarifs bestimmt. Er wird dann durch tariflich zugesicherte Zu- und Abschläge verändert. Diese Zu- und Abschläge werden dem `Vertrag`, dem `RisikoBuendel` oder dem `ElementarRisiko` assoziiert. Sie wirken sich stets auf alle darunter liegenden Teile des Geschäftsobjekts aus.

4.11 **Wende tarifliche Modifikatoren an:** Auf den maximalen Mindestbeitrag werden alle tariflich vereinbarten Modifikatoren angewendet. Da sich Modifikatoren sowohl absolut als auch prozentual auswirken können, ist die Reihenfolge der Anwendung relevant.

4.12 **Wende tarifliche Modifikatoren an:** Auf den Beitrag werden alle tariflich vereinbarten Modifikatoren angewendet.

4.13 **Modifiziere durch Rabatt der SfKlasse:** Die Rabatte der SfKlassen wirken sich nur auf den Beitrag aus.

4.14 **Wende manuelle Modifikatoren an:** Vergleiche 4.11. In diesem Fall werden manuelle, d.h. durch den Sachbearbeiter vereinbarte, Modifikatoren angewendet. Tariflich zugesicherte Modifikatoren werden stets vor manuell erstellten Modifikatoren angewendet.

4.15 **Wende manuelle Modifikatoren an:** Vergleiche 4.14.

4.16 **Summiere Maximum(Beitrag, MaxMindestBeitrag) zu Beitrag d. RisikoBuendel:** Nachdem alle Modifikatoren angewendet worden sind, wird von jedem `ElementarRisiko` entweder der Beitrag oder der maximale Mindestbeitrag zum Beitrag des RisikoBuendels aufsummiert, je nachdem, welcher Wert größer ist.

4.17 **Summiere Beitrag d. RisikoBuendel zu Prämie des Vertrags auf:** Die Beiträge der `RisikoBuendel` werden zur Prämie des Vertrags aufsummiert.

4.18 **Wende Steuersatz an:** Auf die Prämie wird der aktuell gültige Steuersatz angewendet.

Dieser Geschäftsprozess genügt den oben genannten Kriterien. Das Geschäftsobjekt wird traversiert, da bei der Sfr-Umstufung durch Lesen von `Vertrag` und `SFKlasse` die automatische Umstufbarkeit ermittelt wird. Danach werden `ElementarRisiko`, `EinzelZuAbschlag`, `ZuAbschlagStufe` und `SFKlasse` gelesen, um den neuen Beitrag zu bestimmen. Es wird wegen des veränderten Beitrags ein neuer Stand von `ElementarRisiko` geschrieben. Schließlich werden alle `ElementarRisiko`-Instanzen und der Steuersatz zu einem Vertrag gelesen, um den Vertragsbeitrag zu ermitteln. Der neue Vertragsbeitrag wird durch Schreiben eines neuen Standes von VersicherungsVertrag gespeichert.

Treffermengen zu verschiedenen Anfragen haben verschiedene Mächtigkeit. Die Ermittlung der umstufbaren Verträge produziert in aller Regel eine große Treffermenge. Wird unterstellt, dass allen Versicherungsverträgen Schadensfreiheitrabatte gewährt werden und die Hauptfälligkeitstermine über ein Jahr gleichverteilt sind, so nimmt ein Zwölftel des Bestands an einem monatlichen Sfr-Umstufungsbatch teil. Die Ermittlung aller aktuellen ElementarRisiken zu einem Vertrag bringt dagegen nur eine kleine Treffermenge; weniger als 10 Instanzen. Die Ermittlung der aktuellen Sfr-Stufe schließlich

bringt keinen oder einen Treffer. Der Batchlauf lässt sich parallelisieren, da der Beitrag eines Vertrags unabhängig vom Beitrag eines anderen Vertrags ist. Dass sowohl lesende als auch schreibende Zugriffe auf die Daten stattfinden, wurde bereits im Zusammenhang mit dem Traversieren der Struktur des Geschäftsobjekts dargestellt.

Die zu Anfang des Kapitels an den Geschäftsprozess gestellten Kriterien sind erfüllt. Daher lässt sich an ihm zeigen, ob eine Implementierung die Elemente der EJB-Architektur geeignet ausnutzt. Der Geschäftsprozess ist also als Beipiel für dieses Buch tauglich.

6.3 Abgrenzung von einem produktiven System

Das vorliegende Buch ist allein an dem *performanten* Ablauf des Beispielprozesses interessiert. Die sich daraus ergebenden Einschränkungen des Beispiels werden im Folgenden genannt. Zuerst werden infrastrukturelle Einschränkungen aufgezählt, d.h. solche, die angrenzende Systeme oder das allgemeine Rahmenwerk, wie es in Kapitel 7.3 detailliert wird, betreffen. Danach werden die Einschränkungen der Umsetzung des Geschäftsprozesses beschrieben.

Das Rahmenwerk , das für verschiedene Batchläufe Infrastruktur (Fehlerbehandlung, Logging) bietet, wird nur soweit wie nötig implementiert. Zu einem produktiv einsetzbaren Rahmenwerk fehlen damit die folgenden Anforderungen.

- **24×7 Betrieb:** Es gibt vielerorts Bestrebungen, dem Kunden Geschäftsprozesse zu eröffnen. Es muss daher möglich sein, jederzeit auf die Datenbestände zuzugreifen. Diese hohe Erreichbarkeit wird auch als 24×7 Betrieb bezeichnet: »24 Stunden pro Tag an sieben Tagen pro Woche erreichbar«. Üblich war bisher, Batchläufe bei Nacht oder am Wochenende laufen zu lassen, um so die volle Leistung des Systems zu haben. Alle Schnittstellen des Systems für den Online-Betrieb sind während dieser Zeit abgeschaltet worden. Hinter dem 24×7 Betrieb steckt also die Anforderung, Online- und Batch-Betrieb parallel zu betreiben.
- **Belieferung von Randsystemen per Batchschnittstelle:** Wie in Abbildung 6.2 dargestellt, unterhält das Geschäftsobjekt Schnittstellen zu anderen Subsystemen wie Partner oder InkassoExkasso. Der Umgang mit Randsystemschnittstellen wird in der Praxis oft auf eine der beiden folgenden Arten gelöst:

  1. **Organisatorisch:** Teure Zugriffe auf entfernte Rechnersysteme werden organisatorisch umgangen, indem in einem vorbereitenden Schritt alle notwendige Information aus dem entfernten System temporär in das eigene geladen wird. Dadurch sind manche Datensätze nicht korrekt. Beispielsweise kann Adressinformation aus dem Partnersystem stets morgens vor einem Inkassolauf in die Bestandsführung kopiert werden. Aus einem Zugriff auf ein räumlich entferntes System wird dann ein Zugriff auf eine zusätzliche Tabelle. Er kann wiederum nach Design-Regeln dieses Buchs optimiert werden. Sollte nach dem Anlegen der Kopie eine Adressänderung im Partnersystem eingehen, so wird sie beim Inkassobatch nicht berücksichtigt; es wird darauf gehofft, dass ein Nachsendeantrag gestellt wurde.

2. **Batchschnittstelle:** Randsysteme werden oft per Batchschnittstelle beliefert, d. h. die Information zu allen bearbeiteten Geschäftsobjekten wird in einem einzigen Zugriff auf die Schnittstelle übergeben. Nicht für jeden Vertrag erfolgt ein Zugriff auf das Drucksystem, um eine Information zur veränderten Prämie zu verschicken. Anstelle dessen wird die Druckinformation gesammelt und für alle Verträge zusammen übergeben. So werden die Kosten gespart, eine Verbindung zum Drucksystem mehrfach zu öffen und wieder zu beenden.

Der Umgang mit Schnittstellen zu entfernten Systemen ist ein allgemeines Problem bei verteilten Anwendungen, das hier nicht weiter erörtert werden muss. Dadurch wird nicht in Abrede gestellt, dass die Belieferung von Randsystemen zu den wichtigsten Funktionen gehört, die ein produktives System haben muss.

- **Ergebnisübersicht:** Nach Abschluss des Batchlaufs muss nachvollziehbar sein, bei welchen Geschäftsobjekten der Batchlauf gescheitert ist und warum. Die Fehlerquote muss möglichst gering sein, da eine manuelle Nachbearbeitung sehr zeitintensiv ist. Ein Logging der erfolgreichen Bearbeitungen hat ebenfalls zu erfolgen. Gemessen an der Gesamtdauer eines Batchlaufs hat das Logging nur einen sehr geringen und wenig beeinflussbaren Anteil und wird daher nicht behandelt. Denkbar wäre der Einsatz eines XML-basierten Logging. Diese Log-Dateien könnten automatisch weiterverarbeitet werden zu Arbeitsaufträgen an zuständige Sachbearbeiter.

- **Rechenzentrumsbetrieb:** Der Batchlauf muss automatisch zu vorbestimmten Terminen oder Intervallen starten. Ein fehlerhaftes Beenden darf nicht den restlichen Betrieb blockieren.

- **Restartfähigkeit:** Bricht der Batchlauf ab oder muss er gestoppt werden, so ist bei einem Wiederanlaufen sicherzustellen, dass kein Geschäftsobjekt doppelt bearbeitet oder ausgelassen wird.

- **Verkettung von Batchläufen:** Was zuvor als Batchlauf zum Rechnungsdruck beschrieben wurde, ist bei genauerem Hinsehen eine Verkettung mehrerer Batchläufe. So wird beispielsweise im ersten Lauf ermittelt, welche Information der Geschäftsobjekte an die Druckschnittstelle zu liefern ist und im zweiten Lauf im Drucksystem zu Druckinformation umgewandelt, um im dritten Lauf als Stapel dieser druckbaren Dateien durch eine Druckstraße verarbeitet zu werden. Die gleiche Menge einmalig zu Anfang identifizierter Geschäftsobjekte wird stufenweise durch mehrere Batches bearbeitet. Es muss möglich sein, mit genau einer Übergabe ein Zwischenergebnis aller Geschäftsobjekte von einem Batchlauf zum nächsten zu reichen.

Die genannten Anforderungen beeinflussen nur mittelbar das Design der Anwendung. Daher werden sie nicht weiter betrachtet.

Wird der gewählte Geschäftsvorfall selbst betrachtet, zeigen sich die folgenden Abweichungen zu einem produktiven System:

- **Vereinfachte Berechnungen:** Die in den Implementierungen vorgenommenen Berechnungen entsprechen im Detail nicht den Berechnungen eines produktiven Systems. Es werden beispielsweise die rechtlichen Bestimmungen nicht beachtet, wie

im Verlauf einer Prämienberechnung zu runden ist. Diese Vereinfachung ist unter der Annahme vorgenommen worden, dass die Rechenzeit, die eine Berechnung kostet, zu vernachlässigen ist gegenüber der Zugriffszeit auf benötigte Daten in der Datenbank. Die Praxisberichte [70] und [1] bestätigen die Richtigkeit dieser Annahme. Damit sind die Ergebnisse dieses Buchs nur begrenzt übertragbar auf Systeme, die mit wenigen Daten performanzkritische Berechnungen durchführen.

- **Historisierung:** Eine Anforderung an produktive Bestandsführungssysteme ist vereinfachte Historisierung. Historisierung ist die Fähigkeit eines Systems, Entwicklung von Sachen, Personen oder Vorgängen über die Zeit festhalten zu können. Für die Versicherungsverträge des Bestands bedeutet das, dass jede Änderung rekonstruierbar sein muss. Es reicht nicht aus, nur geänderte Attribute zu speichern, wenn z.B. ein neuer Vertragsbeitrag nach SfrUmstufung ermittelt wurde. Der alte Vertragsstand muss konserviert werden und dazu die Information, von wann bis wann er gültig war und wann dem System die Änderung bekannt gegeben wurde. Ein neuer Stand muss gespeichert werden. Es muss jederzeit genau einen aktuell gültigen Stand geben.

  Eine mögliche Umsetzung der Historisierung von Versicherungsverträgen mit Enterprise JavaBeans ist detailliert in [33] nachzulesen. In dieser Umsetzung wird vorgeschlagen, durch vier Zeitattribute pro historisierbarem Element den Gültigkeitszeitraum und den Zeitraum der Erfassung im System abzubilden. Verändert sich beispielsweise in einem `ElementarRisiko` ein Risikomerkmal, so werden in dem bestehenden `ElementarRisiko` die Zeiträume der Erfassung und der Gültigkeit angepasst und eine Kopie des `ElementarRisiko` mit dem neuen Risikomerkmal angelegt. Diese Kopie ist an den Zeitattributen als die aktuell gültige erkennbar. Die Referenz auf ein `ElementarRisiko` wird durch einen fachlich eindeutigen Schlüssel hergestellt, der bei allen Historienständen gleich ist. Um einen gewissen Stand aufzufinden, muss also neben diesem fachlichen Schlüssel noch eine Wertbelegung der Zeitattribute angegeben werden.

  Auf die komplette Umsetzung einer Historisierung wird verzichtet. Die für die Historisierung notwendigen Funktionen werden vereinfacht zu Funktionen, die das gleiche Datenvolumen zu bearbeiten haben und die gleiche Anzahl an Lese- und Schreiboperationen in der Datenbank vornehmen, wobei der konkret geschriebene oder gelesene Wert unerheblich ist. Bespielsweise wird anstelle des Anlegens eines neuen Standes nur eine Kopie des bestehenden Standes in die Datenbank geschrieben, wobei alle historisierbaren Elemente die genannten Zeitattribute und den fachlichen Schlüssel mitführen. Dadurch wird eine Transaktion gleichen Volumens ausgeführt, sodass der Einfluss der Historisierung auf die Performanz weiterhin gegeben ist.

  Für das vorliegende Buch ist Historisierung eine Möglichkeit, neben lesenden Transaktionen auch einen hohen Anteil an schreibenden Transaktionen in den Beispielgeschäftsprozess zu integrieren. Die gewählte Art der Historisierung hat sich in der Praxis als wenig performant erwiesen, sodass sich für ein konkretes Projekt eine andere Art der Historisierung empfiehlt. Für das vorliegende Buch geht es nicht um performante *Historisierung*. Die Anschaulichkeit dieser Umsetzung steht im Vordergrund.

- **SfrUmstufung als einzelner Batchlauf:** Für das vorliegende Buch ist eine SfrUmstufung ein eigener Batchlauf. Ein anschließender Batchlauf könnte dann eine Abrechnung vornehmen bis zum Zeitpunkt der Umstufung. Dieses Vorgehen ist denkbar. In den meisten produktiven Systemen wird die Umstufung aber eher als Teil der Abrechnung ausgeführt, da diese Schritte fachlich eng zueinander gehören. Diese *fachliche* Einschränkung, SfrUmstufung sei eher ein Teil der Abrechnung als ein eigenständiger Vorgang, beeinträchtigt jedoch nicht die Übertragbarkeit der *technischen* Ergebnisse dieses Buchs. [28] argumentiert, dass es bei Batchläufen, die nicht innerhalb der zur Verfügung stehenden Zeit beendet sind, sogar notwendig ist, sie in kleinere Batchläufe an mehreren Terminen zu spalten.

7 Messung der EJB-Anwendung

Nachdem das vorangegangene Kapitel das Anwendungsbeispiel dargelegt hat, wird in diesem Kapitel das Vorgehen zur Ermittlung der Design-Regeln beschrieben.

Eine einfache Basisvariante des Geschäftsprozesses setzt durch ihre processing time (siehe Abschnitt 5.1) eine Referenzmarke. Jede folgende Variante diskutiert eine Möglichkeit zur Verminderung der processing time. Die tatsächlich eingetretene Verminderung wird durch eine Messung und den Vergleich zur oben genannten Referenzzeit ermittelt. Durch die eingetretene Verminderung kann die diskutierte Möglichkeit als Design-Regel für verbesserte Performanz validiert werden.

Für alle Varianten gelten folgende Grundsätze:

- **Verwendung proprietärer Ergänzungen des EJB 2.0-Standards vermeiden:** Es sollen Möglichkeiten des EJB 2.0-Standards sinnvoll ausgeschöpft werden. Insbesondere sollen selbst entwickelte Erweiterungen des Standards vermieden werden. Das sind u.a. eigene Transaktionskonzepte, eigene Caching-Konzepte, eigene Architekturerweiterungen usw.
- **Wiederverwendbarkeit und Modularität:** Aus der Perspektive des einzelnen Geschäftsobjekts ist es unwesentlich, ob es durch einen manuellen Geschäftsprozess oder einen Batchlauf bearbeitet wird. Der Entwurf muss also berücksichtigen, dass der Code zur Abarbeitung eines Geschäftsprozesses wenigstens teilweise sowohl im Online- als auch im Batchbetrieb verwendbar ist. Dadurch lässt sich die Entwicklungszeit für eine Gesamtanwendung verkürzen.
- **Verallgemeinerbarkeit des Batchkonzepts:** Das erarbeitete Batchkonzept muss sich einfach auf andere als den beispielhaft beschriebenen Geschäftsprozess verallgemeinern lassen.

Es werden im Verlauf des Buchs verschiedene Implementierungsvarianten des gleichen Geschäftsprozesses durch Messläufe bezüglich ihrer processing time verglichen. Die einzelnen Varianten unterscheiden sich nur durch genau einen gezielten Eingriff ins Design, sodass allein dessen Verbesserung an der Implementierung nachweisbar ist.

Die Messung der processing time einer einzelnen Variante ist in Abbildung 7.1 als Aktivitätsdiagramm dargestellt:

1. **Anlegen des Datenbestands:** Mit einem Skript werden Instanzen des beschriebenen Geschäftsobjekts vor dem Start *jeder* Batchvariante neu angelegt. Diese Instanzen dienen als workload (siehe Abschnitt 5.1). Die workload für die konkreten Batchläufe wird in Abschnitt 7.1 ausführlich vorgestellt. Der eingesetzte Applikationsserver speichert Beans in einem Cache, um die Bearbeitung zu beschleunigen. Dieser Cache lässt sich nicht abschalten. Um den Einfluss des Cache auf alle Messläufe gleich zu halten, wird der Applikationsserver vor jedem Messlauf neu gestartet. Dadurch ist sichergestellt, dass sich keine Instanzen aus den vorangegangen Läufen im Cache befinden.

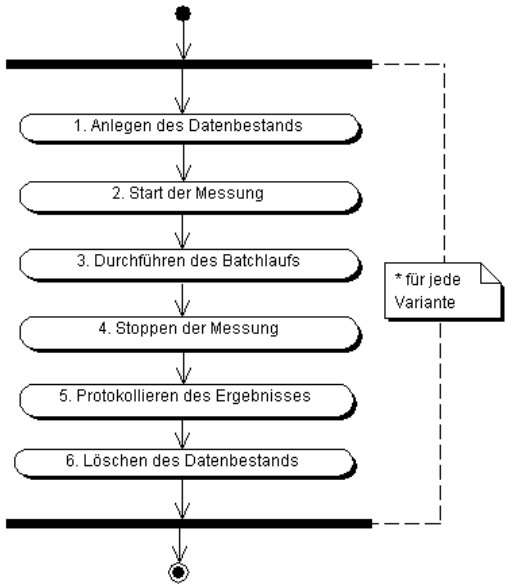


Abbildung 7.1: Aktivitätsdiagramm: Durchführung eines einzelnen Messlaufs

2. **Start der Messung:** Als Ergebnis des Messlaufs wird die processing time einer Variante ermittelt. Um sie zu ermitteln, wurden zwei einfache Klassen, `Stoppuhr` und `Zeitnehmer`, implementiert. Abschnitt 7.2 stellt vor, wie mit ihnen die Messungen durchgeführt werden.

3. **Durchführen des Batchlaufs:** Eine Session Bean fungiert als Rahmen für die Batchverarbeitung. Die Varianten haben damit ein gemeinsames, allgemeines Konzept zur Abarbeitung des Batchlaufs. Der Entwurfsansatz dazu wird in Abschnitt 7.3 präsentiert.

4. **Stoppen der Messung:** Ist das Ergebnis des letzten bearbeiteten Vertrags in die Datenbank geschrieben, wird die Zeit gestoppt. Die zwischen Starten und Stoppen verstrichene Zeit ist die processing time (vgl. Abschnitt 5.1).

5. **Protokollieren des Ergebnisses:** Die Verarbeitungszeit für einzelne Arbeitsschritte und die processing time wird durch die `Zeitnehmer`-Klasse ermittelt und ausgegeben. Entwurfsansatz, Umsetzung der Variante und diese Messergebnisse werden in einem einheitlichen Ergebnisprotokoll dokumentiert. Der Aufbau dieses Protokolls wird in Abschnitt 7.4 motiviert. Die Unterkapitel 8.2 bis 8.6 geben die Ergebnisse zu den einzelnen Messläufen gemäß dieses Aufbaus wieder.

6. **Löschen des Datenbestandes:** Der Datenbestand wird gelöscht, sodass ein weiterer Lauf mit identischen Testdaten durchgeführt werden kann.

Als Abschluss der Messläufe wird eine aggregierte Variante alle einzeln vorgestellten Verbesserungen zusammen umsetzen, um zu zeigen, welche Verbesserungen im Zusammenspiel der Varianten erreichbar sind. Diese Variante ist in Kapitel 8.7 dargestellt.

7.1 Datenbestand

Der Bestand an Testdaten wird mit einem Skript als Menge von einzelnen Instanzen des Geschäftsobjekts angelegt. Nachdem ein Batchlauf stattgefunden hat, wird der komplette Datenbestand gelöscht und für den nächsten Lauf neu angelegt, sodass alle Varianten die gleichen Startbedingungen haben.

Eine einzelne Instanz der KFZ-Haftpflichtversicherung hat folgendes Aussehen (siehe Abbildung 7.2). Die Instanz »vertraegeMitXY« von `VertragsBuendel` ist die Wurzel der baumartigen Vertragsstruktur. Zu diesem Bündel gehört nur ein einzelner Vertrag: »PersonenkraftwagenZurEigenverwendung«. Der Vertrag enthält drei thematische `RisikoBuendel`: »KFZ-Haftpflicht«, »InsassenUnfallPauschal« und »Teilkasko«. Auf alle Elemente des Teilkaskobündels wird in zweifacher Weise Rabatt gewährt. Einmal 10 %, da der Versicherungsnehmer bereit ist, 300 DM Selbstbehalt im Schadensfall zu tragen, und einmal einen »WerkstattRabatt«, da der Versicherungsnehmer als Werkstattbesitzer mit der Versicherung zusammenarbeitet.

An den Teilkaskobaustein ist eine Deckung gekoppelt, die vorgibt, bis zu welcher Summe Sach- und Vermögensschäden von der Versicherung getragen werden. Diese Deckung gilt für alle zur Teilkasko gehörigen ElementarRisiken gleichermaßen: »Brand«, (Zusammenstoß mit) »Haarwild«, »Diebstahl« und (Schäden durch) »Unwetter«.

Beim »InsassenUnfallPauschal«-`RisikoBuendel` ist jedem `ElementarRisiko` hingegen eine eigene Deckung zugeordnet: zum `ElementarRisiko` »Invalidität« gehört die Deckung »Leistunginvalidität« und regelt, bis zu welcher Summe Personenschäden getragen werden. Für die anderen ElementarRisiken sind die Strukturen analog.

Das `RisikoBuendel` »KFZ-Haftpflicht« wird modifiziert durch einen Schadensfreiheitsrabatt; der Versicherungsnehmer befindet sich in `SfKlasse`1 und erhält den entsprechenden Rabatt. Zum `RisikoBuendel` gehört nur ein `ElementarRisiko` »PrivatPKWHaftPflicht« mit einer Deckung »LeistungHaftpflicht«. Der auf den Vertrag anzuwendende Steuersatz »16 Prozent« ist dem Vertrag assoziiert.

Ein Skript legt für jede Variante die gleiche Anzahl an Geschäftsobjekten an. Je nach Implementierung kann die Struktur der Teilobjekte der Instanzen voneinander abweichen. Beispielsweise kann die Information über die `SfKlasse` mit in den `Vertrag` integriert werden. Die fachliche Aussagekraft ist in allen Varianten gleich. Die angelegten Daten müssen ein realistisches Verhältnis aufweisen zwischen Instanzen, die am Batchlauf teilnehmen und solchen, die nicht am Batchlauf teilnehmen werden. Ansonsten liefern Zeitmessungen von Suchanfragen an die Datenbank unrealistische Werte.

Im vorliegenden Beispiel wird davon ausgegangen, dass in jedem Monat gleichviele Verträge hauptfällig sind. Es nimmt an einem monatlichen Batchlauf also stets ein Zwölftel der aktuell gültigen Verträge teil.

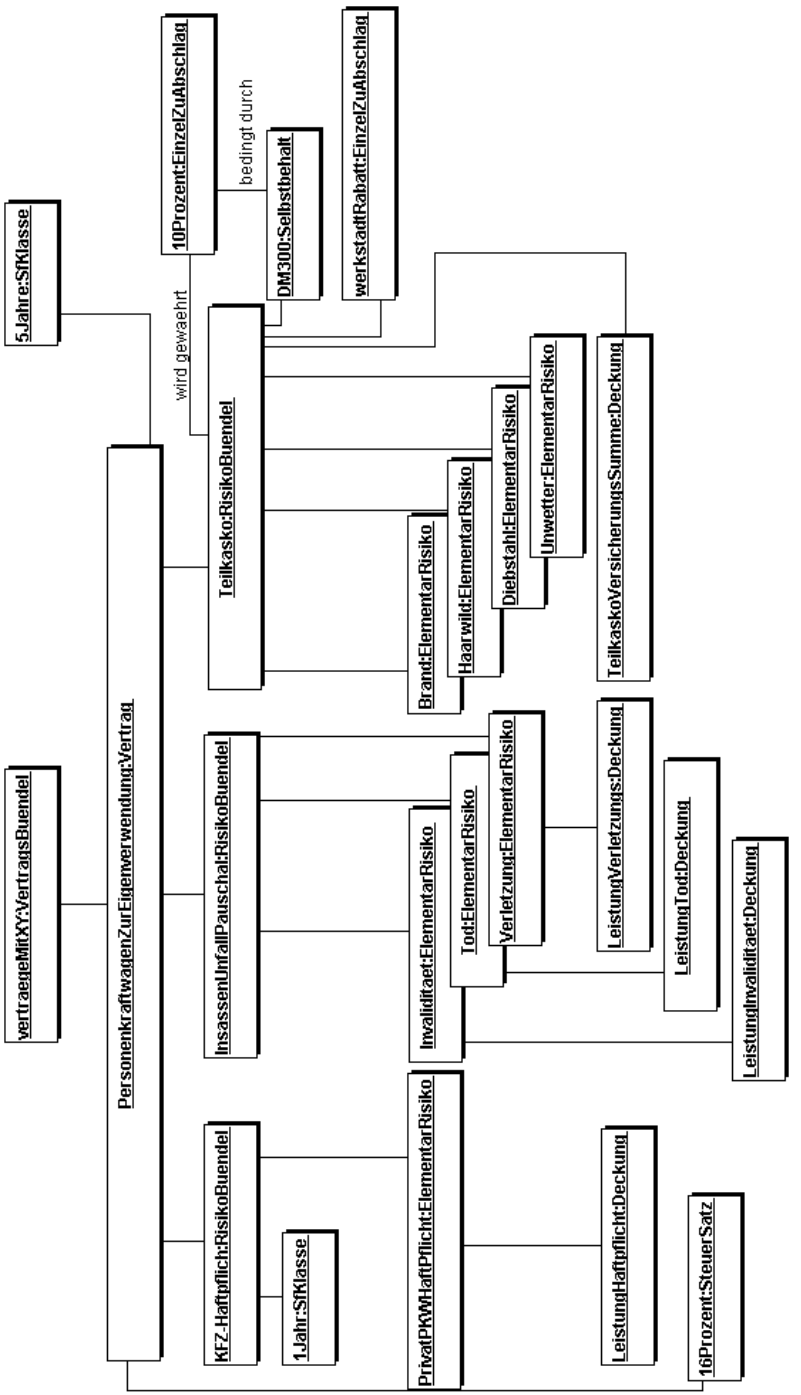


Abbildung 7.2: Instanzen-Diagramm: Versicherung eines Autos

Die Historisierung hat ebenfalls auf den Anteil der zu bearbeitenden Daten Einfluss. Unterstellt man, dass zu jedem Vertrag neun Historienstände existieren, so muss der Bestand nochmals um den Faktor 10 vergrößert werden.

Einem zu bearbeitenden Vertrag stehen damit elf nicht teilnehmende Verträge gegenüber, die in den anderen Monaten hauptfällig sind. Zu diesen zwölf Verträgen ist je das zehnfache Datenvolumen gespeichert, um die Historienstände zu simulieren. 119 nicht-teilnehmenden Vertragsständen steht also ein teilnehmender Vertrag gegenüber. Im weiteren Verlauf des Buchs wird dieses Verhältnis vereinfachend als »1 zu 120« angenommen. Von diesem Verhältnis wird nur abgewichen, um zu ermitteln, ob die processing time einer Variante von der Anzahl *nicht*-teilnehmender Verträge abhängig ist.

Als workload für den Batchlauf werden also am Batchlauf teilnehmende und am Batchlauf nicht-teilnehmende Verträge angelegt. Um das Datenvolumen der Historienstände nachzubilden, werden weitere nicht-teilnehmende Verträge angelegt, sodass auf jeden teilnehmenden Vertrag 120 nicht-teilnehmende Verträge kommen.

Zu jeder Variante werden Messläufe mit 1000 teilnehmenden Verträgen und entsprechend vielen nicht-teilnehmenden Verträgen durchgeführt.

7.2 Klassen zur Messung der processing time

Die processing time ist die Zeitspanne von Beginn des Batchlaufs bis zum Ende des Batchlaufs. Das Ziel einer Veränderung am Design ist die Verminderung der processing time. Die Klassen zur Messung der processing time sind in Abbildung 7.3 dargestellt.

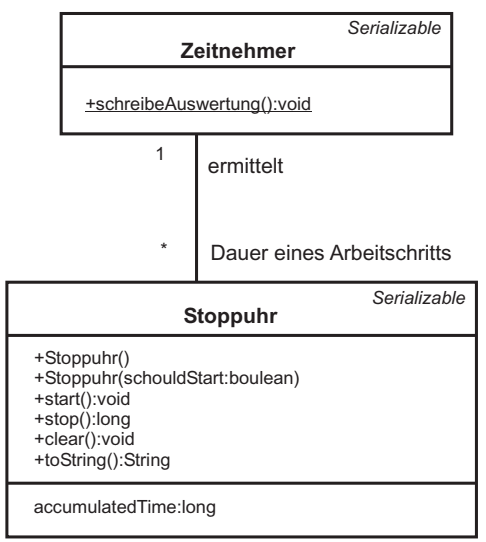


Abbildung 7.3: Klassendiagramm: Zeitnehmer und Stoppuhr

Der Zeitnehmer ermittelt die Dauer eines Arbeitsschritts durch Starten und Anhalten einer Stoppuhr. Nach Ende eines Messlaufes kann die ermittelte Dauer durch Aufrufen der Methode `schreibeAuswertung` auf der Textkonsole ausgegeben werden.

Die Stoppuhr dient der Messung einer Zeitspanne. Durch `start()` wird die Zeitmessung begonnen, durch `stop()` unterbrochen und durch `clear()` der Zähler auf »0« gesetzt. Die Methode `toString()` gibt die gemessene Zeit im Format »*Stunden*:*Minuten*:*Sekunden*,*Zehntelsekunden*« aus.

Intern benutzt die `Stoppuhr` die Klasse `java.util.Date`. Dadurch wird verstrichene Systemzeit und nicht die CPU-Zeit gemessen. Die CPU-Zeit ermittelt nur, wie viele Taktzyklen der CPU ein gewisser Prozess braucht. Das ist zu feingranular für die Zeitermittlung in den Messläufen, da auch die Wartezeiten für Datenbankzugriffe mitberücksichtigt werden müssen.

Zur Zeitmessung sind verschiedene Aufrufe an den `Zeitnehmer` in den Code des Batchlaufes eingebaut worden. Abbildung 7.5 stellt die Aufrufreihenfolge unabhängig von der Implementierung dar. Starten und Stoppen der Zeitmessung für die processing time sieht im Code von `BatchrahmenBean.java` konkret wie folgt aus:

```
Zeitnehmer.starteProcessingTime();

    Iterator alleTeilnehmer = sucher.finde().iterator();
    int anzahlTeilnehmer = 0;
    while(alleTeilnehmer.hasNext()){

        try {
            bearbeiter.bearbeiteGO(
                alleTeilnehmer.next()
            );
        }catch (BatchException b) {
            //...
        }
    }

Zeitnehmer.stopProcessingTime();
```

Listing 7.1: Messung der processing time

Der Zeitnehmer ist als Singleton implementiert, d.h. es gibt serverseitig genau eine Instanz von ihm. Start- und Stoppaufruf, die durch Einrückung vom restlichen Code abgesetzt sind, umschließen serverseitig die Methodenaufrufe zum Ausführen des Batchlaufs:

- `sucher.finde()` stellt eine Menge aller teilnehmenden Verträge zusammen.
- `bearbeiter.bearbeiteGO(...)` führt auf dem einzelnen Geschäftsobjekt die notwendigen Arbeitsschritte aus.

Durch die serverseitige Messung wird vermieden, dass Übertragungszeiten eines Netzwerks zwischen einem messenden Client und einem zu messenden Server das Ergebnis verfälschen. Das Zusammenspiel der Klassen für den Batchlauf wird detailliert in Abschnitt 7.3 erläutert.

Bei der Durchführung der Messläufe hat sich gezeigt, dass das Messen der processing time allein ausreicht, um eine Design-Regel zu validieren. Um allerdings einen Ansatzpunkt zu haben, wo die nächste Verbesserung sinnvoll ansetzen könnte, müssen weitere Messdaten genommen werden. Durch die zusätzlichen Messdaten lassen sich zeitaufwendige Operationen identifizieren. Eine Verbesserung des Anwendungs-Designs, sodass diese zeitaufwendigen Operationen weniger Zeit benötigen, verbessert damit die processing time.

Die folgende Auflistung erläutert, welche Zeitanteile der processing time zusätzlich gemessen wurden. Die Verfeinerung des Messrasters orientiert sich an den Aktivitätsdiagrammen in Abbildung 6.3 und Abbildung 6.4, wobei ähnliche Aktivitäten zusammengefasst werden. In Klammern ist vermerkt, welche Aktivitäten der Diagramme durch die genannte Messung abgedeckt werden. Da das Aktivitätsdiagramm den Vorgang sehr grob wiedergibt, ist die Zuordnung zwischen Aktivitäten und gemessenen Zeiten nicht eindeutig.

Es wird gemessen, wie viel Zeit der Batchlauf innerhalb dieser Aktivitäten verbringt. Werden Aktivitäten mehrfach ausgeführt, so ist der gemessene Wert also *die Summe* über der Dauer der einzelnen Ausführungen. Die processing time besteht im Wesentlichen aus den folgenden *Messpunkten*:

- **Sucher** misst den Zeitanteil der processing time, der benötigt wird, um die Menge der am Batchlauf teilnehmenden Verträge zu ermitteln. Die zugehörige Suchfunktionalität wird genau einmal zu Beginn des Batchlaufs aufgerufen (Aktivität 1 und 2). Zur Verbesserung der Basisvariante ist es notwendig, diesen Messpunkt nochmals zu unterteilen. Die Unterteilung ist ausführlich in Abschnitt 8.1 beschrieben.
- **aendereSfKlasse** ermittelt, wie lange es dauert, dem `Vertrag` die neue `SfKlasse` und damit den neuen Schadenfreiheitsrabatt zuzuweisen. Darin ist auch die Dauer der Suche nach der neuen `SfKlasse` enthalten (Aktivität 3).
- **historisiereVertrag** misst die Zeit, die zur Historisierung der Instanzen von `Vertrag` verbraucht wird. Die Funktionalität zur Historisierung wird einmal pro teilnehmendem Vertrag aufgerufen (Aktivität 4.1).
- **findeSfKlasse** misst die Zeit zum Auffinden der `SfKlasse` zu einem `Vertrag`. Durch diese `SfKlasse` wird der Rabatt vorgegeben, der in die Prämienneuberechnung eingeht (Aktivität 4.2).
- **ermittleZuAbschlagStufen** berechnet die Dauer für das Auffinden von `ZuAbschlag–Stufen` zu einem `Vertrag` oder `RisikoBuendel` (Aktivität 4.3 und 4.6).
- **ermittleSteuerSatz** berechnet die Dauer für das Auffinden des aktuellen `Steuersatzes` zu einem `Vertrag` (Aktivität 4.18).

- **findeRisikoBuendel** misst, wie lange es dauert, die `RisikoBuendel` zu einem Vertrag zu finden.
- **historisiereRisikoBuendel** misst die Zeit, die zur Historisierung der Instanzen von `RisikoBuendel` verbraucht wird. Die Funktionalität zur Historisierung wird dreimal pro teilnehmendem Vertrag aufgerufen (Aktivität 4.4).
- **findeEinzelZuAbschlaege** berechnet die Dauer für das Auffinden von `EinzelZuAbschlag`-Instanzen zu einem `RisikoBuendel` oder **`ElementarRisiko`** (Aktivität 4.7, 4.11, 4.12, 4.14 und 4.15).
- **findeElementarRisiken** misst, wie lange es dauert, die **`ElementarRisiko`**-Instanzen zu einem `RisikoBuendel` zu finden.
- **historisiereElementarRisiko** misst die Zeit, die zur Historisierung der Instanzen von `ElementarRisiko` verbraucht wird. Die Funktionalität zur Historisierung wird achtmal pro teilnehmendem Vertrag aufgerufen, da jede Vertragsinstanz acht Elementarrisiken besitzt (Aktivität 4.8).
- **ermittleMaxMindestBeitrag** bestimmt die Dauer der Ermittlung eines maximalen Mindestbeitrags aus einem Tarif (Aktivität 4.9). Diese Methode wird achtmal pro teilnehmendem `Vertrag` aufgerufen, d.h. einmal pro **`ElementarRisiko`** des Beispielvertrags (siehe Abbildung 7.2).
- **ermittleBeitrag** berechnet die Dauer für das Ermitteln eines Beitrags aus dem Tarif (Aktivität 4.10). Diese Methode wird achtmal pro teilnehmendem `Vertrag` aufgerufen, d.h. einmal pro `ElementarRisiko` des Beispielvertrags (siehe Abbildung 7.2).
- **berechnePrämie** Aus den tariflich ermittelten Werten und den `ZuAbschlag`-Instanzen, usw. wird der Beitrag im `ElementarRisiko` berechnet (Aktivitäten 4.11–4.15).

Diese Messwerte werden nach Ende des Batchlaufs durch den Aufruf von `Zeitnehmer.schreibeAuswertung()` ausgegeben. Die Auswertungskapitel zu den einzelnen Implementierungsvarianten geben die gemessenen Werte tabellarisch wieder. Die einfache Apparatur ist der große Vorteil dieser Art zu messen. Eine solche Messung kann mit einem Aufwand im Bereich weniger Stunden an jedem in Java implementierten Prototypen vorgenommen werden. Eine Einarbeitung in Bibliotheksklassen ist nicht notwendig. Das Ergebnis erklärt sich unmittelbar selbst, sodass auch die Auswertung leicht fällt.

Die geringe Messgenauigkeit ist ein Nachteil der Messmethode. Die Messungen haben gezeigt, dass die gemessene processing time bei mehreren Messungen mit der gleichen Variante im Bereich von einer Minute schwankt. Diese Ungenauigkeit liegt jedoch weit unter den Verbesserungen, die erreicht werden. Die Bewertung zur zusammengefassten Variante wird dies bestätigen (siehe Abschnitt 8.7.3).

Die einzelnen Messpunkte können teilweise stark verfälscht werden durch die Garbage Collection. Die Garbage Collection ist ein Dienst der Java-Laufzeitumgebung, der nicht mehr benötigte Objekte aus dem Hauptspeicher entfernt und so Speicherplatz freigibt. Dieser Dienst kann je nach Größe des Speichers und Systemlast pro Ausführung mehr als 10 Sekunden dauern. Aufgrund der Dauer eines Batchlaufs konnten nicht genügend

Messungen vorgenommen werden, um Mittelwerte und Standardabweichungen anzugeben. Anstelle dessen wurden mehrere Messungen durchgeführt und anhand der sich abzeichnenden Tendenzen eine »repräsentative« Messung ausgewählt.

Eine Verbesserung gilt als »wirksam«, wenn die processing time sich um mehr als eine Minute verringert hat. Durch die Messung wird lediglich eine zweiwertige Entscheidung getroffen: »Verbesserung eingetreten« oder »Verbesserung nicht eingetreten«. Aufgrund des beispielhaften Charakters der Anwendung und der unrealistischen Hardware wird darauf verzichtet, quantitative Aussagen bezüglich der Effektivität der Verbesserung auf einem produktiven System zu machen. Für die Bestätigung einer Design-Regel reicht es damit, eine *Tendenz* anzugeben, sodass die absolut gemessene Zeit nicht von Bedeutung ist.

Neben dem gewählten Ansatz gibt es zwei Gruppen von alternativen Messmethoden:

1. **Standardisierte Open-Source Frameworks:** Sind die absolut gemessenen Zeiten wichtig, um z.B. am Ende eines Projekts Performance Tuning zu betreiben, so ist ein standardisiertes Messverfahren vorzuziehen. Das »Application Response Measurement« (ARM) in der Version 3.0 für Java ist beispielsweise ein Framework zur Messung von Java-Anwendungen. Es ist durch die Open Group standardisiert und frei verfügbar (vgl. [14]). Dieses Werkzeug hat den Nachteil, dass es nur in selbst geschriebenen Quelltext eingebunden werden kann. Mehr als die Hälfte der erzeugten ausführbaren Dateien (Class-Dateien) der Anwendnung sind jedoch durch Werkzeuge des Applikationsserverherstellers generiert. In diesen Klassen kann durch ein Verfahren wie die ARM 3.0 nicht gemessen werden. Der zusätzliche Aufwand für Einarbeitung in die Messmethode und die Bibliotheksklassen der ARM 3.0 kann auf zwei Wochen geschätzt werden. Eine sinnvolle Steigerung der Messgenauigkeit für das Ziel dieses Buchs wird dadurch nicht erreicht, da die Verbesserungen der processing time deutlich im Bereich mehrerer Minuten liegen.

2. **Profiling-Werkzeuge:** Profiling-Werkzeuge für Java führen Messungen über Speicherverbrauch, Geschwindigkeit und andere Performanzkriterien über eine Schnittstelle zur Laufzeitumgebung oder zum Container eines Applikationsservers aus. Dadurch sind auch die Dienste des Containers messbar. Bei den zuvor genannten Varianten zeigen sie sich nur indirekt in der Differenz der processing time zu der Summe der angegebenen Messpunkte. Wenn beispielsweise das Aktualisieren der persistenten Daten bis zum Ende der Transaktion verzögert wird, ist ein Profiling-Werkzeug notwendig, das innerhalb des Containers messen kann. Ein solches Werkzeug stand zu Beginn der Arbeiten an diesem Buch für die benutzte Version des Applikationsservers nicht zur Verfügung, da der Applikationsserver selbst sich noch im Beta-Stadium befand. Mittlerweile werden auf den Internet-Seiten des Herstellers verschiedene Werkzeuge empfohlen.

Der Performanzunterschied fällt bei grundlegenden Änderungen am Design wesentlich markanter aus, als bei Korrekturen, die im Rahmen eines nachträglichen Tuning vorgenommen werden [53]. Für die Messung der Prototypen in diesem Buch wird ein einfaches, wenn auch wenig genaues Messwerkzeug selbst geschrieben. Da dieses Buch

Design-Varianten beschreibt, werden die Unterschiede deutlich über der Messgenauigkeit des Werkzeugs liegen, wie die verschiedenen Messungen zeigen.

Für ein abschließendes Tuning in Projekten wird, soweit die Zeit und das Budget es erlauben, der Erwerb eines Profiling-Werkzeuges empfohlen. Die Firma bea empfiehlt beispielsweise seit kurzem für ihre Server den Workload-Generator »The Grinder«, der im Internet unter grinder.sourceforge.net verfügbar ist (Stand November 2001). Er wird kombiniert mit einem Profiling-Werkzeug eingesetzt, um eine fertige Anwendung zu analysieren und einem Tuning zu unterziehen. Wertvolle Hinweise zum Thema Profiling und Tuning von Java-Anwendungen können [60] entnommen werden.

7.3 Batchrahmen

In jeder Variante wird ein Rahmenwerk zur Steuerung des Batchlaufs benötigt. Das ist in allen Fällen eine Session Bean, innerhalb derer verschiedene andere Beans aufgerufen werden.

Entwurfsansatz: Ein erster Lauf über den Bestand identifiziert die Menge teilnehmender Geschäftsobjekte. Ist diese Menge erstellt, greift ein zweiter Prozess nacheinander auf die Geschäftsobjekte zu und bearbeitet sie. Zu jedem Geschäftsobjekt kann durch einen Logging-Mechanismus ein Eintrag erstellt werden, der beschreibt, ob es erfolgreich bearbeitet wurde.

Such- und Bearbeitungsfunktionalität sind voneinander getrennt. Das heißt es gibt einen allgemeinen Batchrahmen, der für Fehlerbehandlung, Logging, Transaktionssicherheit und Verkapselung von zu beliefernden Batchschnittstellen zuständig ist. In diesen werden abhängig vom Batchlauf verschiedene Funktionalitäten zum Auffinden und Bearbeiten der teilnehmenden Geschäftsobjekte eingebunden.

Das Klassendiagramm zu diesem Entwurf ist in Abbildung 7.4 dargestellt. Die Klassen nehmen folgende Aufgaben wahr:

- **`Batchrahmen`:** stellt allgemeine Dienste zur Verfügung, die bei Batchläufen benötigt werden. Das sind Start- und Stopp-Methoden und Fehlerbehandlungsroutinen. Der `Batchrahmen` wird als Session Bean implementiert, um die Transaktionslogik des Containers zu nutzen. Er erhält bei seiner Erzeugung ein Java-Objekt, das das Interface **`Sucher`**, und ein weiteres Objekt, das das Interface **`Bearbeiter`** implementiert. Zusätzlich könnte ein Objekt, das das Interface **`BatchExceptionHandler`** implementiert, übergeben werden. In diesem Objekt ist die für den Batchlauf spezifische Fehlerbehandlung implementiert.
- **`Sucher`:** ist ein Interface, das die Methodensignatur zum Ermitteln der teilnehmenden Geschäftsobjekte definiert. Klassen, die dieses Interface implementieren, haben die Aufgabe, nach geschäftsvorfallspezifischen Kriterien aus dem Bestand die Elemente zu identifizieren, die am Batchlauf teilnehmen. Das Ergebnis dieser Methode ist eine Menge von Verweisen auf die Geschäftsobjekte.

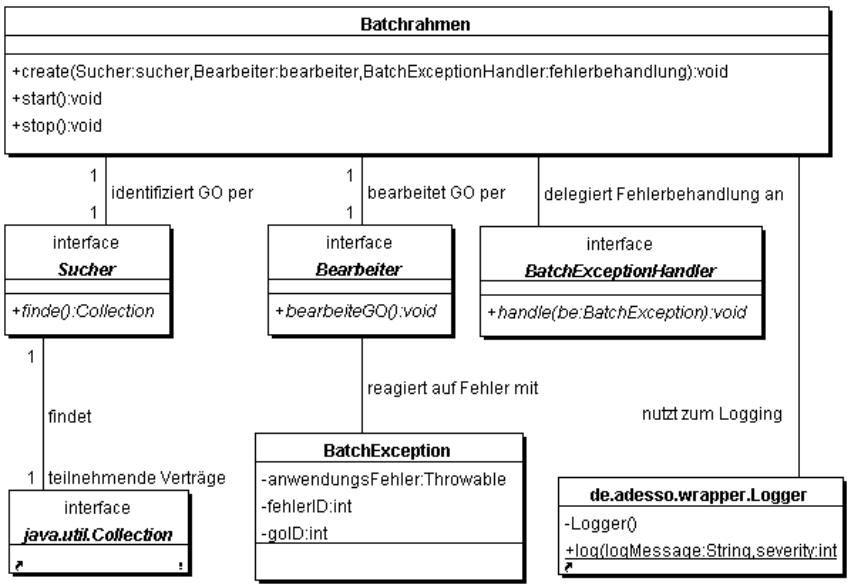


Abbildung 7.4: Klassendiagramm: allgemeine Klassen für den Batchlauf

- **`Bearbeiter`:** ist ein Interface, das eine einzige Methode definiert, innerhalb derer genau ein Geschäftsobjekt bearbeitet wird. Als Eingabe erhält die Methode ein Objekt `java.lang.Object`. Die Methode hat keinen Rückgabewert. Bei einem Scheitern wird eine **`BatchException`** geworfen, die nähere Information zur aufgetretenen Ausnahme enthält. Die Fehlerbehandlung liegt außerhalb des `Bearbeiters`. Daher ist es Aufgabe des `Bearbeiters`, alle auftretenden Exceptions in `BatchExceptions` zu wrappen.
- **`BatchExceptionHandler`:** definiert eine Methode, an die eine `BatchException` übergeben werden kann. Die implementierenden Klassen verarbeiten dann die Ausnahme, die in der `BatchException` enthalten ist. Dadurch kann je nach Geschäftsvorfall eine Fehlerbehandlung implementiert werden.
- **`Collection`:** dient als Rückgabetyp der Methode im `Sucher`-Interface.
- **`BatchException`:** kapselt alle Information, die für ein Fehlerbehandlungskonzept eines Batchlaufs notwendig sind. Die Klasse kapselt den eigentlich aufgetretenen Anwendungsfehler im Attribut `BatchException.anwendungsFehler` und kann ihn so an den `BatchExceptionHandler` übermitteln.
- **`Logger`:** wird benutzt zum Protokollieren von Erfolg oder Misserfolg eines Arbeitsschrittes. Für die prototypische Implementierung in diesem Buch delegiert der Logger nur an den Logging-Mechanismus des benutzten Applikationsservers.

Das beschriebene Rahmenwerk für Batchläufe ist für alle Varianten bis auf kleinere Änderungen gleich. Es wird sich beispielsweise in Kapitel 8.5 zeigen, dass es sinnvoll sein kann, das `Bearbeiter`-Interface durch so genannte Message-Driven Beans (siehe dort) zu ersetzen.

Dem zuvor dargestellten Entwurf folgend muss der Geschäftsprozess »Sfr-Umstufung« passend als `Sucher-Bearbeiter`-Pärchen umgesetzt werden. Das Sequenzdiagramm in Abbildung 7.5 verdeutlicht das Zusammenspiel von `Sucher` und `Bearbeiter`. Dargestellt ist ein Messlauf.

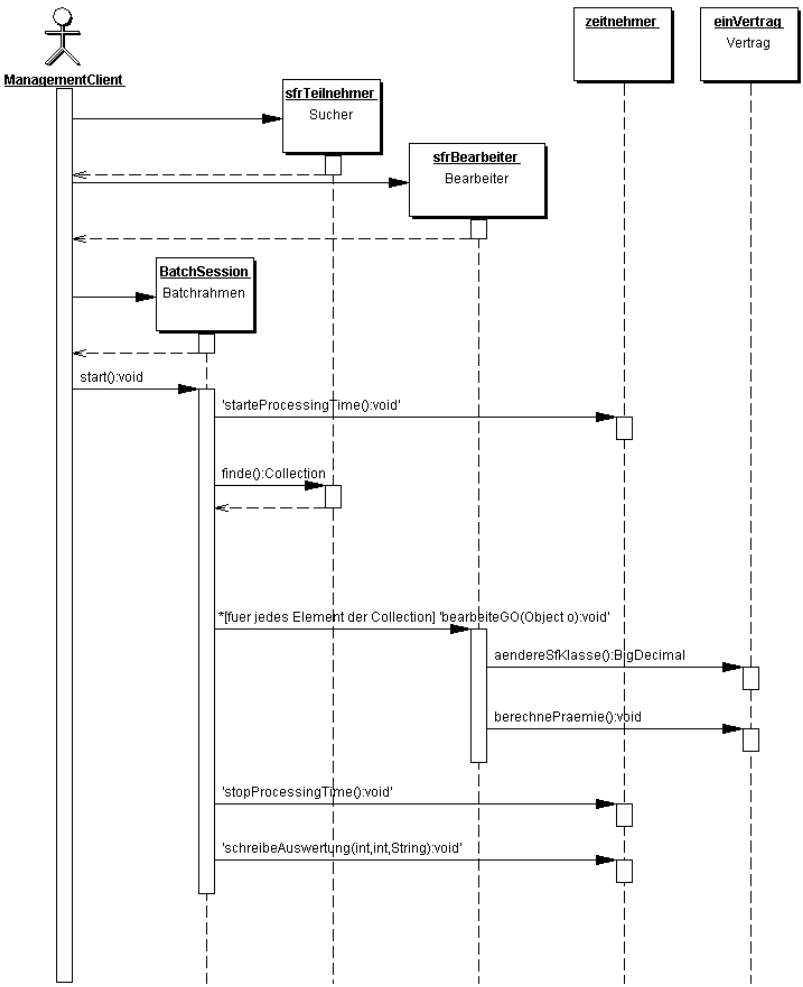


Abbildung 7.5: Sequenzdiagramm: Sfr-Umstufung mit `Sucher` und `Bearbeiter`

Das Programm `ManagementClient` startet den Batchlauf. Es erzeugt für die Sfr-Umstufung zuerst zwei Klassen, die das `Sucher`- und das `Bearbeiter`-Interface implementieren. Die `Sucher`-Klasse kapselt die Kriterien, nach denen ein Geschäftsobjekt an einem Batchlauf teilnimmt. Schwellenwerte der Kriterien können als Parameter beim Erzeugen übergeben werden. Die `Bearbeiter`-Klasse enthält die Information, wie ein Geschäftsobjekt zu bearbeiten ist. Mit diesen Klassen wird dann der `Batchrahmen` »BatchSession« in Form

einer SessionBean erzeugt. Unmittelbar vor dem Anfang des Batchlaufs wird durch den Aufruf von `Zeitnehmer.starteProcessingTime()` die Zeitmessung begonnen. Die Session Bean stößt danach die Methode `finde()` auf dem `Sucher` »Sfr-Teilnehmer« an. Rückgabewert dieser Methode ist eine `Collection` von `Vertrag`-Objekten. Für jedes Element der Collection wird dann die Bearbeitungsfunktionaliät auf dem `Bearbeiter` »sfrBearbeiter« mittels `bearbeiteGO(Object o)` aufgerufen.

Auf welche Weise diese Bearbeitung geschieht, kann durch die Interface-Konstruktion offengehalten werden. Welche Aufrufe von dieser Klasse zur Bearbeitung ausgehen könnten (siehe unterer Steuerfokus von »sfrBearbeiter«), ist von Variante zu Variante verschieden und daher nicht mehr in diesem Diagramm dargestellt. Eine Verfeinerung der eigentlichen Bearbeitungsaufrufe wird in den einzelnen Varianten dokumentiert.

Zum Ende der Bearbeitung wird mit `Zeitnehmer.stopProcessingTime()` die Zeitmessung beendet. Die processing time und alle weiteren gemessenen Werte werden mit `Zeitnehmer.schreibeAuswertung(...)` auf der Konsole ausgegeben. Sie werden tabellarisch in den Messprotokollen zu den Implementierungsvarianten festgehalten. Anhand dieser Messwerte wird verglichen, wie performant das Design einer gewissen Variante ist. Damit sind die Messwerte zentrales Ergebnis der Implementierungen und dienen der Validierung der Design-Regeln.

Aspekte der Restartfähigkeit eines Batchsystems bleiben bei diesem Entwurf unberücksichtigt. `Bearbeiter` und `Sucher` müssen passend zueinander implementiert werden, sodass die Instanzen, die der eine als Ergebnis erzeugt, von dem anderen verarbeitbar sind. Solche Typfehler lassen sich aber bereits beim ersten Testlauf finden.

Wichtigster Vorteil dieses Entwurfs ist der Freiraum, den er für die Umsetzung eines konkreten Batchlaufs lässt. Durch die gewählte Modularisierung bleibt es für den spezifischen Geschäftsprozess offen, in welcher Form das Geschäftsobjekt vorliegt: serialisiert, als Value-Object (vgl. [46, S. 257ff]) oder als Remote Interface. Weiterhin bleibt offen, wo der eigentliche Code der Bearbeitung liegt. Die Klasse, die das `Bearbeiter`-Interface implementiert, kann eine Methode auf einer Bean des Geschäftsobjekts aufrufen, eine SessionBean zur Bearbeitung starten, selbst die Bearbeitung durchführen oder serialisierte Information an ein Message-Driven Bean senden.

Diese Modularisierung stellt sicher, dass der `Batchrahmen` durch Austausch von `Sucher` und `Bearbeiter` für beliebige andere Batchläufe eingesetzt werden kann. Damit ist die Forderung nach Verallgemeinerbarkeit und Modularität des Batchkonzepts des Kapitels 7 eingelöst.

7.4 Ergebnisprotokoll zu den Messläufen

Die Durchführung und das Ergebnis eines Messlaufs sind geeignet zu dokumentieren. Jede Variante wird daher nach dem gleichen Muster in einem Messprotokoll dokumentiert. Dieses Messprotokoll wird im folgenden Abschnitt beschrieben.

Die Dokumentation eines Messlaufs besteht aus den folgenden Teilen:

- **Überblick:** in wenigen Sätzen wird das Ergebnis des Tests zusammengefasst. Das heißt es wird beschrieben, welche Verbesserung wie erreicht wurde.
- **Literatur:** die Quellen, auf der die Verbesserung beruht, werden referiert.
- **Übertragung und Entwurf:** Die in der Literatur gefundene Verbesserungsmöglichkeit wird auf die Beispielanwendung übertragen. Die Veränderungen an der Basisvariante ist begrenzt und zielgerichtet. Anhand geeigneter Diagramme wird die Abweichung des Entwurfs von der Basisvariante beschrieben. Es wird erörtert, welche Verbesserung zu erwarten ist.
- **Messergebnis:** Die ermittelten Zeiten der Einzelschritte und die »processing time« werden mit den Werten der Basisvariante verglichen. Anhand dessen lässt sich bestimmen, ob die erwartete Verbesserung eingetreten ist. Weitere Einflussgrößen, die sich bei der Implementierung als relevant erwiesen haben, werden diskutiert.
- **Bewertung:** Die Testergebnisse werden interpretiert. Die systematischen Wechselwirkungen werden beschrieben. Das heißt durch das geänderte Design der Implementierungsvariante werden andere Geschäftsprozesse beeinträchtigt. Diese Beeinträchtigungen werden offen gelegt. Auf immanente Wechselwirkungen, z.B. verschlechterte Performanz der Online-Geschäftsprozesse bei aktivem Batchlauf, wird nicht eingegangen. Der Anpassungsaufwand wird mit dem erzielten Ergebnis verglichen.
- **Design-Regel:** als Zusammenfassung des Testergebnisses wird eine Design-Regel für die performante Nutzung der EJB-Architektur formuliert.

8 Die Implementierungsvarianten

Die beiden vorangegangenen Kapitel erläuterten den Geschäftsprozess (vgl. Kapitel 6) und wie mit ihm Design-Regeln ermittelt werden können (vgl. Kapitel 7). Die folgenden Unterkapitel beschreiben die zu implementierenden Varianten des Batchlaufs »Sfr-Umstufung«. Sie sind nach der in Kapitel 7.4 motivierten Struktur aufgebaut und bilden den praktischen Kern des Buchs. Ergebnis jeder Implementierungsvariante ist eine Verbesserung der processing time. Die Basisvariante wird die Wichtigkeit der Nutzung der EJB QL herausstellen. Alle weiteren Varianten basieren daher auf der EJB QL-Variante (siehe Abschnitt 8.2).

Abgeschlossen werden die Varianten durch das Unterkapitel 8.7. In diesem Kapitel wird die Kombination der vorangegangenen Varianten vorgenommen und ihr Laufzeitverhalten gemessen. Da nicht alle Begründungen und Quellen zu den Design-Entscheidungen wiederholt werden sollen, wird von der Kapitelstruktur der Verbesserungsvarianten abgewichen. Stattdessen wird der Entwurf ausführlich beschrieben und zur Begründung nur auf vorangegangene Kapitel verwiesen. Im Vergleich zur Basisvariante wird gezeigt, welche Verbesserung überhaupt erzielbar ist.

8.1 Die Basisvariante

Die Basisvariante ist gekennzeichnet durch eine naive Implementierung, die die Datenbank unter dem Applikationsserver unbedacht lässt. Alle Geschäftslogik wird auf Ebene der Beans ausprogrammiert wie bei gewöhnlichen Java-Applikationen ohne besondere Persistenzdienste. Die Datenbank nimmt nur Änderungen an Attributen zur Speicherung entgegen oder liefert Instanzen angeforderter Objekte. Jedes Objekt des Analysemodells wird als eigenständige Enterprise JavaBean umgesetzt, die per CMP in der Datenbank gespeichert werden. Jede Bean bildet auf genau eine Tabelle in der Datenbank ab, jede Tabelle enthält nur Daten einer Bean.

Es werden keine EJB 2.0-spezifischen Möglichkeiten genutzt. Proprietäre Optimierungen des Applikationsservers werden abgeschaltet, da sich ihr Einfluss auf anderen Plattformen nicht nachbilden lässt. Die wichtigsten Konfigurationsparameter des Applikationsservers werden im Anschluss an die Basisvariante diskutiert.

8.1.1 Entwurf der persistenten Struktur

Da die objektorientierte Analyse des Geschäftsobjekts und des Geschäftsprozesses bereits in Kapitel 6 behandelt worden ist, kann für die Basisvariante direkt mit dem Entwurf der zu speichernden Struktur begonnen werden.

Das Entwurfsziel der *statischen Struktur* wurde im voranstehenden Überblick erläutert. Im Folgenden wird anhand eines Ausschnitts des Analysemodells die Überführung des Analysemodells in ein Implementierungsdiagramm skizziert. Im Anschluss daran wird der Entwurf des Datenbankschemas erläutert. Als Ausschnitt des Analysemodells seien

die Klassen `Vertrag`, `SfKlasse`, `RisikoBuendel`, `ElementarRisiko` und `EinzelZuAbschlag` gewählt (siehe Abbildung 8.1).

Da es zuerst um die Umsetzung der statischen Struktur gehen soll, sind keine Methoden abgebildet. Die dargestellten Attribute haben sich beim Durchspielen verschiedener Anwendungsszenarien ergeben (vgl. dazu [51, S. 189ff]).

Das Attribut `Vertrag.zahlweise` speichert ab, in welchen Rhythmen der Versicherungsnehmer seine Prämie entrichten will: ist eine »1« gespeichert, bedeutet das jährliche Zahlweise, eine »2« halbjährliche, usw. Für den Geschäftsprozess »Sfr-Umstufung« sind die folgenden Attribute notwendig. Im Vertrag bestimmt das Attribut `automatischeSfr–Umstufung`, ob überhaupt ohne Bestätigung des Sachbearbeiters die `SfKlasse` maschinell zu ändern ist. Das Datum der letzten Umstufung `letzteSfrUmstufung` gibt an, wann zuletzt die SfKlasse geändert wurde.

Vom `beginnDatum` ist die Hauptfälligkeit abhängig. Bei der zu der Umstufung gehörenden Beitragsneuberechnung muss der `ratenZuschlag` berücksichtigt werden. Der neuberechnete Beitrag wird schließlich in textcodepraemie gespeichert.

Die `SfKlasse` gehört zu einem Schadensfreiheitsrabatt »sfr«, der beispielsweise in einer Produktkomponente (siehe Abbildung 6.2) gespeichert ist. `beitragsFaktor` gibt an, durch welchen Faktor der tariflich ermittelte Beitrag zu modifizieren ist. `schadenfreieJahr` speichert, ab wie vielen schadensfreien Jahren man in diese `SfKlasse` gelangen kann. `hochstufungsKlasse` und `rueckstufungsKlasse` sind Referenzen auf `SfKlasse`, in die ein Versicherungsnehmer eingestuft wird, falls er ein weiteres schadensfreies Jahr versichert war bzw. falls er einen Schaden geltend gemacht hat. Durch die Assoziation zu Vertrag und `RisikoBuendel` wird ermöglicht, sowohl auf thematisch gebündelte Einzelzrisiken als auch auf den ganzen Vertrag Schadenfreiheitsrabatt zu gewähren.

Das `RisikoBuendel` enthält Information, die für die Beitragsberechnung von Belang ist. `beitragsfrei` gibt an, ob überhaupt für die darunter liegenden `ElementarRisiken` eine Prämie berechnet wird. `beitrag` enthält die Summe der Beiträge der untergeordneten EinzelRisiken.

Im `EinzelZuAbschlag` beschreibt das Attribut `wert`, welcher absolute Betrag auf den Beitrag zu addieren oder zu subtrahieren ist.

Über das `ElementarRisiko` wird mit den Risikomerkmalen `risikoMerkmalFarbe`, `risiko–MerkmalHubraum` und `risikoMerkmalGruppe` über den Tarif `relevanterTarif` eine Prämie ermittelt, falls dieses `ElementarRisiko` nicht beitragsfrei ist. Falls im Tarif ein maximaler Mindestbeitrag `maxMindestBeitrag` definiert ist, ist dieser ebenfalls bei der Beitragsberechnung zu bestimmen. Die Bedeutung der anderen Attribute ist auf der beiliegenden CD nachzulesen.

Bei der *Implementierung* dieser Struktur ist zu überlegen, wie die Beziehungen zwischen Klassen gespeichert werden, und welche zusätzlichen technischen Attribute notwendig sind. Zur Speicherung der Instanzen der Klassen in einer Datenbank ist ein eindeutiger Primärschlüssel notwendig. Dazu erhält jede Klasse ein Attribut `id`, wobei der Typ eine zehnstellige Zahl sein soll.

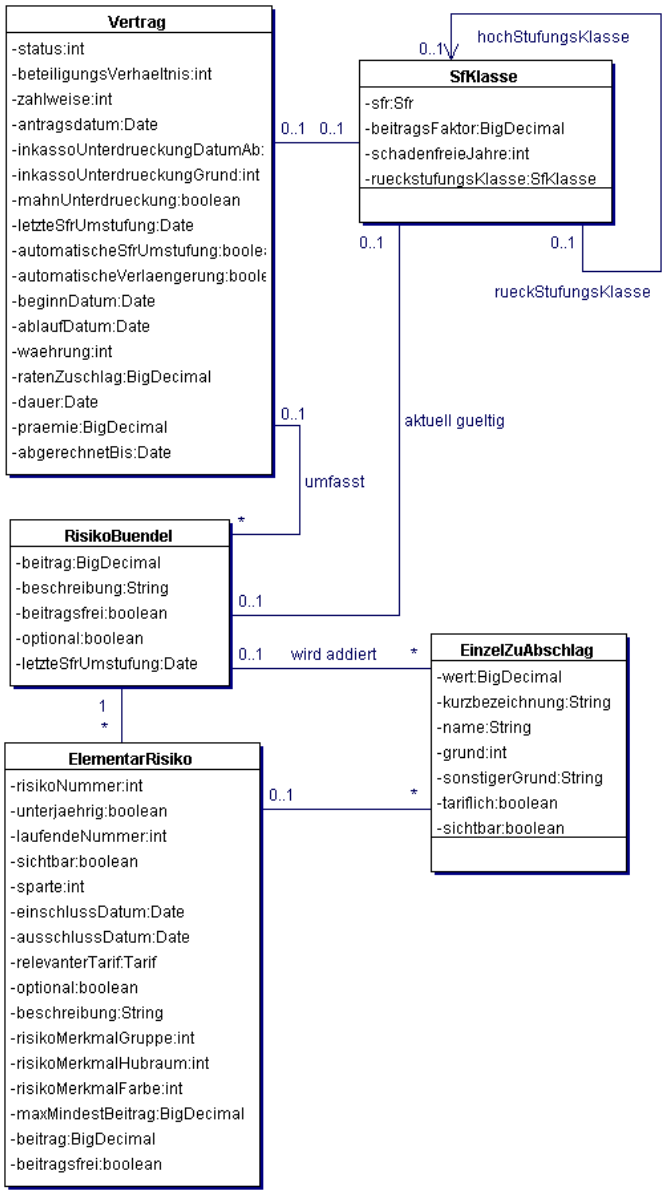


Abbildung 8.1: Klassendiagramm: Ausschnitt des Analysemodells mit Attributen

Eine Anforderung an das System ist die Historisierbarkeit der einzelnen Klassen. Wie in Kapitel 6.3 angedeutet und in [33] detailliert beschrieben, muss dafür der Gültigkeitszeitraum und der Erfassungszeitraum gespeichert werden. Für die Gültigkeit werden die Attribute `hGueltigAb` und `hGueltigBis` gespeichert. Für die Erfassung im System werden

die Attribute `hErfasstAm` und `hErsetztAm` gewählt. Ein weiteres Attribut `hStorniert` gibt an, ob ein Vertragsstand storniert wurde.

Um fachliche Identität einer Klasse über die Zeit hinweg und in Assoziationen darstellen zu können, wird ein zusätzliches Attribut `fachId` benutzt. Die historisierten Stände eines `ElementarRisiko` haben alle die gleiche `fachId`, da es sich fachlich um genau ein `ElementarRisiko` handelt. Technisch ist aber jeder Stand eine eigene Instanz, sodass sie alle eine andere `id` als Primärschlüssel haben.

Bei Anwendung der beschriebenen Überlegungen ergibt sich das Klassendiagramm in Abbildung 8.2. Im unteren Bereich der Klassen sind die sieben zusätzlichen Attribute `id`, `fachid`, `hStorniert`, `hGueltigBis`, `hGueltigAb`, `hErfasstAm` und `hErsetztAm` erkennbar.

Bei der Klasse `SfKlasse` wurde eine Ausnahme gemacht. Die Schadensfreiheitsrabattstaffel soll von allen Verträgen gleichermaßen benutzt werden und ist damit vom einzelnen Vertrag nicht zu ändern. Erhält ein Vertrag einen anderen Rabatt, so wird eine andere Instanz von `SfKlasse` referenziert und der Vertragsstand mit der alten, gespeicherten Referenz historisiert. Die `SfKlasse` selbst wird also nicht historisiert.

Die Referenzen zwischen den dargestellten Klassen sollen auch persistent gemacht werden. Dies geschieht durch das Speichern des Primärschlüssels von Vertrag als Fremdschlüssel in `RisikoBuendel`. Da es sich hierbei um eine fachliche Beziehung handelt, wird das Attribut `fachId` in `RisikoBuendel` als Fremdschlüssel gespeichert. Werden alle Risikobündel eines Vertrags gesucht, so muss die Datenbank nach `RisikoBuendeln` durchsucht werden, die die `fachId` des Vertrags enthalten. Über die zeitlichen Attribute lässt sich dann die zu dem fraglichen Zeitpunkt gültige Instanz finden. Der Entwickler einer Bean muss dafür in der Bean abstrakte get- und set-Methoden deklarieren, die RemoteInterfaces bzw. Collections benutzen. Im Deployment Descriptor sind dann auf Ebene von Fremdschlüssel und Datenbanktabellen die Relationen zu definieren. Mit den Fremdschlüsselbeziehungen und den Historisierungsattributen ergibt sich für die Klassen des Entwurfsausschnitts das in Abbildung 8.3 dargestellte Datenbankschema.

In der Darstellung des Datenbankschemas fällt auf, dass keine Relationen zwischen Tabellen dargestellt sind. Das hat seinen Grund in der gewählten Form der Historisierung. Als Konsequenz können zwischen historisierten Klassen keine vom Container verwalteten Relationen benutzt werden. Diese Relationen werden durch manche Applikationsserver (z.B. Persistence PowerTier) mit einem speziellen Cache beschleunigt, sodass an dieser Stelle weitere Performanz gewonnen werden könnte.

An der Namensgebung ist deutlich zu sehen, wie die Strukturen der Datenbank von den Strukturen auf Ebene der Beans entkoppelt sind. Jeder Bereich kann seine eigene Namenskonvention behalten. Auf der Ebene von Datenbanken werden gerne global eindeutige Bezeichner nach einer unternehmensweiten Konvention benutzt, daher der dreibuchstabige Kürzel vor jedem Attribut. Diese Konvention vereinfacht die Programmierung von SQL-Skripten und die Pflege der Datenbank. Daher ist es sehr nützlich, eine Trennung zwischen dem Namensraum der Datenbank und dem Namensraum auf Ebene der Beans zu haben. Auf der Ebene von Beans können — wie vom objektorientierten

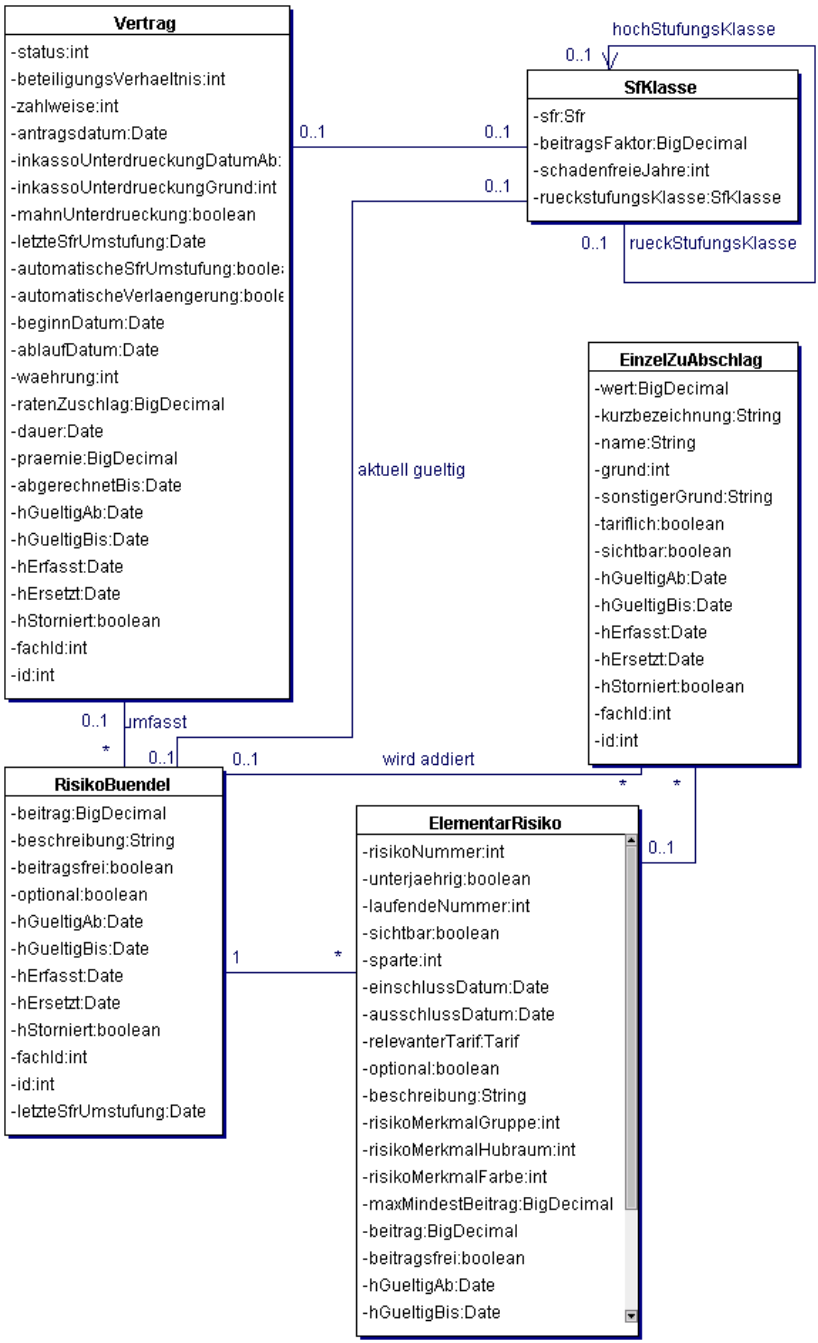


Abbildung 8.2: Klassendiagramm:Ausschnitt des Analysemodells mit Historisierungsattributen

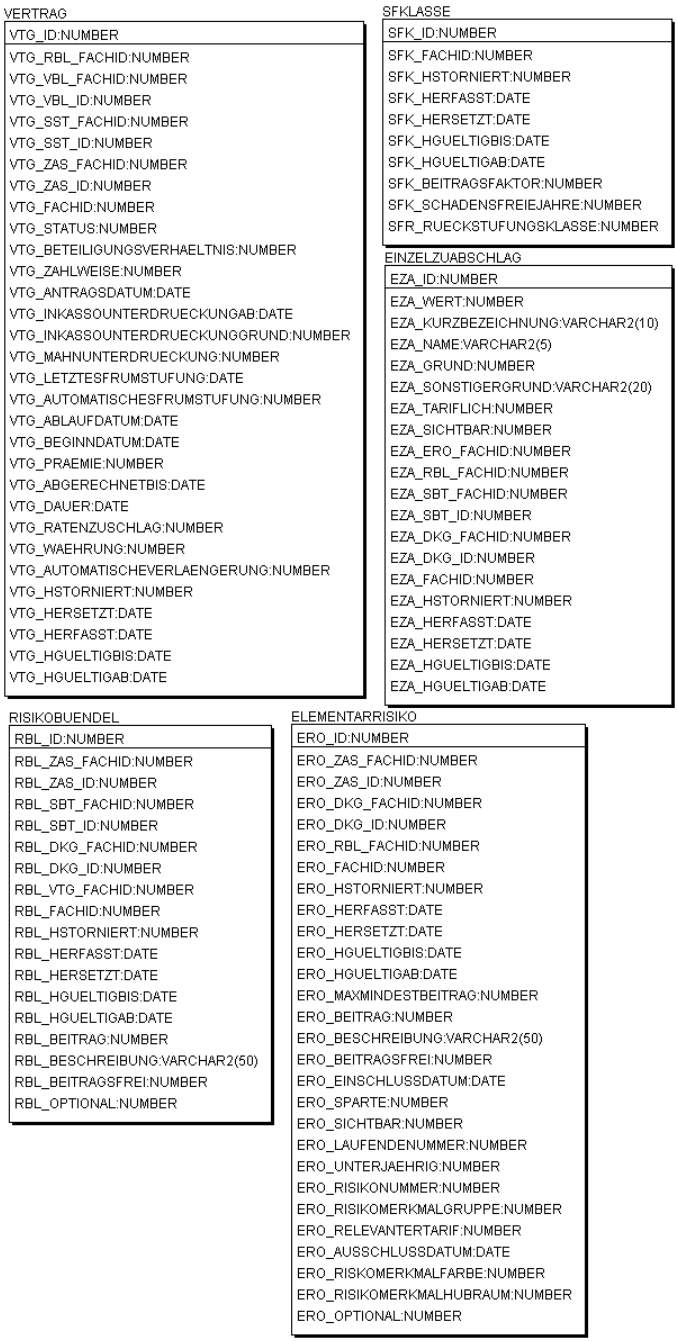


Abbildung 8.3: Ausschnitt des Entwurfs als Datenbankschema

Entwurf gewohnt — sprechende Namen benutzt werden. Das Mapping zwischen beiden Namensräumen wird durch den Deployment Descriptor bestimmt.

8.1.2 Entwurf der Interaktion

Der Entwurf der Interaktion zwischen den Klassen orientiert sich an dem Sequenzdiagramm aus Abbildung 7.5. Dieses Diagramm schreibt vor, dass der Batchlauf auf eine `Sucher`- und eine `Bearbeiter`-Klasse zu verteilen ist.

Für den `Sucher` müssen die Kriterien gefunden werden, nach dem ein Geschäftsobjekt an einer Sfr-Umstufung teilnimmt. Im Einzelnen sind das:

- Ist der Vertrag hauptfällig?
- Ist der Vertrag automatisch umstufbar, d.h. ohne explizite Anweisung des Sachbearbeiters?
- Liegt die letzte Umstufung ein Jahr oder mehr zurück?
- Ist noch eine günstigere als die gegenwärtig zugeordnete `SfKlasse` erreichbar?
- Ist der Vertrag aktuell gültig (Abfrage bzgl. der Historisierungsattribute)?

Da diese Kriterien alle den Vertrag betreffen, erhält er eine Methode `pruefeSfrUmstufung()`, die all dies abfragt. Das Ergebnis ist als Sequenzdiagramm in Abbildung 8.4 dargestellt. Dieses Sequenzdiagramm bezieht sich als Verfeinerung auf das in Abbildung 7.5 dargestellte Sequenzdiagramm. Es wird hier dargestellt, welche Aufrufe innerhalb der Methode `finde()` stattfinden.

Es werden *alle* Vertragsinstanzen im System einmal mit der Methode `pruefeSfrUmstufung()` aufgerufen. Diese Methode delegiert an weitere Methoden des Vertrags. Zuerst wird geprüft, ob die Instanz einen aktuell gültigen Stand darstellt. Danach werden Hauptfälligkeit, automatische Umstufung und letzte Umstufung geprüft. Die günstigste `SfKlasse` ist erreicht, falls die aktuelle Klasse keine Hochstufungsklasse hat. Die Methode `guenstigsteStufeErreicht()` prüft daher die Hochstufungsklasse der aktuellen `SfKlasse` auf `null`. Sind alle Kriterien erfüllt, wird der Vertrag als Teilnehmer mit `add(einVertrag)` in einer `Collection` gespeichert.

Zur Erläuterung der Bearbeitungsfunktionalität wird der zweite Aufruf `bearbeiteGO()` an den `Bearbeiter` in Abbildung 7.5 verfeinert.

Durch die Aktivitätsdiagramme in Abbildung 6.3 und Abbildung 6.4 sind die fachlich notwendigen Schritte bekannt. Die ersten beiden Aktivitäten »ermittleHauptfaelligeVertraege« und »pruefeAutomatischeUmstufbarkeit«werden bereits durch den `Sucher` abgearbeitet. Der `Bearbeiter` muss daher nur noch alle Aktivitäten unterhalb der Entscheidungsraute in Abbildung 6.3 ausführen. Das sind die Schritte »3« und die Verfeinerungen von Schritt »4«. Für jede Aktivität aus Abbildung 6.4 wird entschieden, durch welche Klasse sie am besten ausgeführt werden kann. Dadurch entsteht das Sequenzdiagramm in Abbildung 8.5.

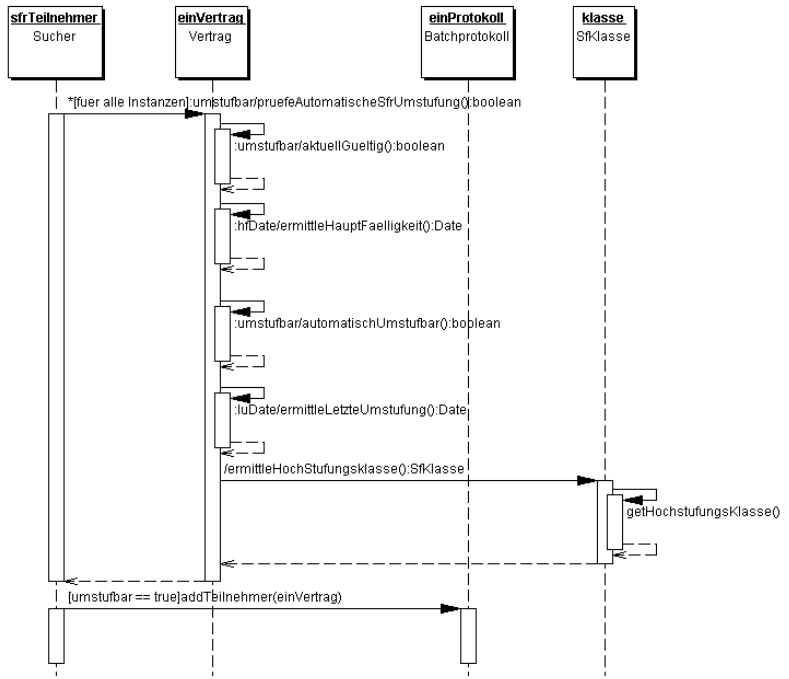


Abbildung 8.4: Sequenzdiagramm: Identifikation der beteiligten Geschäftsobjekte

Im ersten Aufruf `erzeugeHistorienStand()` wird der bisherige Stand des Vertrags gespeichert. Durch `stufeUm()` erhält der Vertrag eine neue `SfKlasse`. Das entspricht Aktivität »3« des genannten Diagramms.

Danach muss der Beitrag anhand des neuen Rabatts berechnet werden. Dazu wird auf jeder Ebene die Menge der relavanten Modifikatoren mit `ermittleZuAbschlaege()` bestimmt. Die ermittelten Modifikatoren werden zur weiteren Berechnung an die darunter liegende Ebene übergeben: `ermittleBeitrag(sfrWert, Collection)`.

Auf Ebene der `RisikoBuendel` werden nach Erzeugung eines Historienstands die dort relevanten Modifikatoren den bereits ermittelten hinzugefügt und mit `ermittleBeitrag(...)` an die `ElementarRisiko`-Instanzen gegeben. Auch diese Instanzen bestimmen weitere Modifikatoren. Schließlich wendet sich das `ElementarRisiko` mit den Attributen für die Risikomerkmale an den für ihn gültigen Tarif zur Beitragsberechnung. Mit `getMaxMindestBeitrag` wird zusätzlich der maximale Mindestbeitrag ermittelt. Anhand aller Modifikatoren, tariflichem Beitrag und Mindestbeitrag wird nun der Beitrag ermittelt. Die Beiträge der `ElementarRisiken` werden zum Beitrag der `RisikoBuendel` und zur Prämie des `Vertrags` addiert. Auf diesen Wert wird die Steuer angewendet.

Aus dem Aktivitätsdiagramm wurden die fachlichen Schritte ermittelt. Die Zuständigkeit und das Zusammenarbeiten der Klassen ist darauf aufbauend in den vorangegangenen Sequenzdiagrammen dargestellt worden. Damit ist der Entwurf der Basisvariante beendet.

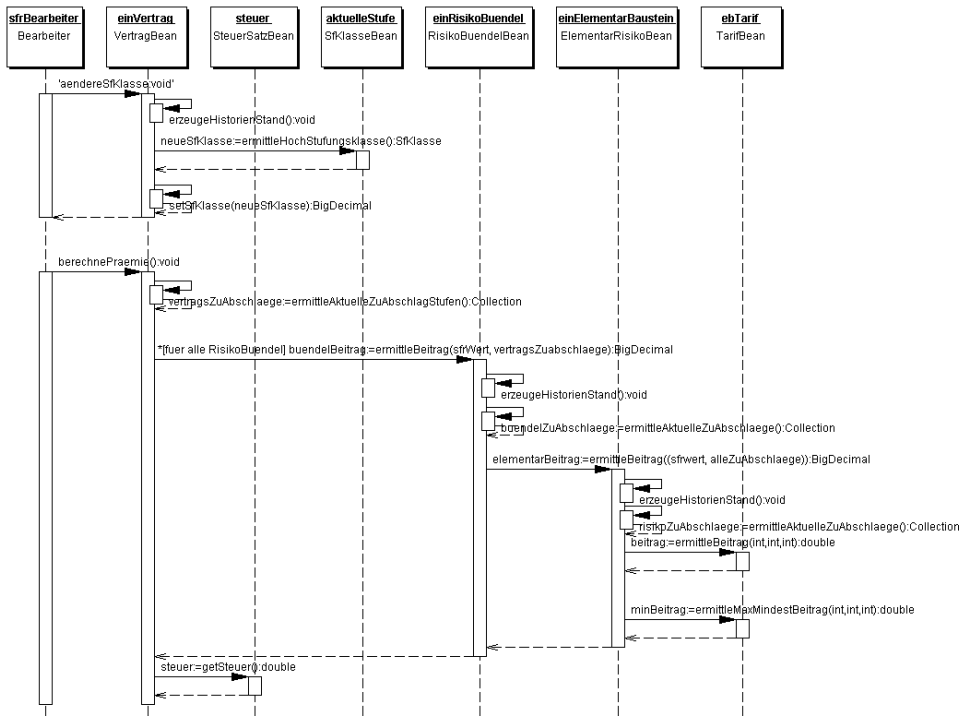


Abbildung 8.5: Sequenzdiagramm: Durchführung einer Umstufung

Als Zusammenfassung der Ergebnisse des dynamischen Entwurfs wird das Klassendiagramm um die im Sequenzdiagramm ermittelten Methoden erweitert. In Abbildung 8.6 sind die gleichen Klassen wie in den anderen Ausschnitten dargestellt. Alle Attribute sind ausgeblendet, nur Methoden, die dem Geschäftsprozess »Sfr-Umstufung« dienen, werden dargestellt. Die Methoden entsprechen den Methoden der vorangegangen Sequenzdiagramme und werden hier nicht erneut erläutert.

Deutlich ist an den Methoden der Einfluss der Historisierung auf den modellierten Geschäftsprozess zu erkennen. Alle Klassen besitzen eine Methode `erzeugeHistorienStand`, die aufzurufen ist, um vor Beginn aller Änderungen den alten Stand speichern zu können.

Weiterhin sind in den Klassen `Vertrag`, `RisikoBuendel` und `ElementarRisiko` Methoden mit dem Namensmuster `ermittleAktuelle[...]` zu finden. Da Beziehungen wie weiter oben beschrieben nur noch über `fachId` und Historisierungsattribute abgebildet werden können, muss bei jedem Zugriff auf referenzierte Klassen der Zweck bedacht werden. Für eine Beitragsberechnung sind nur aktuelle Klassen notwendig; für Auskunftsfunktionalität müssten zusätzliche Methoden `findeAnDatumGueltige[...]` und `findeAnDatumBekannte[...]` implementiert werden.

Auf Ebene der Datenbank befinden sich keine Fremdschlüsselbeziehungen zwischen historisierbaren Klassen zur Navigation. Anstelle dessen werden Suchanfragen mit

mehreren Parametern benutzt. Die Navigation zwischen Instanzen ist von Hand auszuprogrammieren und daher wartungsintensiv. Weniger wartungsintensiv sind Container-Managed Relations. Da sie keine zeitlichen Aspekte berücksichtigen können, muss jedoch auf sie verzichtet werden.

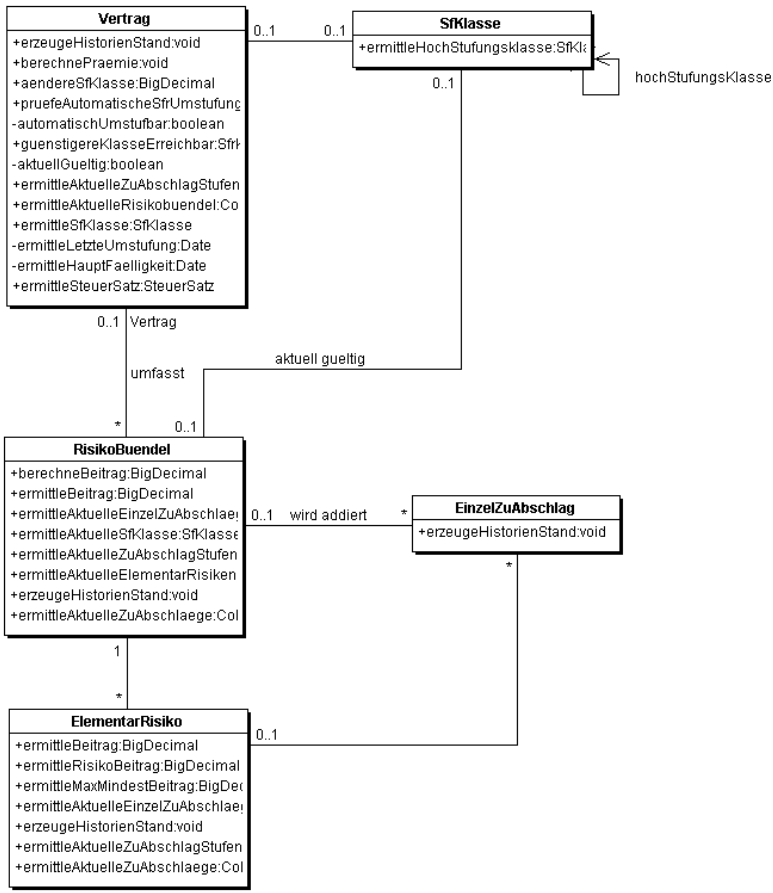


Abbildung 8.6: Klassendiagramm:Ausschnitt mit Methoden

8.1.3 Deployment

Nachdem die Entwicklung der Komponente Vertrag für den Batchlauf beendet worden ist, wird im Folgenden auf das Deployment der Komponente eingegegangen. Unter *Deployment* versteht man das Vorbereiten einer Komponente für den Betrieb in einem Applikationsserver. Dazu gehört das Generieren der Klassen, die den Datenbankzugriff verkapseln bei Container-Managed Persistence, und das Konfigurieren des Laufzeitverhaltens.

Für die Konfiguration des Laufzeitverhaltens bieten Applikationsserver verschiedene Optionen, die die Performanz beeinflussen können. Diese Optionen gehören zumeist proprietär zu den Applikationsservern und ihre Wirkung ist auf anderen Produkten kaum reproduzierbar. Um die Nachvollziehbarkeit der Messungen zu gewährleisten, ist es dennoch wichtig, offen zu legen, mit welchen Deployment-Parametern eine Komponente betrieben worden ist.

Im Folgenden wird vorgestellt, welche wichtigen Parameter beim Deployment der Vertragskomponente im Bea WebLogic 6.1 benutzt wurden. Der BEA WebLogic 6.1 besitzt drei xml-Dateien, die Deployment-Information enthalten. Für die Basisvariante und alle anderen Varianten befinden sie sich auf der beiliegenden CD.

- ejb-jar.xml ist Teil des EJB 2.0-Standards und enthält die dort vorgeschriebenen Strukturen, wie das abstrakte Schema der Bean oder die Transaktionsattribute.
- weblogic-cmp-rdbms-jar.xml enthält Information zur Umsetzung der Container-Managed Persistence in WebLogic 6.1. Laut Spezifikation wird die Verwendung eines eigenen DeploymentDescriptors für CMP vorgeschrieben. Der Inhalt und die Struktur des Deployment Descriptors sind jedoch applikationsserver-spezifisch.
- weblogic-ejb-jar.xml enhält für den WebLogic Applikationsserver spezifische Deploymentinformation. Das sind u.a. Optionen zum Tuning der Applikation. Im Folgenden soll es um einen Überblick über diese Optionen gehen.

Für die Basisvariante sind proprietäre Verbesserungen abgeschaltet worden, da sich nur so der Einfluss einer Designänderung bestätigen lässt. Wenn eine der späteren Varianten von der im Folgenden beschriebenen Konfiguration abweicht, ist dies ausdrücklich im Messprotokoll zu der Variante vermerkt.

Folgende Parameter wurden explizit gesetzt:

- `<idle-timeout-seconds>`

  Dieser Wert gibt an, nach wie vielen Sekunden eine nicht benutzte Instanz passiviert werden soll. Für die Basisvariante werden alle Instanzen nach einer Sekunde passiviert. Bei der Passivierung wird die Instanz serialisiert und z.B. im Plattencache gespeichert [4]. Beim Aktivieren der Instanz muss diese deserialisiert werden. Wird eine Instanz selten gebraucht, ist das Vorgehen sinnvoll, da sie serialisiert weniger Ressourcen einnimmt als im Zustand »idle« im Cache.

  Für häufig benutzte Instanzen bietet es sich jedoch an, sie lange im Cache zu halten. Als Performanzverbesserung könnten die wenigen benutzten Instanzen von `SfKlasse` über längere Zeit nicht passiviert werden. Der Zugriff auf diese Instanzen wird durch das entfallende Serialisieren und Deserialisieren beschleunigt.

- `<enable-call-by-reference>`

  Laut EJB-Standard müssen Parameter eines Methodenaufrufs zwischen Beans per Wert übergeben werden. Dazu wird eine Kopie der Parameter angelegt und serialisiert, um so übertragen zu werden. Sind Aufrufer und Aufgerufener einer Methode in

der gleichen Java Virtual Machine, können die Serialisierungskosten gespart werden. Die Parameter werden dann per Referenz übergeben (engl: call by reference). Bei der Basisvariante werden die Parameter auf die langsame, aber spezifikationskonforme Art übergeben: sie haben den Wert `false`. Abschnitt 8.4 stellt eine Design-Variante zur Verminderung der Serialisierungskosten ausführlich vor.

- `<max-beans-in-cache>`

  Dieser Wert gibt an, wie viele aktive Instanzen maximal in dem Cache gehalten werden dürfen. Für den Verlauf der Messungen wurde der Vorgabewert 100 gewählt.

- `<concurrency-strategy>`

  Dieser Parameter bestimmt, ob Applikationsserver oder Datenbank für das Setzen der Transaktionssperren (»Locking«) zuständig ist. Für die Beispiele des Buchs obliegt das Locking der Datenbank. Diese Konfiguration erfordert mehr Erfahrung mit der Datenbank und bietet mehr Tuning-Möglichkeiten.

- `<delay-updates-until-end-of-tx>`

  Mit diesem Parameter kann bestimmt werden, wann die auf Ebene der Beans veränderten Attribute in der Datenbank aktualisiert werden. Hat der Parameter den Wert `true`, so wird erst zum Ende einer Transaktion die Datenbank aktualisiert. Das verkompliziert die Programmierung, da auf Zwischenergebnisse aus der Datenbank nicht zugegriffen werden kann. Andererseits wird die Abarbeitung beschleunigt, da nur *eine* Datenbankverbindung erstellt wird, um *alle* Änderungen zu übermitteln. Für fast alle Messläufe hat der Parameter den Wert `false`, d.h. für jedes geänderte Attribut wird eine neue Datenbankverbindung erstellt. Eine Transaktion erstellt und schließt also mehrere Datenbankverbindungen. Um große Änderungsaufwände zu vermeiden, muss die *Wahl des Parameters von Anfang an bei der Programmierung berücksichtigt* werden. Die finale Variante zeigt den Einfluss dieses Parameters auf die »processing time« (vgl. Unterkapitel 8.7).

- `<db-is-shared>`

  Dieser Parameter gibt an, ob neben dem Applikationsserver weitere Prozesse die Daten in der Datenbank ändern können. Ist dies der Fall, so wird vor jedem Zugriff auf eine Bean die Belegung der Attribute aus der Datenbank ermittelt. Da es bei großen produktiven System unwahrscheinlich ist, alleiniger Nutzer des Datenbestands zu sein, hat der Parameter den Wert `true`. Von diesem Parameter ist auch die Kontrolle der Nebenläufigkeit abhängig. Kontrolliert der Applikationsserver die Nebenläufigkeit, so werden Sperren auf Datenbankzeilen stets exklusiv angelegt. Kontrolliert die Datenbank die Nebenläufigkeit, so muss `<db-is-shared>` den Wert true haben. Für einen höheren Grad an Parallelität kann nun aber der Isolierungsgrad vermindert werden, sodass mehrere Transaktionen gleichzeitig einen Datensatz lesen können.

- `<finders-load-bean>`

  Beim Ausführen einer Find Method wird zuerst mit einem Datenbankzugriff eine Treffermenge erstellt. Beim Aufruf eines Elements der Treffermenge wird mit einem

weiteren Zugriff das Element geladen. Um alle Elemente zu laden, werden (1 + Mächtigkeit der Treffermenge) Zugriffe benötigt. Hat der Parameter den Wert `true`, werden alle Beans mit dem ersten Datenbankzugriff geladen. Das kann sinnvoll sein bei kleinen Treffermengen, deren Instanzen sofort im Anschluss bearbeitet werden. Da es sich um einen proprietären Verbesserungsmechanismus handelt, wurde er abgeschaltet. Der Parameter hat den Wert `false`.

8.1.4 Messergebnis

Zur Ermittlung der Messergebnisse ist der beschriebene Entwurf implementiert worden. Für das Deployment wurden die Parameter laut Abschnitt 8.1.3 gewählt. Die Tabelle in Abbildung 8.7 zeigt als Ergebnis eine *katastrophale Performanz*.

Methoden	teilnehmende Verträge // nicht teilnehmende Verträge				
	10 // 0	10 // 10	10 // 100	10 // 1200	100 // 12000
findeAlleVertraege	00:00:00.0	00:00:00.0	00:00:00.2	00:00:01.2	
pruefeVertrag	00:00:00.2	00:00:00.5	00:00:03.8	00:01:31.1	
pruefeStufe	00:00:00.3	00:00:00.5	00:00:03.0	00:00:58.7	
aendereSfKlasse	00:00:00.6	00:00:00.6	00:00:00.8	00:00:02.6	
historisiereVertrag	00:00:00.8	00:00:00.8	00:00:01.1	00:00:04.0	
findeSFKlasse	00:00:00.9	00:00:00.8	00:00:01.1	00:00:03.6	
ermittleZuAbschlagStufen	00:00:03.0	00:00:02.5	00:00:02.6	00:00:07.0	
ermittleSteuersatz	00:00:02.0	00:00:04.6	00:00:27.2	00:00:53.4	
findeRisikoBuendel	00:00:18.4	00:00:31.4	00:02:33.7	01:17:03.1	Abbruch: OutOfMemory
historisiereRisikoBuendel	00:00:03.1	00:00:02.8	00:00:02.7	00:00:11.1	
findeEinzelZuAbschlaege	00:01:40.4	00:03:16.2	00:18:06.6	08:43:06.0	
findeElementarRisiken	00:03:14.2	00:05:36.3	00:27:14.2	13:27:37.9	
historisiereElementarRisiko	00:00:09.1	00:00:08.4	00:00:08.8	00:00:38.8	
ermittleMaxMindestBeitrag	00:00:06.3	00:00:06.0	00:00:05.1	00:00:14.1	
ermittleBeitrag	00:00:06.3	00:00:06.0	00:00:06.0	00:00:13.8	
berechnePraemie	00:00:14.9	00:00:15.6	00:00:16.6	00:00:49.5	
processing time	**00:06:03.4**	**00:10:16.0**	**00:49:17.2**	**23:33:43.9**	0

Abbildung 8.7: Gesamtdauer der Methoden in Abhängigkeit der Anzahl nicht *teilnehmender Verträge*

Die Tabelle ist aufgebaut nach dem in Abschnitt 7.2 vorgestellten Schema. Über den Spalten ist vermerkt, wie viele Verträge am Batchlauf teilgenommen haben (linke Zahl vor den Schrägstrichen //) und wie viele Verträge als Fülldaten in der Datenbank waren (rechte Zahl). Ein »Vertrag« meint in diesem Zusammenhang das Geschäftsobjekt, also die baumartige Struktur aus einzelnen Bausteinen, die in Abbildung 7.2 dargestellt ist.

Bei 10 teilnehmenden Verträgen und 1200 nicht teilnehmenden Verträgen dauert die Bearbeitung fast 24 Stunden! Für 100 teilnehmende Verträge und 12 000 nicht teilnehmende Verträge stürzte das System wegen Speichermangels ab. Diese Zeit für 100 teilnehmende Verträge war als Referenzzeit gedacht. Durch das Scheitern dieser Messung wird eine andere Variante als Referenzzeit dienen müssen.

Erstaunlich ist, dass die processing time mit der Anzahl der *nicht* teilnehmenden Verträge wächst. So dauert es wesentlich länger 10 Verträge zu bearbeiten, wenn 1200 Fülldaten

vorhanden sind, als 10 Verträge zu bearbeiten, wenn keine *nicht teilnehmenden* Verträge zusätzlich in der Datenbank sind.

Die Darstellung der Messwerte als Säulendiagramm in Abbildung 8.8 verdeutlicht, welche Methoden am meisten Zeit verbrauchen. Jede Säule repräsentiert einen Messlauf. Die ganze Säule entspricht 100 % der Gesamtzeit (processing time). Jedes Segment einer Säule entspricht in seiner Höhe dem Anteil an der processing time.

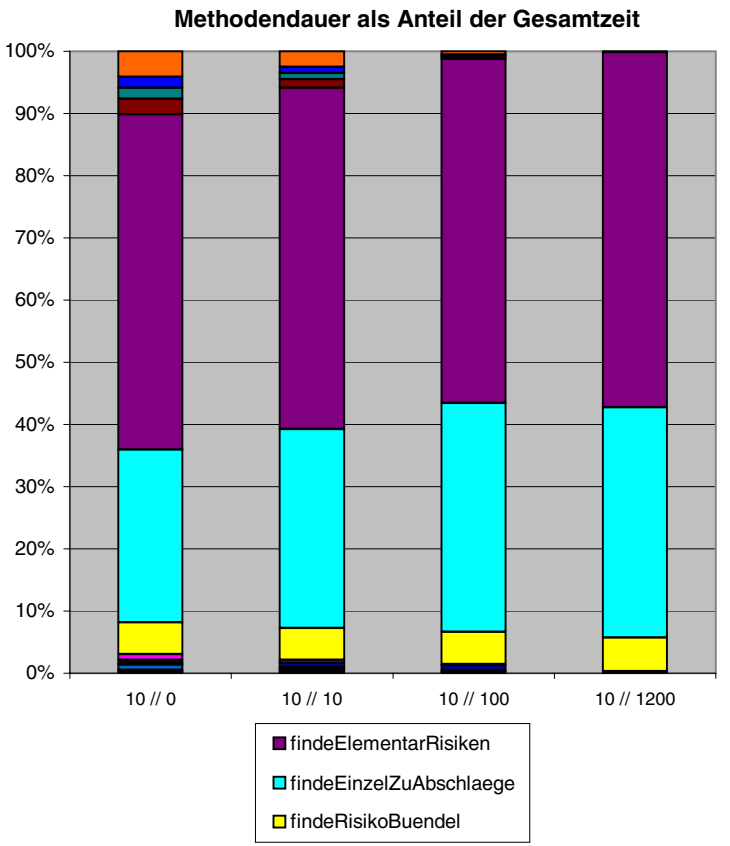


Abbildung 8.8: Anteil der Methoden an processing time

In der Legende sind aus Gründen der Übersichtlichkeit nur die Messpunkte mit den größten Anteilen vermerkt: findeElementarRisiko, findeEinzelZuAbschlaege, findeRisikoBuendel. Sie machen in allen Messläufen über 80 % der »processing time« aus. Das alleine sagt noch nicht viel aus. Es könnte sein, dass die gemessenen Methoden pro Aufruf wenig Zeit brauchen und erst durch die Aufrufhäufigkeit zu so einem großen Anteil kommen. Deshalb zeigt Abbildung 8.9 eine Tabelle, in der die Zeit pro einzelnem Aufruf dargestellt ist. Die Aufrufhäufigkeiten der Methoden sind in Abschnitt 7.2 dokumentiert.

In dieser Tabelle wird deutlich, dass die Funktionalität zum Suchen von Instanzen *pro Aufruf* teurer ist als alle anderen Schritte. Durch die Darstellung der Tabelle als Säu-

Methoden	teilnehmende Verträge // nicht teilnehmende Verträge				
	10 // 0	10 // 10	10 // 100	10 // 1200	100 // 12000
findeAlleVertraege	00:00:00.0	00:00:00.0	00:00:00.2	00:00:01.2	
pruefeVertrag	00:00:00.2	00:00:00.5	00:00:03.8	00:01:31.1	
pruefeStufe	00:00:00.3	00:00:00.5	00:00:03.0	00:00:58.7	
aendereSfKlasse	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.3	
historisiereVertrag	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.4	
findeSFKlasse	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.4	
ermittleZuAbschlagStufen	00:00:00.0	00:00:00.0	00:00:00.0	00:00:00.1	
ermittleSteuersatz	00:00:00.2	00:00:00.5	00:00:02.7	00:00:05.3	
findeRisikoBuendel	00:00:01.8	00:00:03.1	00:00:15.4	00:07:42.3	
historisiereRisikoBuendel	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.4	
findeEinzelZuAbschlaege	00:00:00.9	00:00:01.8	00:00:09.9	00:04:45.3	
findeElementarRisiken	00:00:06.5	00:00:11.2	00:00:54.5	00:26:55.3	
historisiereElementarRisiko	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.5	
ermittleMaxMindestBeitrag	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.2	
ermittleBeitrag	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.2	
berechnePraemie	00:00:00.2	00:00:00.2	00:00:00.2	00:00:00.6	
processing time	**00:00:36.3**	**00:01:01.6**	**00:04:55.7**	**02:21:22.4**	0

Abbruch: OutOfMemory

Abbildung 8.9: Dauer pro Aufruf in Abhängigkeit von der Anzahl nicht *teilnehmender Verträge*

lendiagramm in Abbildung 8.10 sticht dies sofort ins Auge. Die Darstellung der Säulen wurde dreidimensional gewählt, da anders die kleineren Messwerte nicht erkennbar werden. Auf der Rechtsachse sind die in Abschnitt 7.2 vorgestellten Messpunkte verzeichnet. Säulen einer Farbe gehören zu einem Messlauf. Auf der Hochachse ist die Zeit angetragen, die ein einzelner Methodenaufruf benötigt.

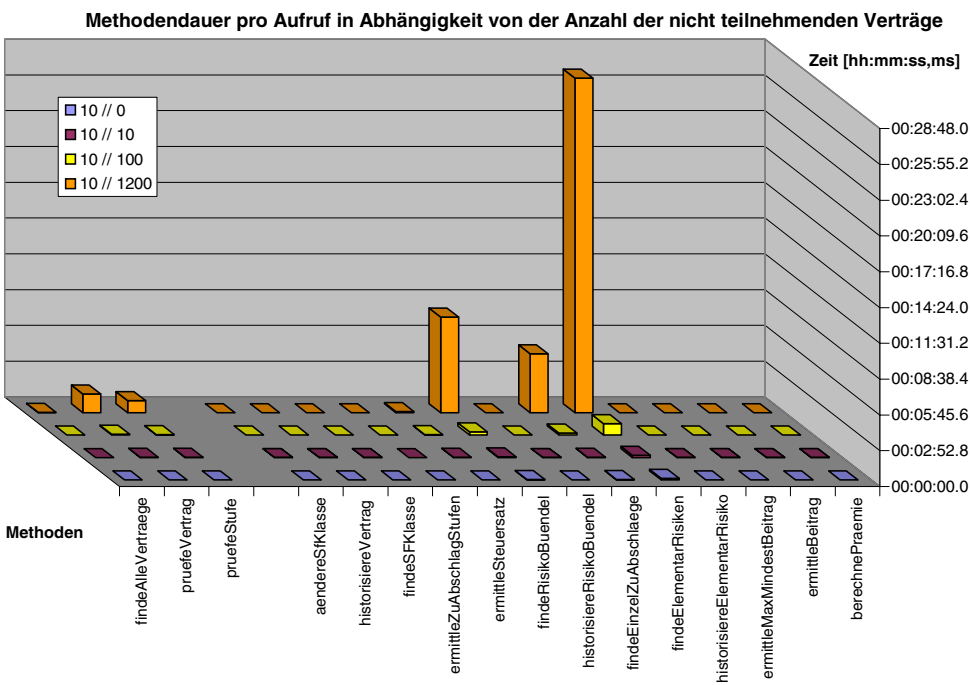


Abbildung 8.10: Dauer einzelner Methoden pro Aufruf in Abhängigkeit von der Anzahl nicht teilnehmender Verträge

Es zeigt sich deutlich, dass keine anderen Aufrufe so teuer sind, wie findeRisikoBuendel, findeEinzelZuAbschlaege, findeElementarRisiken und der in seine drei Einzelschritte zerlegte `Sucher`. Die Einzelschritte des `Suchers` sind die drei Messpunkte links im Diagramm: findeAlleVerträge, prüfeVertrag und pruefeStufe.

Alle Suchmethoden sind dem Aufbau nach gleich: zuerst werden alle Instanzen aus der Datenbank geladen, um dann auf Ebene der Beans traversiert zu werden. Genügt eine Instanz den Suchkriterien, so wird sie einer Ergebnismenge hinzugefügt.

An der Zerlegung der Sucherfunktionalität lässt sich sehr gut zeigen, welche Teilschritte der Suchfunktionalität teuer sind. Am wenigsten Zeit benötigt stets der Aufruf für findeAlleVerträge. Dieser Aufruf fordert lediglich alle Instanzen aus der Datenbank und stellt als Ergebnis eine Menge von Remote-Referenzen zusammen. Sobald auf eine dieser Referenzen zugegriffen wird, werden die Attribute der Bean durch einen weiteren Datenbankzugriff nachgeladen. Dieser Zugriff erfolgt im Schritt pruefeVertrag. Weitere Zugriffe folgen im Schritt pruefeStufe.

8.1.5 Bewertung

Das Design der Basisvariante zeigt, wie man EJB-Applikationen besser *nicht* umsetzt. Die katastrophale Performanz entsteht durch das Laden unnützer Beaninstanzen in den Applikationsserver. Jede Instanz wird erst einmal geladen, um danach zu entscheiden, ob sie bearbeitet wird oder nicht. Es entsteht eine *riesige Zahl von unnützen Zugriffen auf die Datenbank*, nur um festzustellen, dass die geladene Instanz nicht benötigt wird.

Dieser Design-Fehler tritt überall dort zu Tage, wo über einer Ergebnismenge nachträglich nochmals iteriert wird, um die Instanzen zu finden, »die man eigentlich braucht«.

Beim *Traversieren von Relationen* über mehrere Zwischenglieder wird dieser Fehler ebenfalls häufig gemacht. Werden beispielsweise vom `Vertrag` aus die `ElementarRisiko`-Instanzen zu allen `RisikoBuendel`-Instanzen gesucht, so kostet es unnötig Performanz, zuerst alle `RisikoBuendel` zu laden, wenn diese nicht weiter verändert werden sollen. Anstelle dessen sollten durch einen besonderen Finder die `ElementarRisiko`-Instanzen *in der Datenbank ermittelt* und allein die Instanzen, die wirklich zu bearbeiten sind, in den Applikationsserver geladen werden.

8.2 Sucherimplementierung mit EJB QL

Die naive Implementierung in Kapitel 8.1 hat eindrucksvoll gezeigt, wie inperformant das Ausprogrammieren einer Suchanfrage auf Ebene der Beans ist. Die folgende Implementierungsvariante ist dadurch gekennzeichnet, dass die Zusammenstellung der Treffermenge bereits auf Ebene der Datenbank vorgenommen wird. Mit Bezug auf den Lebenszyklus einer BeanInstanz wird erläutert, worin die Ersparnis des Ansatzes liegt.

EJB QL ist als Anfragespezifikationssprache [18, S. 217] gedacht, die in eine Zielsprache der angebundenen Datenbank übersetzt wird. Dadurch macht sich die EJB-Architektur

unabhängig von den SQL-Dialekten relationaler Datenbanken, bzw. Anfragesprachen objektorientierter oder hierarchischer Datenbanksysteme. Für die Unabhängigkeit der Applikation von der Persistenzschicht ist die EJB QL von großer Bedeutung.

8.2.1 Literatur

Performanz lässt sich gewinnen, indem Daten bereits in der Datenbank verarbeitet werden. Das nutzlose Umherschieben von Information soll vermieden werden: »Most serious performance problems in DBMS applications come from moving raw data around needlessly« [3].

Bei der Klärung der Frage, wo das Vorgehen der naiven Variante unnütz Daten bewegt, muss der Lebenszyklus (siehe [18, S. 247ff]) einer Bean-Instanz unter der Fragestellung betrachtet werden, welche Zustände eine Bean durchläuft, wenn sie Teil der Ergebnismenge einer Suchanfrage ist.

Bei einer Suchanfrage wird als Erstes eine Ergebnismenge in der Datenbank ermittelt. Zu dieser Ergebnismenge erstellt der Container eine Menge entsprechender Remote-Referenzen und reicht sie als Collection oder Enumeration nach außen. Bisher wurden noch keine Bean-Instanzen erzeugt. Auf der Datenbank ist genau eine Suchanfrage ausgeführt worden. Wird auf den als Ergebnismenge herausgegebenen Remote Interfaces eine Methode angesprochen, so muss die zugehörige Beaninstanz im Container diese Methode ausführen. Dazu wird eine neue Instanz erzeugt und ihr mit `setEntityContext()` Information über ihre Umgebung gegeben. Die Instanz befindet sich nun im Zustand »pooled«. Durch ein `ejbActivate()` wird der Instanz ihre Identität zugewiesen; erst jetzt hat sie Bezug zu einer identifizierbaren Zeile einer Datenbanktabelle. Vor dem Abarbeiten einer Methode wird schließlich mit `ejbLoad()` die Belegung der Attribute aus der Datenbank in die Bean gelesen.

Zu diesem Vorgang ist zweierlei anzumerken.

1. Ist der Applikationsserver nicht alleiniger Nutzer der Datenbank, so ist `ejbLoad()` nicht nur zum initialen Laden der Attribute aufzurufen, sondern auch zum Beginn jeder Transaktion, sodass Bean und Datenbank abgeglichen werden. Für jedes `ejbLoad()` ist also ein Datenbankzugriff in form eines `select` notwendig.

2. Um zu vermeiden, dass stets eine neue Instanz zu erzeugen ist, kann der Container einen Pool von Instanzen unterhalten. Alle Instanzen im Pool werden als gleich betrachtet. Sie haben keinen Bezug zu einer bestimmten Zeile einer Tabelle. Ob ein Pool unterhalten wird oder nicht, ist *nicht* durch den Standard vorgeschrieben [18, S. 249].

Nach dem Überblick, wie eine Treffermenge im Applikationsserver zusammengestellt wird, erläutert das folgende Kapitel, wo im konkreten Beispiel Datenbankzugriffe vergeudet werden.

Für die verbesserte Variante soll die mit EJB 2.0 eingeführte Anfragesprache »EJB QL« genutzt werden [18, S. 217ff]. Eine Einführung in diese Anfragesprache wurde bereits in

Abschnitt 4.4 gegeben. Der folgende Abschnitt beschränkt sich daher auf die Erläuterung der konkreten Anfragen für das umzusetzende Beispiel.

Durch Datenbankwerkzeuge kann die SQL-Anfrage, die der Applikationsserver generiert, analysiert werden. In [54] und [27] ist die Analyse von SQL-Anfragen erläutert. Dort werden Verbesserungen durch z.B. Indexerstellung und Partitionierung empfohlen.

Eine wichtige Hilfe für die Analyse ist die Möglichkeit vieler Applikationsserver, die an die Datenbank gerichteten Anfragen mitprotokollieren zu können. Dieses Protokoll kann auf zwei Arten genutzt werden:

1. **Zugriffspfade ermitteln:** Die einzelne Anfrage kann durch zusätzliche Datenbankwerkzeuge analysiert werden. Dadurch kann die Zeit ermittelt werden, die die *Datenbank* zur Abarbeitung der Anfrage benötigt. Die Zeit, die *allein* der Applikationsserver benötigt, wird ermittelt, indem von der gemessenen Zeit die Zeit für die Datenbankanfrage abgezogen wird. Durch Vergleich der einzelnen Zeiten lässt sich entscheiden, ob eine Verbesserung auf Ebene der Datenbank oder auf Ebene des Applikationsservers vorzunehmen ist.

   Außerdem kann eine detaillierte Analyse aufzeigen, welche Tabellen in welchem Umfang zur Bearbeitung der Anfrage benutzt werden. Durch Umstellen der Anfrage und den Einsatz eines Index kann dieser Zugriffspfad verbessert werden. Die Verbesserung lässt sich durch eine weitere Analyse validieren, ohne dazu den Applikationsserver zu benötigen.

2. **Algorithmische Fehler:** Das Protokoll der Datenbankanfragen kann dazu dienen, algorithmische Fehler aufzudecken. Wird mehrfach innerhalb einer fachlichen Methode eine identische Anfrage mit identischer Treffermenge gestellt, kann das ein Indiz dafür sein, dass diese Anfrage an der falschen Stelle im Algorithmus vorgenommen wird. Durch Umstellen des Algorithmus wird erreicht, dass diese Anfrage nur einmal an die Datenbank gerichtet wird. Ihr Ergebnis wird als Parameter an untergeordnete Aufrufe übermittelt. Ebenso lässt sich mit `UPDATE`-Befehlen verfahren. Wird mehrfach auf einer Instanz ein `UPDATE` aufgerufen, so ist zu prüfen, ob der Algorithmus sich so umstellen lässt, dass alle Änderungen durch nur ein `UPDATE` in die Datenbank geschrieben werden.

Die Analyse und Optimierung von SQL-Anweisungen ist weithin bekannte Praxis. Das vorliegende Buch beschränkt sich daher auf das Implementieren einer einfachen Anfrage, ohne tiefer auf Optimierungsmöglichkeiten einzugehen.

8.2.2 Übertragung und Entwurf

Nachdem der Ablauf einer Suchanfrage im Applikationsserver skizziert worden ist, können die Kosten für die `Sucher`-Implementierung der Basisvariante ermittelt werden. Als Kostenmaß soll die Anzahl der Datenbankzugriffe genutzt werden. Ein Datenbankzugriff umfasst das Erstellen einer Verbindung zur Datenbank, das Ausführen eines SQL-Statements auf Instanzen mindestens einer Tabelle der Datenbank, das Umsetzen der

Ergebnismenge in Objekte der EJB-Architektur und schließlich das Beenden der Datenbankverbindung mit allen Schritten, die für die Transaktionslogik notwendig sind. Detaillierte Interaktionsdiagramme sind unter [18, S. 208ff; S. 281ff] nachzulesen.

Die Basisvariante fordert zuerst alle Instanzen aus der Datenbank an. Dafür ist ein Datenbankzugriff notwendig. Im Anschluss wird über alle Instanzen iteriert und geprüft, ob sie den Kriterien zur Teilnahme am Geschäftsprozess genügen. Vor dem Aufruf der Methode `pruefeAutomatischeSfrUmstufrung()` wird, wie oben erläutert, eine Bean-Instanz angelegt und diese mit `ejbActivate()` und `ejbLoad()` zur Ausführung der Methode vorbereitet. Jeder Aufruf von `pruefeAutomatischeSfrUmstufrung()` bewirkt damit einen Datenbankzugriff. Die Anzahl der Datenbankzugriffe wächst linear mit der Anzahl der Instanzen.

Wird per SQL oder EJB QL auf der Datenbank gesucht, so wird demgegenüber nur genau ein Datenbankzugriff benötigt. Innerhalb dieses Zugriffs betrachtet die Datenbank alle enthaltenen Instanzen und ermittelt eine Ergebnismenge. Datenbanken sind für diese Art von Anfragen optimiert und funktionieren daher wesentlich schneller als ausprogrammierte Suchen auf Ebene des Appliaktionsservers. Aus der Ergebnismenge der Datenbank werden Remote-Referenzen erstellt und als Collection zurückgegeben. Diese Art der Anfrage hat damit eine *konstante Anzahl von Datenbankzugriffen*.

Es lässt sich entgegenhalten, dass im nächsten Schritt bei einer Sucherimplementierung mit EJB QL ein `ejbLoad()` erfolgen muss, wo hingegen bei der naiven Implementierung schon alle Instanzen im Cache vorliegen. Dieses Gegenargument besagt damit, dass die Ersparnis der EJB QL-Variante wieder verloren ist. Mehrere Gründe widersprechen ihm:

1. Nicht alle Applikationsserver implementieren ein Caching. Ist die Applikation auf ein Cluster von Rechnern verteilt, so verringert sich auch der Effekt des Caching.
2. Ist der Applikationsserver nicht ausschließlicher Nutzer der Datenbank, so wird zur Synchronisation zu Beginn einer Transaktion ohnehin `ejbLoad()` aufgerufen. Die naive Variante hätte also bereits den zweiten Datenbankzugriff *pro Instanz*.
3. Durch die historisierten Vertragsstände ist der Anteil der teilnehmenden Verträge gegenüber den nicht teilnehmenden gering. Er ist in Abschnitt 7.1 mit *»1 zu 120«* beziffert worden. Für die Mehrzahl der Instanzen wird damit `ejbLoad()` unnötig aufgerufen; diese Instanzen werden nicht im Applikationsserver gebraucht. Jede dieser Instanzen erzeugt ein unnötiges `select` auf der Datenbank.

Damit ist gezeigt, dass selbst bei der Nutzung von Caches das Ausprogrammieren der Ermittlung von Treffermengen auf Ebene der Anwendung die Performanz massiv verschlechtert.

Neben der Implementierung der Sucherfunktionalität kommen viele weitere verbesserungswürdige Suchanfragen in dem Geschäftsprozess vor. Alle Funktionen zur Navigation der Relationen unter Berücksichtigung der Historisierung müssen per EJB QL umgesetzt werden. Der folgende Abschnitt beschreibt nur die Umsetzung des `Sucher`. Gleichwohl sind für den Messlauf alle Suchmethoden per EJB QL umgesetzt worden.

Als Übertragung auf den Entwurf bleibt nur noch die EJB QL-Anfrage zu formulieren. Im Home Interface des `VertragsBean` ist dazu eine Methode namens

`findeAlleSfrUmstufungTeilnehmer` zu deklarieren, die zu einem Hauptfälligkeitszeitraum (von `monatsAnfang` bis `monatsEnde`) und einer letzten SfrUmstufung (`monatsEndeMinusEinJahr`) eine Collection von Remote Interfaces liefert:

```
public abstract Collection findeAlleSfrUmstufungTeilnehmer(
                                Date monatsAnfang,
                                Date monatsEnde,
                                Date monatsEndeMinusEinJahr
                            )
    throws RemoteException, FinderException;
```

Listing 8.1: Deklaration eines Finder im Home Interface

Die Umsetzung per EJB QL erfolgt deklarativ im Deployment Descriptor **ejb-jar.xml**. In der Bean selbst wird keine Implementierung angegeben.

Im Deployment Descriptor wird gemäß der EJB QL-Syntax die Abfrage formuliert (siehe dazu im Detail: [18, S. 217ff]). Für das vorliegende Beispiel ergibt sich der folgende Block im Deployment Descriptor:

```
<query>
<query-method>
  <method-name>findeAlleSfrUmstufungTeilnehmer</method-name>
  <method-params>
    <method-param>java.sql.Date</method-param>
    <method-param>java.sql.Date</method-param>
    <method-param>java.sql.Date</method-param>
  </method-params>
</query-method>
<ejb-ql>
  <![CDATA[
    SELECT OBJECT(v) FROM VertragBean v
        WHERE (
                v.hGueltigBis > ?2
            AND v.sfKlasse is not null
            AND v.sfKlasse.hochStufungsKlasse is not null
            AND automatischeSfrUmstufungBW = 1
            AND v.abgerechnetBis >= ?1
            AND v.abgerechnetBis <= ?2
            AND v.letzteSfrUmstufung < ?3
        )
  ]]>
</ejb-ql>
</query>
```

Listing 8.2: Deklaration eines Finder per EJB QL

Im oberen Teil des Blocks sieht man, dass der Methodenname aus dem HomeInterface in den Deployment Descriptor zu übernehmen ist, um den Zusammenhang zwischen der Deklaration im HomeInterface und dem Block im Deployment Descriptor herzustellen. Darunter ist dann Typ und Anzahl der Methodenparameter angegeben. Dadurch ergibt sich die Möglichkeit, mehrere Finder mit gleichem Namen, aber verschiedener Anzahl Parameter zu erstellen (Überladen von Methoden).

Der untere Teil des Blocks enthält die eigentliche EJB QL-Anfrage. Abschnitt 4.4 erläutert die grundsätzliche Struktur solcher Anfragen. Das dargestellte Beispiel erschließt sich dadurch wie folgt. Gesucht werden die Objekte v aus dem im Deployment Descriptor definierten Schema `VertragBean`, die *alle* der folgenden Bedingungen genügen (vgl. Abschnitt 6.2):

- `v.hGueltigBis > ?2`

  Der zweite Parameter der Suchanfrage ist `monatsEnde`, wie in der Deklaration im HomeInterface zu sehen ist. Durch diese Abfrage wird geprüft, ob der Vertrag überhaupt aktuell gültig ist.

- `v.sfKlasse is not null`

  Im Deployment Descriptor ist eine CMR (Container-Managed Relation) zwischen `Vertrag` und `SfKlasse` definiert. Daher kann auf diese Art auf die dem `Vertrag` assoziierte `SfKlasse` zugegriffen werden, um zu prüfen, ob sie `null` ist. Fachlich gesehen prüft diese Anfrage, ob auf den `Vertrag` überhaupt ein Schadenfreiheitsrabatt angewendet wird.

- `v.sfKlasse.hochStufungsKlasse is not null`

  Mit diesem Teil der Anfrage wird geprüft, ob der Vertrag bereits die günstigste Klasse erreicht hat. Es werden dazu zwei im Deployment Descriptor deklarierte Relationen traversiert: die Relation zwischen `Vertrag` und `SfKlasse` und die Relation von `SfKlasse` zu `SfKlasse`, nach der zu jeder `SfKlasse` eine Hochstufungsklasse gehören kann.

- `automatischeSfrUmstufungBW = 1`

  Es wird abgefragt, ob der `Vertrag` überhaupt ohne explizite Anweisung durch einen Sachbearbeiter umstufbar ist.

- `v.abgerechnetBis <= ?2`

  Diese Bedingung erklärt sich nur, wenn sie zusammen mit der folgenden Bedingung betrachtet wird. Das Attribut `Vertrag.abgerechnetBis` soll zwischen der Parametern `monatsAnfang` und `monatsEnde` liegen. Dadurch wird die Hauptfälligkeit des Vertrags geprüft.

- `AND v.abgerechnetBis >= ?1`

  Siehe vorangegangene Bedingung.

- `AND v.letzteSfrUmstufung < ?3`

  Die letzte Sfr-Umstufung soll ein Jahr oder weiter in der Vergangenheit liegen.

Wichtiger Vorteil dieser Umsetzung ist, dass die Unabhängigkeit gegenüber der Datenbank bewahrt bleibt. Ändert sich etwa das Namensschema der Datenbanktabellen und -spalten von der Entwicklungsumgebung zum produktiven System, so ist in der Umsetzung mit EJB QL nur der Deployment Descriptor bezüglich des Mapping zwischen Attributen der Bean und Spalten der Datenbank anzupassen. Es muss nicht neu übersetzt werden.

Bei einer Umsetzung mit SQL ist im gleichen Fall *jedes* SQL-Statement anzupassen. Der Code muss anschließend neu übersetzt und getestet werden. Dadurch entstehen *erhebliche Portierungskosten*, falls für eine Komponente verschiedentlicher Einsatz geplant ist.

8.2.3 Messergebnis

Anhand der Messergebnisse kann geklärt werden, ob eine Verbesserung eingetreten ist. Falls die processing time für die Messläufe nicht mehr maßgeblich abhängig ist von der Anzahl nicht teilnehmender Instanzen, kann das Design der vorangegangenen Variante als verbessert gelten. Abbildung 8.11 zeigt die gemessenen Werte für die Designvariante mit EJB QL.

Methoden	teilnehmende Verträge // nicht teilnehmende Verträge				
	10 // 0	10 // 10	10 // 100	10 // 1200	100 // 12000
Sucher	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.1	00:00:00.4
aendereSfKlasse	00:00:00.6	00:00:00.4	00:00:00.4	00:00:00.3	00:00:01.2
historisiereVertrag	00:00:01.0	00:00:01.2	00:00:01.1	00:00:01.0	00:00:05.5
findeSFKlasse	00:00:01.2	00:00:01.4	00:00:01.4	00:00:01.3	00:00:01.4
ermittleZuAbschlagStufen	00:00:03.2	00:00:03.1	00:00:03.3	00:00:03.0	00:00:07.4
ermittleSteuersatz	00:00:00.3	00:00:00.2	00:00:00.2	00:00:00.3	00:00:01.5
findeRisikoBuendel	00:00:00.3	00:00:00.4	00:00:00.4	00:00:00.4	00:00:09.3
historisiereRisikoBuendel	00:00:02.8	00:00:03.0	00:00:02.9	00:00:02.6	00:00:12.0
findeEinzelZuAbschlaege	00:00:06.7	00:00:07.5	00:00:07.3	00:00:07.9	00:12:10.4
findeElementarRisiken	00:00:01.3	00:00:01.5	00:00:01.4	00:00:01.8	00:12:10.3
historisiereElementarRisiko	00:00:10.0	00:00:10.8	00:00:10.2	00:00:10.1	00:00:36.8
ermittleMaxMindestBeitrag	00:00:06.4	00:00:06.6	00:00:06.3	00:00:06.2	00:00:46.9
ermittleBeitrag	00:00:06.0	00:00:06.1	00:00:06.0	00:00:06.5	00:00:46.9
berechnePraemie	00:00:05.4	00:00:04.2	00:00:04.7	00:00:05.5	00:00:05.6
processing time	**00:01:02.2**	**00:01:08.0**	**00:01:04.2**	**00:01:05.5**	**00:27:26.3**

Abbildung 8.11: Gesamtdauer der Methoden in Abhängigkeit der Anzahl nicht teilnehmender Verträge

In der Tabelle sind fünf Messungen dargestellt. Die ersten vier Messungen haben 10 teilnehmende Verträge und eine variable Anzahl nicht teilnehmender Verträge. Sie stellen die Vergleichbarkeit mit der Basisvariante her. Die etwas abgesetzte Messung ganz rechts in der Tabelle bearbeitet 100 teilnehmende Verträge und soll im Weiteren als Referenzzeit gelten.

Beim Vergleich der Messungen mit 10 teilnehmenden Verträgen wird anhand der processing time deutlich, dass die Messwerte nicht mehr maßgeblich durch die Anzahl der nicht teilnehmenden Verträge bestimmt sind. Im Vergleich zu der processing time der Basisvariante für 1200 nicht teilnehmende Verträge zeigt sich der Unterschied zum vorangegangenen Design am deutlichsten. Die Variante mit EJB QL ist um einen Faktor von etwa 86 000 schneller; 1 Sekunde im Vergleich zu 24 Stunden (86 400 Sekunden).

Die processing time liegt für 10 teilnehmende Verträge im Bereich von 60 bis 70 Sekunden. Das Design ist wie erwartet verbessert. Alle folgenden Varianten werden daher auf der EJB QL-Variante aufbauen.

Nachdem sich die processing time deutlich verbessert hat, kann die Zahl der teilnehmenden Verträge gesteigert werden. Bei 100 teilnehmenden Verträgen dauert der Batchprozess etwa 30 Minuten.

Beim Entwurf der EJB QL-Variante wurde angedeutet, dass es lohnenswert ist, die tatsächlich an die Datenbank gerichteten Anfragen zu analysieren. Im Fall des implementierten Beispiels sind das SQL-Anfragen an eine relationale Datenbank. Um zu verdeutlichen, welcher Zugewinn an Performanz damit erreicht werden kann, wird die Messung für 100 teilnehmende Verträge im Folgenden detaillierter untersucht.

Abbildung 8.12 stellt die processing time als Balkendiagramm dar. Durch die Blöcke werden die einzelnen Methoden repräsentiert. Wie aus der nebenstehenden Legende ersichtlich wird, ist der oberste Block »berechnePraemie« kaum erkennbar. Darunter folgt »ermittleBeitrag« und »ermittleMaxMindestBeitrag« usw. Die Messpunkte »historisiere RisikoBuendel« und alle darunter liegenden sind mit zusammen ca. 5 % gleichsam bedeutungslos. Den größten Anteil machen die Messpunkte »findeElementarRisiko« und »findeEinzelZuAbschlaege« mit je ca. 45 % aus.

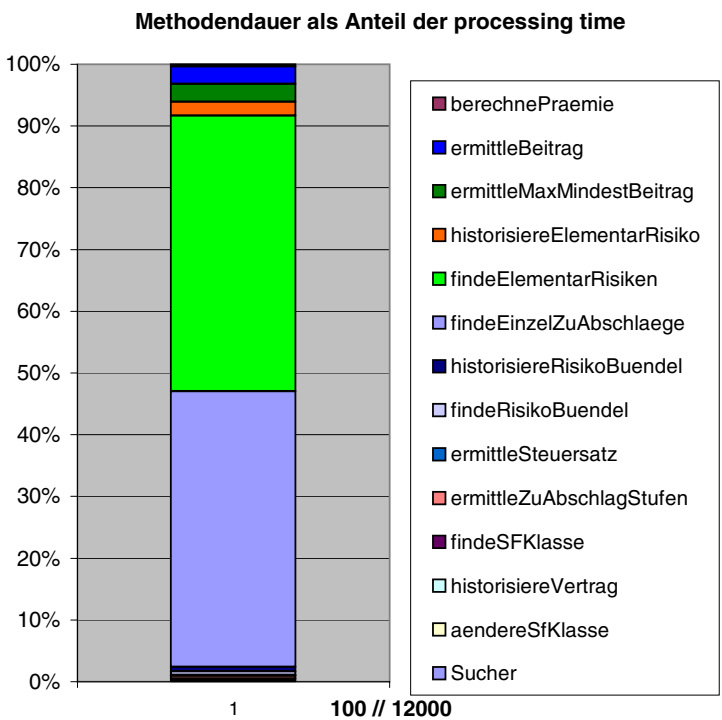


Abbildung 8.12: Anteile der gemessenen Einzelzeiten an processing time

Die beiden genannten Methoden stellen durch eine EJB QL-Anfrage eine Menge von Instanzen zusammen. Die in Abbildung 8.13 dargestellte Tabelle verzeichnet die Dauer eines *einzelnen* Aufrufs. Die Aufrufhäufigkeiten sind in Abschnitt 7.2 dokumentiert. Der gerahmte Wert von »findeElementarRisiken« ergibt sich beispielsweise, indem die Zeit 12:10.3 [min:sec.ds] aus der Tabelle in Abbildung 8.11 durch 300 geteilt wird. Da 100 Verträge am Batchlauf teilnehmen und jeder Vertrag drei `RisikoBuendel` besitzt, von denen die Funktionalität aufgerufen wird, dauert der einzelne Aufruf von »findeElementarRisiko« ca. 2,4 Sekunden (siehe Hervorhebung durch den Rahmen). Er ist daher lohnenswert für die weitere Betrachtung.

Methoden	100 // 12000
Sucher	00:00:00.4
aendereSfKlasse	00:00:00.0
historisiereVertrag	00:00:00.1
findeSFKlasse	00:00:00.0
ermittleZuAbschlagStufen	00:00:00.0
ermittleSteuersatz	00:00:00.0
findeRisikoBuendel	00:00:00.1
historisiereRisikoBuendel	00:00:00.0
findeEinzelZuAbschlaege	00:00:00.7
findeElementarRisiken	00:00:02.4
historisiereElementarRisiko	00:00:00.0
ermittleMaxMindestBeitrag	00:00:00.1
ermittleBeitrag	00:00:00.1
berechnePraemie	00:00:00.0
processing time	**00:00:16.5**

Abbildung 8.13: Dauer einzelner Aufrufe in Abhängigkeit der Anzahl nicht *teilnehmender Verträge*

Die EJB QL-Anfrage zu diesem Aufruf ermittelt zu einem Datum und dem Attribut `RisikoBuendel.fachId` alle gültigen `ElementarRisiko`-Instanzen:

```
<query>
<query-method>
  <method-name>findeAktuelleZuBuendel</method-name>
  <method-params>
    <method-param>int</method-param>
    <method-param>java.sql.Date</method-param>
  </method-params>
</query-method>
<ejb-ql>
  <![CDATA[SELECT OBJECT(v)
                   FROM  ElementarRisikoBean v
                   WHERE   v.risikoBuendelFachId = ?1
                       AND v.hGueltigBis = ?2
  ]]>
```

```
</ejb-ql>
</query>
```

Listing 8.3: EJB QL-Anfrage zur Ermittlung aktueller ElementarRisikoBeans

Durch Mitprotokollieren der Datenbankanfragen des Applikationsservers stellt sich heraus, dass über 90 % der Aufrufzeit von »findeElementarRisiko« durch die Abarbeitung der SQL-Anfrage in der Datenbank verbraucht werden. Die zugehörige SQL-Anfrage ist die folgende:

```
SELECT WL0.ero_id
    FROM BATCHUSER.ElementarRisiko WL0
    WHERE (     (WL0.ero_rbl_fachid  = :1)
            AND (WL0.ero_hGueltigBis = :2)
    );
```

Listing 8.4: Generierte SQL-Anfrage zu EQL-Anfrage

Wie nicht anders zu erwarten war, werden die Datenbankspalten `ero_rbl_fachid` und `ero_hGueltigBis` angefragt. Es liegt also nahe, diese Anfrage per Index zu beschleunigen (vgl. dazu [66] und [49]).

Die Erstellung eines Index auf einer bestehenden Tabelle beansprucht nur wenige Minuten. Die so modifizierte EJB QL-Variante führt zu der processing time, die in der folgenden Tabelle (Abbildung 8.14) dargestellt ist.

Methoden	100 // 12000	
	ohne Index	mit Index
Sucher	00:00:00.4	00:00:00.7
aendereSfKlasse	00:00:01.2	00:00:01.5
historisiereVertrag	00:00:05.5	00:00:08.3
findeSFKlasse	00:00:01.4	00:00:02.0
ermittleZuAbschlagStufen	00:00:07.4	00:00:09.6
ermittleSteuersatz	00:00:01.5	00:00:01.3
findeRisikoBuendel	00:00:09.3	00:00:10.8
historisiereRisikoBuendel	00:00:12.0	00:00:20.1
findeEinzelZuAbschlaege	00:12:10.4	00:11:51.6
findeElementarRisiken	00:12:10.3	00:00:04.2
historisiereElementarRisiko	00:00:36.8	00:01:03.2
ermittleMaxMindestBeitrag	00:00:46.9	00:00:51.0
ermittleBeitrag	00:00:46.9	00:00:47.4
berechnePraemie	00:00:05.6	00:02:00.0
processing time	**00:27:26.3**	**00:15:33.2**

Abbildung 8.14: Verbesserte Suchanfragen durch Index

Deutlich ist an der umrahmten Zeile zu sehen, wie sich die Dauer der indexierten Suchanfrage verbessert hat. Die dargestellte processing time hat ohne Index fast doppelt so viel Zeit benötigt (vgl. Abbildung 8.11). In der unmodifizierten EJB QL-Variante beanspruchte diese Anfrage ca. 12 Minuten der processing time. In der modifizierten EJB QL-Variante nur noch ca. 4 Sekunden. Insgesamt hat sich die processing time um diese 12 Minuten vermindert.

8.2.4 Bewertung

Die Messungen zeigen deutlich, dass die processing time sich verbessert, wenn nur Instanzen in den Applikationsserver geladen werden, die tatsächlich gebraucht werden.

Die EJB QL-Anfragen werden vom Applikationsserver in die Anfragesprache der Datenbank übersetzt. Durch Analyse der Anfragen an die Datenbank kann weitere Performanz gewonnen werden, wie durch die modifizierte EJB QL-Variante bestätigt wird.

8.2.5 Design-Regel

Suchanfragen müssen stets auf der Datenbank die exakte Treffermenge zusammenstellen. Das Laden von nicht benötigten Bean-Instanzen oder mehrfache Laden bereits vorhandener Instanzen kostet erhebliche Performanz. Insbesondere ist beim *Traversieren mehrerer Relationen* zu vermeiden, dass alle Zwischenstationen geladen werden, wenn Sie nicht unbedingt notwendig sind (siehe vorangegangenes Beispiel mit `Vertrag`, `RisikoBuendel` und `ElementarRisiko`).

Für EJB 2.0-Applikationen ist eine intensive Nutzung von EJB QL-Anfragen unumgänglich. Bei *EJB 1.0 und EJB 1.1* muss zwangsläufig per eingebettetem SQL o. Ä. die exakte Treffermenge bestimmt werden.

Die Wartbarkeit und Portierbarkeit nimmt durch eingebettete SQL-Befehle bzw. Bean-Managed Persistence stark ab. Die Persistenzschicht ist nicht mehr unabhängig, da z.B. Tabellen- und Spaltennamen in die Bean-Klassen hineinkodiert werden. Ändert sich das Namensschema von der Testumgebung zur Produktionsumgebung einer EJB-Applikation oder erzwingt ein Kunde ein spezielles Namensschema, ist damit ein hoher Änderungsaufwand notwendig. Alle Beans müssen nach der Namensänderung neu übersetzt und getestet werden. Bei der Verwendung von CMP und EJB QL testet dagegen der Applikationsserver, ob das Namensschema der Datenbank zum Deployment Descriptor passt.

8.3 Verbesserung durch Tabellteilung

Eine Tabelle einer Datenbank lässt sich unter Beibehaltung der logischen Struktur in mehrere verwaltbare Einheiten spalten. Der dafür gebräuchliche Begriff ist »Partitionierung«. Partitionierung wird bei besonders großen Datenmengen eingesetzt und ist

für den Nutzer einer Tabelle transparent. Sie wird bei der datenbankinternen Optimierung der Anfragen berücksichtigt, sodass bei einer *dem Geschäftsprozess angemessenen* Spaltung der Tabelle eine Performanzverbesserung erreicht wird.

Im folgenden Beispiel wird eine sinnvolle Spaltung erörtert und umgesetzt. Die Anpassungen auf Ebene des Applikationsservers und der Geschäftslogik werden begründet.

8.3.1 Literatur

Auf Ebene der Datenbank gibt es drei besondere Arten der Strukturierung von Sätzen, um das Volumen der zu einer Anfrage betrachteten Daten zu verringern:

- **Partitionen:** Durch Partitionen wird eine Tabelle in mehrere verwaltbare Einheiten zerlegt.
- **Segmentierung:** Die Segmentierung wird beim Design des Datenbankschemas eingesetzt. Dabei wird eine Tabelle in mehrere Tabellen zerlegt, die auch nur als einzelne Tabellen angesprochen werden können.
- **Views:** Durch einen View kann eine Teilmenge einer anderen Tabelle oder auch verknüpfte Information aus mehreren Tabellen repräsentiert werden. Views verhalten sich mit Einschränkungen wie Tabellen. Sie enthalten jedoch selbst keine Daten, sondern verweisen auf die Datensätze der Tabelle(n), auf der/denen der View basiert.

Diese drei Strukturierungsmöglichkeiten werden im Folgenden voneinander abgegrenzt.

[50] klassifiziert verschiedene Konzepte zur Spaltung einer Tabelle in Teiltabellen. Zu unterscheiden ist horizontale Partitionierung und vertikale Partitionierung. Bei horizontaler Partitionierung werden vollständige Zeilen (Tupel) einer Tabelle auf verschiedene Wartungseinheiten verteilt. Bei vertikaler Partitionierung werden die Tupel selbst zerlegt und in verschiedenen Wartungseinheiten gespeichert. In der Praxis besonders relevant ist wertbasierte, horizontale Partitionierung. Nach diesem Ansatz verwaltet jede Partition abhängig vom »Partitionierungsschlüssel« eine Teilmenge aller gespeicherten Instanzen. Dadurch wird die Zuordnung eines Tupels zu einer Partition im Nachhinein rekonstruierbar und kann bei der Anfrageoptimierung benutzt werden.

Bezogen auf das Beispiel des Buchs könnte als Partitionierungsschlüssel die Versicherungsscheinnummer gewählt werden. Jede Partition enthält dann einen Nummernkreis von Versicherungsscheinnummern, die zu gewissen Geschäftsstellen gehören. Anfragen einer Geschäftsstelle greifen nach der internen Optimierung durch die Datenbank nur noch auf eine Partition zu und müssen nicht stets den ganzen Bestand durchsuchen.

»Vertikale Partitionierung« bildet Teilmengen bezüglich der Attribute aller Instanzen. Für das Beispiel des Buchs interessieren am Vertrag Attribute wie `inkassoUnterDrückungGrund` oder `beteiligungsVerhältnis` nicht. Die Spaltung sieht daher eine Partition vor, die für alle Instanzen die geschäftsvorfallrelevanten Attribute enthält, und eine Partition für alle weiteren Attribute. Der Primärschlüssel muss in jeder Partition vorhanden sein, sodass über ihn partitionierte Instanzen wieder zusammengefügt werden können. Diese

Art der Partitionierung wird für die Auslagerung viel Speicherplatz benötigender und selten gebrauchter Attribute empfohlen.

Von dem Begriff der Partitionierung ist der Begriff der Segmentierung abzugrenzen. Er wird in [22] benutzt. Bei der Segmentierung wird in gleicher Weise wie bei der Partitionierung zwischen horizontaler und vertikaler Spaltung unterschieden. Während Partitionierung die inhaltliche Einheit der Tabelle bewahrt und sie lediglich in verschiedene Verwaltungseinheiten unterteilt, wird bei der Segmentierung eine einzelne Tabelle tatsächlich in mehrere *Tabellen* aufgeteilt im Sinne eines modifizierten Datenbankschemas.

Bei Segmentierung ist auf *Ebene einer nutzenden Anwendung* Sorge dafür zu tragen, dass mehrere Tabellen durchlaufen werden, um alle Elemente der zuvor unsegmentierten Tabelle aufzufinden. Bei einer Partitionierung trägt die Datenbank Sorge, alle Verwaltungseinheiten anzusprechen; aus Sicht der Anwendung wird bei Partitionierung stets eine einzelne Tabelle angesprochen.

Mit einem View kann ein Ausschnitt der Information einer Tabelle festgehalten werden. Dieser Ausschnitt wird durch eine Suchanfrage zusammengestellt und ab diesem Zeitpunkt aktuell gehalten. Ändern sich die Daten, auf denen der View basiert, wird der View ebenfalls angepasst. Die Konsistenz zwischen View und Tabelle ist Aufgabe der Datenbank. Auf Ebene der Anwendung ist eine weitere Bean zu erzeugen, die auf den View anstelle einer Tabelle abbildet. Es ist damit denkbar, die Anzahl teilnehmender Geschäftsobjekte vor dem Start des Batchlaufs in einem View zusammenzustellen. Auf diesen View bildet dann eine eigene Bean ab.

Im verbleibenen Abschnitt wird nach Umsetzungen der beschriebenen Konzepte auf dem verwendeten RDBMS gesucht. Segmentierung betrifft den Entwurf des Datenbankschemas, sodass keine besondere technische Unterstützung nötig ist. Views gehören zur Grundfunktionalität jedes RDBMS und werden per SQL-Befehl angelegt.

Das Konzept horizontaler Partitionierung wird in der Wortprägung der Oracle-Datenbank schlicht mit »Partitionierung« bezeichnet. Die Beschreibung des Entwurfs folgt diesem Wortgebrauch. Die (horizontale) Partitionierung wird in [66, S. 237ff] besonders empfohlen für Tabellen, die eines der drei folgenden Kriterien erfüllen:

- Die enthaltenen Daten haben einen zeitlichen Bezug.
- Die Anzahl der Datensätze ist hoch.
- Es existiert ein *fachliches* Unterteilungskriterium.

Partitionierung beschleunigt Anfragen, indem nur noch relevante Partitionen berücksichtigt werden: »Partitions that are not required by a query can be transparently eliminated by the cost-based optimizer (CBO)« [66, S. 247]. Aus Sicht des Applikationsservers als Client der Datenbank ist die Partitionierung der Tabellen transparent. Sie muss nicht bei der Implementierung der Beans berücksichtigt werden. Partitionen werden per SQL-Befehl angelegt. Technische Details zu den beschriebenen Umsetzungsmöglichkeiten sind in [38], [49] und [66] zu finden.

8.3.2 Übertragung und Entwurf

Um *Segmentierung* innerhalb der Beispielapplikation zu betreiben, muss die Trennung der Daten auch auf Ebene der Beans nachvollzogen werden.

Soll beispielsweise die Vertragstabelle nach Versicherungsscheinnummern horizontal segmentiert werden, so sind ebensoviele unterschiedliche Bean-Typen anzulegen. Funktional sind diese Typen gleich. Sie unterscheiden sich durch ihr Persistenzschema, da jede Bean in eine andere durch Segmentierung entstandene Tabelle abbildet. Zusätzliche Logik entscheidet abhängig von der Versicherungsscheinnummer, auf welche Tabelle und welche Bean gerade zugegriffen wird. Werden alle Instanzen benötigt, so hat die *Anwendung* dafür Sorge zu tragen, dass auch alle Tabellen angefragt werden.

Vertikale Segmentierung verursacht ebenso großen Aufwand. Pro Segment muss eine zusätzliche Bean erstellt werden, wobei zusätzliche Logik die Integrität der Daten sicherstellen muss. Eine solche Lösung wird bei mehr als zwei Segmenten schwierig zu warten, sodass davon abzuraten ist.

Schreibende Zugriffe sind auf Views nur mit technischen Einschränkungen möglich. Durch die Historisierung in der Anwendung resultiert eine Änderung eines Attributs stets in das Schreiben einer neuen Instanz als neuem Historienstand. Ausschließlich lesende Zugriffe kommen damit allein beim Ermitteln der Teilnehmer des Batchlaufs vor. Für den Enwurf müssen damit die technischen Einschränkungen bei schreibenden Zugriffen bedacht werden.

Damit das Einfügen einer Instanz in einen View möglich ist, müssen alle nicht-null Attribute der dem View zugrunde liegenden Tabellen im View enthalten sein [38, S. 437]. In dem Entwurf sind jedoch kaum nicht-null Attribute vorhanden. Die vorhandenen nicht-null Attribute sind vom Datenvolumen so gering, dass sich der Aufwand, einen View zu nutzen, nicht lohnt. Speichert der Vertrag etwa Fotos des versicherten Objekts, könnten diese bei einem View ausgeschlossen werden, um so Ladezeit und Bandbreite zu sparen. Für das vorliegende Beispiel ist es daher nicht ratsam, einen View umzusetzen.

Für die sinnvolle Nutzung einer (horizontalen) Partitionierung ist zu überlegen, an welcher Teilmenge von Instanzen dieser Geschäftsprozess interessiert ist. Die Teilnehmer am Geschäftsprozess sind aktuelle Verträge. Weiterhin ist die Teilnahme am Geschäftsprozess abhängig von der Hauptfälligkeit eines Vertrags. Die Hauptfälligkeit wird einmal pro Jahr erreicht, da als Versicherungsperiode für die Beitragsberechung in der Regel genau ein Jahr gewählt wird [25, S. 549].

Eine grobe Partitionierung könnte drei Partitionen vorsehen. Eine für nicht aktuell gültige Historienstände, eine für teilnehmende Verträge und eine für bereits bearbeitete Verträge. Dieses Schema lässt sich weiter verbessern. Da der Batchlauf monatlich stattfindet, findet er stets alle Verträge, die innerhalb des zurückliegenden Monats hauptfällig geworden sind. Die zuvor genannte Partition der teilnehmenden Verträge wird also nach ihrem Fälligskeitmonat verfeinert.

Zur Partitionierung im Oracle-RDBMS gehört ein Partitionierungsschlüssel. Er ist eine Spalte der zu partitionierenden Tabelle und damit ein Attribut der zugehörigen Objekte.

Abhängig von den Werten dieses Attributs werden die Instanzen einer Partition zugeordnet. Wenn das Datum der nächsten Hauptfälligkeit als persistentes Attribut in den Vertrag aufgenommen wird, obwohl es aus dem `beginnDatum` zu errechnen ist, so ist es nach allen Vorüberlegungen ein idealer Partitionierungsschlüssel. Hätten alle Verträge eine Vertragsdauer von einem Jahr, so wäre die Versicherungsperiode gleich der Vertragsdauer und es könnte das Ablaufdatum als Partitionierungsschlüssel genutzt werden. Auf diese fachliche Einschränkung soll aber nicht eingegangen werden.

Zur Partitionierung der Vertragstabelle wird das »create table«-Statement um die Partitionierungsinformation erweitert:

```
-- ======================================================
--   Table: VERTRAG
-- ======================================================
create table VERTRAG
(
    VTG_ID                  NUMBER(10) not null,
    -- weitere Spalten ...
    VTG_HAUPTFAELLIGKEIT    DATE       not null,
    -- weitere Spalten ...
    constraint PK_VERTRAG primary key (VTG_ID)
)
-- Partitionierung
partition by range (VTG_HAUPTFAELLIGKEIT)
(
    partition VERALTET
        values less than to_date (
            01-JAN-2001, DD-MON-YYYY
        ),
    partition JANUAR
        values less than to_date (
            01-FEB-2001, DD-MON-YYYY
         ),
    -- ... weitere Partitionen
    partition DEZEMBER
        values less than to_date (
            1-JAN-2002, DD-MON-YYYY
        ),
    partition BEARBEITET
        values less than (MAXVALUE)
)
```

Listing 8.5: Create table-Anweisung mit Partitionierung

Der Vertrag hat das zusätzliche Attribut `VTG_HAUPTFAELLIGKEIT` erhalten. Es muss ebenso in die `VertragBean` aufgenommen werden. Ist der Vertrag zur Hauptfälligkeit abgerech-

net, so muss das Attribut vom Abrechnungsprozess auf die nächste Hauptfälligkeit aktualisiert werden.

Die Vertragsinstanzen werden nun in 14 Partitionen eingeordnet, abhängig von Ihrer Hauptfälligkeit. Die Partition `VERALTET` enthält alle Verträge, deren Hauptfälligkeit hinter dem Januar des aktuellen Jahres liegen. Wenn sie im aktuellen Jahr nicht hauptfällig sind, so kann es sich nur um veraltete Historienstände oder stornierte, d. h. nicht mehr gültige Verträge handeln. Zwölf weitere Partitionen enthalten aktuelle Verträge, die im entsprechenden Monat ihre Hauptfälligkeit erreichen. Ist die Abrechnung zur Hauptfälligkeit durchgeführt, so ist die nächste Hauptfälligkeit im nächsten Jahr, die bearbeitete Instanz landet damit in der Partition `BEARBEITET`.

Bei dieser Anwendung von Partitionierung entsteht zusätzlicher Aufwand für das Aktualisieren der Partitionen. Dieses Vorgehen wird auch als »bewegliches Zeitfenster« beschrieben. Wird nicht aktualisiert, werden alle aktuellen Instanzen in der letzten Partition gespeichert und die Performanz sinkt langsam. Da alle Partitionen nach außen wie *eine* Tabelle behandelt werden, ist jedoch durch ein vergessenes Aktualisieren keine Fehlfunktion der Geschäftslogik zu erwarten. Damit sprechen keine schwer wiegenden Einwände gegen eine Nutzung von Partitionen.

Um der Datenbank die Möglichkeit zu geben, eine Anfrage durch partition pruning zu beschleunigen, d. h. durch Beschränken einer Anfrage auf gewisse Partitionen, muss der Partitionsschlüssel auch in der entsprechenden Anfrage vorkommen. Damit ist in der Beispielanwendung die Implementierung des `Suchers` so zu verändern, dass die Hauptfälligkeit berücksichtigt wird.

Bisher ist nur der Vertrag als Einstiegspunkt in die Struktur des Geschäftsobjekts bearbeitet worden. Alle anderen Bausteine wie `RisikoBuendel` oder `EinzelZuabschlag` haben kein Hauptfälligkeitsattribut. Ein solches Attribut hinzuzufügen und abhängig vom Vertrag zu verwalten, ist zu kompliziert. Anstelle dessen wird eine grobere Partitionierung abhängig vom Historisierungsattribut `hGueltigBis` eingesetzt.

Vertragsbausteine, die vor der *letzten Hauptfälligkeit* ihre Gültigkeit verloren haben, sind auch bis zur letzten Hauptfälligkeit abgerechnet worden; sie sind für das Beispiel ohne Bedeutung. Liegt `hGueltigBis` mehr als ein Jahr zurück, ist der Baustein veraltet und dient allein der Rekonstruktion von Vertragsständen. Die Partitionen eines `ElementarRisiko` sehen also wie folgt aus:

```
-- ================================================
--   Table: ELEMENTARRISIKO
-- ================================================
create table ELEMENTARRISIKO
(
    ERO_ID              NUMBER(10)  not null,
    -- ...
    ERO_HGUELTIGBIS     DATE        not null,
    -- ...
    constraint PK_ELEMENTARRISIKO primary key (ERO_ID)
```

```
)
-- Partitionierung
-- gegenwaertig sei Januar 2001!
partition by Range (ERO_HGUELTIGBIS)(

    partition veraltet
        values less than (to_date(
            '01-JAN-2000', 'DD-MON-YYYY'
        )),
    partition aktuell
            values less than (MAXVALUE)

)
```

Listing 8.6: Partitionierung historisierter Instanzen

Alle anderen historisierten Vertragsbausteine werden ebenso partitioniert.

Alle Partitionen sind regelmäßig zu warten, um gleich bleibende Leistung zu erhalten. Für diese Partitionierung sind keine Suchanfragen anzupassen, da ohnehin das Historisierungsattribut `hGueltigBis` bei jeder Anfrage mitbenutzt wird, um den passenden Historienstand zu ermitteln.

Ein Performanzgewinn soll erreicht werden, indem auf Ebene der Datenbank Instanzen nicht betrachtet werden, die allein der Rekonstruktion vergangener Historienstände dienen. Wichtig ist, dass sich die Abbildung der Bean auf die Tabelle nicht ändert; sie bildet weiterhin auf die eine, gleiche Tabelle wie vor der Partitionierung ab. Suchanfragen, die etwa zu einem Vertrag alle historischen Stände ermitteln, funktionieren damit nach wie vor.

8.3.3 Messergebnis

Ausgangspunkt dieser Variante ist die EJB QL-Variante aus Abschnitt 8.2. Die Tabelle in Abbildung 8.15 zeigt die ermittelten Werte für die EJB QL-Variante und die partitionierte Variante.

An der processing time ist zu sehen, dass die partitionierte Variante deutlichen Performanzgewinn bringt. Die EJB QL-Variante benötigt ca. 27 Minuten, wohingegen die partitionierte Variante nur ca. 3 Minuten benötigt.

Abbildung 8.16 zeigt die Tabelle als Balkendiagramm. Balken einer Farbe gehören zu einer gemessenen Variante. Jeder Balken entspricht einem Eintrag der Tabelle. Es ist deutlich zu sehen, dass die Zeitersparnis der partitionierten Variante gegenüber der EJB QL-Variante aus den beschleunigten Suchfunktionen »findeEinzelZuAbschläge«, »findeElementarRisiken« und »findeRisikoBuendel« herrührt.

Methoden	EJB QL 100 // 12000	Partition 100 // 12000
Sucher	00:00:00.4	00:00:00.4
aendereSfKlasse	00:00:01.2	00:00:00.2
historisiereVertrag	00:00:05.5	00:00:05.9
findeSFKlasse	00:00:01.4	00:00:02.0
ermittleZuAbschlagStufen	00:00:07.4	00:00:07.3
ermittleSteuersatz	00:00:01.5	00:00:01.8
findeRisikoBuendel	00:00:09.3	00:00:02.4
historisiereRisikoBuendel	00:00:12.0	00:00:15.2
findeEinzelZuAbschlaege	00:12:10.4	00:00:08.6
findeElementarRisiken	00:12:10.3	00:00:07.7
historisiereElementarRisiko	00:00:36.8	00:00:45.6
ermittleMaxMindestBeitrag	00:00:46.9	00:00:47.2
ermittleBeitrag	00:00:46.9	00:00:46.8
berechnePraemie	00:00:05.6	00:00:05.5
processing time	**00:27:26.3**	**00:03:28.4**

Abbildung 8.15: Vergleich EJB QL-Variante und partitionierte Variante

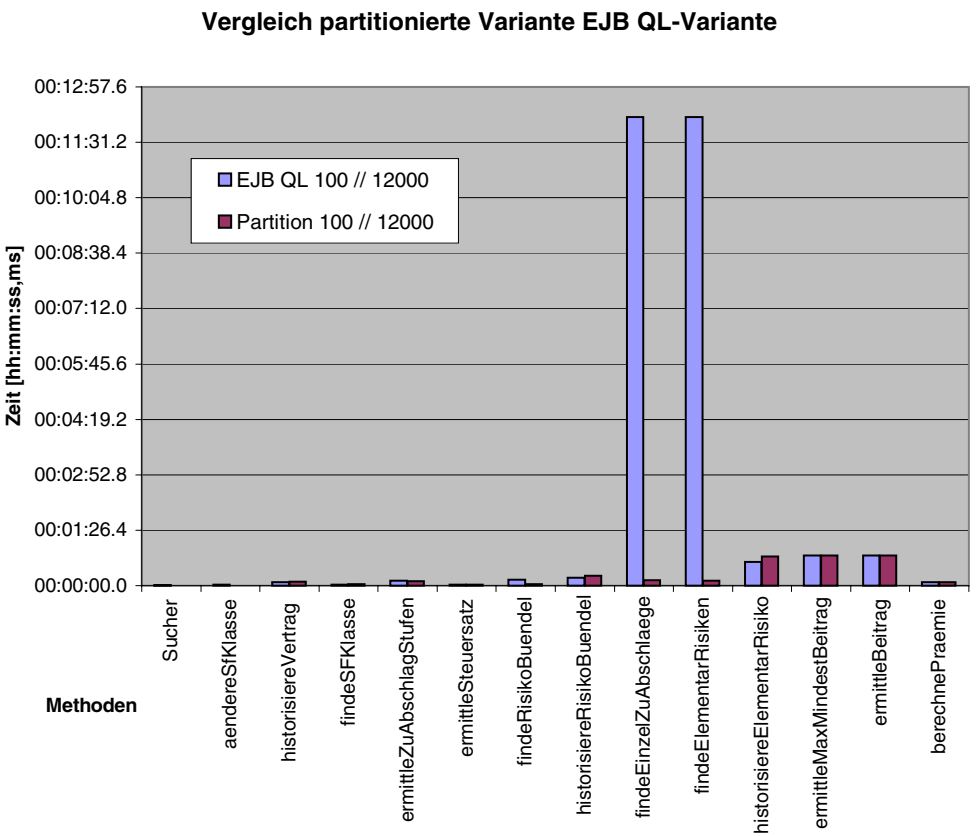


Abbildung 8.16: Balkendiagramm: EJB QL-Variante und partitionierte Variante

8.3.4 Bewertung

Der Vergleich der EJB QL-Variante mit der partitionierten Variante hat deutlich gezeigt, dass eine fachlich getriebene Partitionierung der Datenbanktabellen geeignet ist, gewisse Anfragen zu beschleunigen.

Der zusätzliche Entwicklungsaufwand einer solchen Lösung ist als gering zu erachten. Da diese Verbesserung sich wesentlich auf der Ebene der Persistenzschicht auswirkt, bleibt die Struktur der Beans unberührt. Lediglich die Suchanfragen, die durch die Partitionierung beschleunigt werden sollen, müssen zusätzlich den Partitionierungsschlüssel beinhalten.

Auf Ebene der Datenbank entsteht zusätzlicher Aufwand im Betrieb, da die Partitionen regelmäßig aktualisiert werden müssen. In diesen Zeiten kann die Datenbank nicht für Geschäftsprozesse genutzt werden.

Demgegenüber steht der Vorteil, dass partitionierte Tabellen besser gewartet werden können und besser skalieren [66, S. 247]. Oft werden Partitionen allein aufgrund des Wartungsvorteils eingesetzt.

8.3.5 Design-Regel

Datenbanktabellen werden oft allein aus Gründen der Skalierung und Wartbarkeit partitioniert. Diese Partitionen lassen sich leicht zur Steigerung der Performanz nutzen, wenn als Partitionierungsschlüssel ein fachlich bedeutsames Attribut der zu speichernden Instanzen gewählt wird. Bei zeitabhängigen Daten werden beispielsweise die aufbewahrungspflichtigen, »veralteten« Daten von den aktuellen, für die Geschäftsvorfälle relevanten Daten durch Partitionierung separiert.

Diese Verbesserung lässt sich für EJB 1.0- und EJB 1.1-Anwendungen ebenfalls einsetzen, da sie allein die Persistenzschicht betrifft.

8.4 Local Interfaces

Jeder Aufruf zwischen zwei EntityBean-Instanzen erzeugt Kosten für die RMI-Kommunikation (Remote Method Invocation), für die Containerlogik und für Datenbankzugriffe.

Besonders die RMI-Kommunikation ist kostspielig, da die Aufrufparameter serialisiert und deserialisiert werden müssen. Die RMI-Kommunikation ermöglicht es, EJB-Applikationen einfach über mehrere Rechner zu verteilen.

Gehören Beans jedoch fachlich sehr eng zusammen, sodass eine Bean den Lebenszyklus einer anderen Bean bestimmt, ist es nicht sinnvoll, diese Beans auf verschiedenen Rechnern zu betreiben. Beispielsweise ein `Vertrag` und seine `RisikoBuendel` stehen in einem solchen (Kompositions-) Verhältnis. Wird der `Vertrag` gelöscht, so werden auch die

`RisikoBuendel` gelöscht, da sie nur im Kontext des `Vertrags` Bedeutung haben. Da zwischen solchen Beans ein hohes Kommunikationsaufkommen herrscht, werden sie in der gleichen Serverinstanz betrieben. Es ist damit überflüssig, sie per RMI kommunizieren zu lassen.

In der folgenden Variante sollen diese überflüssigen Kosten reduziert werden. Das ist möglich durch die in EJB 2.0 eingeführten Local Interfaces oder durch proprietäre Optimierungsmöglichkeiten der Applikationsserver. Die proprietären Optimierungsmöglichkeiten bestehen u.a. darin, dass die Parameterübergabe »by value« durch Parameterübergabe »by reference« ersetzt wird. Damit wird aber auch erzwungen, dass die beteiligten Beans im gleichen Container eingesetzt werden.

8.4.1 Literatur

Zur Einsparung von Kommunikationskosten sollen in diesem Kapitel vier Möglichkeiten diskutiert werden.

1. **Dependent Objects:** Bevor im »Proposed Final Draft 2« der EJB 2.0-Spezifikation vom 23. April 2001 Local Interfaces eingeführt wurden, sollte das Problem der Kolokation von Beans und die Parameterübergabe per Referenz durch »Dependent Objects« gelöst werden. Sie sind als Designmöglichkeit mit EJB 2.0 (»Proposed Final Draft 1« vom 23. Oktober 2000) eingeführt worden und aus dem letzten Entwurf der Spezifikation wieder gestrichen worden. Wenn sie hier dennoch beschrieben werden, so hat das zwei Gründe. Erstens lassen sich auf ihrer Basis andere Verbesserungskonzepte einfacher verstehen. Zweitens wurde der zweite »Proposed Final Draft« eher in aller Stille veröffentlicht, sodass viele Entwickler eine Erklärung zu den im ersten Vorentwurf laut angekündigten Dependent Objects erwarten. Dependent Objects sollten nur innerhalb der EntityBean sichtbar und nicht serialisierbar sein. Ihre Persistenz wurde festgelegt durch den Deployment Descriptor und den PersistenceManager des Applikationsservers. Sie sollten sich laut [39] ähnlich wie CMP-EntityBeans verhalten. Performanzgewinn versprachen sie aus zwei Gründen.

   - Es entfallen Serialisierungskosten, da an und von »Dependent Objects« Parameter per Referenz übergeben werden.
   - Die Transaktionslogik wird von den übergeordneten Beans gesteuert, sodass »Dependent Objects« ein geringeres Maß an eigenen Transaktionskosten verursachen. Die Datenbankzugriffe werden damit günstiger.

   Veröffentlichungen im Internet besagen, dass diese Technik wegen Spezifikationslücken und Zweideutigkeiten umstritten ist. In [35] wird kritisiert, dass im Konzept der »Dependent Objects« zwei verschiedene Ziele vermischt und unsauber umgesetzt werden. Einerseits das Ziel Beans lokal, d.h. ohne Serialisierung der Parameter, ansprechen zu können; andererseits das Ziel, auf einfache Art Daten aus der Datenbank lesen zu können. Das hat dazu geführt, dass »Dependent Objects« wieder aus der Spezifikation gestrichen wurden.

   Stattdessen gibt es nun in der Spezifikation das Konzept der Local Interfaces und den Typ der ejbSelect-Methoden (vgl. Abschnitt 4).

2. **ejbSelect-Methoden:** Durch ejbSelect-Methoden kann eine Bean direkt Daten aus der Datenbank lesen, ohne dass dazu eine andere Bean zu erstellen ist. Das erspart die Kosten für die Aktivierung und Erzeugung der Bean-Instanz und die Kosten für die Transaktionslogik, die mit jeder Bean verbunden sind.

3. **Local Interfaces:** Durch Local Interfaces (siehe [18, S. 51ff]) wird erzwungen, dass Aufrufer einer Methode und aufgerufene Instanz in der gleichen JVM (Java Virtual Machine) liegen müssen. Dadurch können Parameter per Referenz ausgetauscht werden. Die Serialisierungskosten werden gespart.

4. **Proprietäre Mechanismen:** Die Nutzung proprietärer Mechanismen des konkret eingesetzten Servers, BEA WebLogic 6.1, ist eine weitere Möglichkeit, Kommunikationskosten zwischen Applikationsserver und Datenbank zu sparen. In der Onlinedokumentation zu den DeploymentProperties des WebLogic 6.1 lässt sich nachlesen, dass der Schalter `enable-call-by-reference` in der Datei WebLogic-ejb-jar.xml die Parameterübergabe per Referenz ermöglicht. Die Kosten für Serialisierung werden so gespart. Dieser Schalter addressiert also das gleiche Problem wie die Local Interfaces.

8.4.2 Übertragung und Entwurf

Die genannten Verbesserungsmöglichkeiten werden nun in der Reihenfolge ihrer Vorstellung auf die Anwendung übertragen.

Wie erwähnt, können die *Dependent Objects* nicht genutzt werden, da sie aus der letzten Version der Spezifikation gestrichen wurden. Die *ejbSelect-Methoden* sind nur nutzbar, wenn alle beteiligten Instanzen per Container-Managed Relation (CMR) verbunden sind. Aufgrund der gewählten Historisierung gibt es in der Beispielanwendung nur zwischen `Vertrag` und `SfKlasse` eine solche Relation. Selbst wenn eine Beschleunigung dieser Relation die davon abhängige Zeit »findeSfKlasse« halbiert, ist für die Beispielanwendung der Gewinn so gering, dass sich der Aufwand nicht lohnt.

Für die Nutzung der *Local Interfaces* ist zuerst zu entscheiden, wie die Beans räumlich verteilt werden können. Ausgehend vom Design-Modell ist zu entscheiden, welche Klassen eine fachliche Einheit bilden. Technisch gesehen wird ein Komponentenschnitt ausgeführt. Als Faustregeln für das Finden zusammengehöriger Klassen gelten [51, S. 185f]:

- **Fachlicher Zusammenhang:** Fachlich eng zusammengehörige Klassen gehören in eine Komponente. Kandidaten für Komponenten sind Geschäftsobjekte
- **Vererbung:** Sub- und Superklasse befinden sich in der gleichen Komponente.
- **Abhängigkeit des Lebenszyklus:** Aggregat und Aggregierende werden der gleichen Komponente zugeordnet.
- **Komplexität der Schnittstellen:** Zwischen verschiedenen Komponenten sollten möglichst einfache Schnittstellen entstehen.
- **OO-Paradigma:** Ziel des Schnitts ist es, »hohe Kohäsion und geringe Kopplung« zu erreichen. Das heißt die Abhängigkeiten und das Kommunikationsaufkommen

zwischen den Komponenten sollten gering sein. Innerhalb einer Komponente dürfen die Abhängigkeiten groß sein.

Die Klassen und Methoden, die dem Inneren einer Komponente zugerechnet werden, werden per Local Interface angesprochen. Kommunikation zwischen verschiedenen Komponenten wird weiterhin über Remote Interfaces abgewickelt.

Der Ausschnitt einer Bestandsführung, der in diesem Buch umgesetzt wird, sollte als Vorgabe bereits eine fachliche Einheit im Sinne eines Geschäftsobjekts bzw. einer fachlichen Komponente bilden (vgl. Abschnitt 6.1). Ein Komponentenschnitt wird an dieser Stelle daher nicht wiederholt.

Als nach außen sichtbare Bean bietet sich die Klasse `Vertrag` als Wurzel der baumartigen Vertragsstruktur an. Alle anderen Klassen können innerhalb der Komponente verborgen bleiben. Wird beispielsweise für Anzeigezwecke eine Auflistung aller `ElementarRisiko`-Instanzen angefordert, so werden nicht RemoteInterfaces dieser Klasse nach außen veröffentlicht. Anstelle dessen wird eine Menge von Value Objects (vgl. [46, S. 257ff]) übergeben. Ein »Value Object« ist eine einfache, serialisierbare Java-Klasse, die alle Client-relevanten Attribute der zugehörigen Bean enthält. Sie fungieren dem Client gegenüber als Stellvertreter der Bean.

Um zu einer bestehenden Bean ein Local Interface anzubieten, wird zusätzlich ein Interface erstellt, das von `javax.ejb.EJBLocalObject` erbt. Es deklariert alle Methoden, die innerhalb der Komponente sichtbar sein sollen.

```
import ...
public interface ElementarRisikoLocal
    extends EJBLocalObject
{
    //... weiter Methoden
    double ermittleBeitrag(
        Collection vertraglicheZuAbschlagStufen,
        Collection buendelZuAbschlagStufen,
        Collection buendelEinzelZuAbschlaege,
        double vertraglicherSfr,
        double buendelBeitragsfaktor
    );
}
```

Listing 8.7: Ein Local Interface

Die Implementierung der Methoden befindet sich, wie auch bei RemoteInterfaces üblich, in der Bean selbst. Ebenso ist ein LocalHomeInterface zu erstellen, das von `javax.ejb.EJBLocalHome` erbt. Im Deployment Descriptor **ejb-jar.xml** muss der Name des LocalHomeInterface und des Local(Buisness)Interface deklariert werden.

```
<entity>
 <ejb-name>ElementarRisikoBean</ejb-name>
 <home>
   de.adesso.finaleVariante.vertrag.ElementarRisikoHome
 </home>
 <remote>
   de.adesso.finaleVariante.vertrag.ElementarRisiko
 </remote>

 <local-home>
   de.adesso.finaleVariante.vertrag.ElementarRisikoLocalHome
 </local-home>
 <local>
   de.adesso.finaleVariante.vertrag.ElementarRisikoLocal
 </local>

 <ejb-class>
   de.adesso.finaleVariante.vertrag.ElementarRisikoBean
 </ejb-class>
 <persistence-type>Container</persistence-type>
 <prim-key-class>java.lang.Integer</prim-key-class>
...
 </entity>
```

Listing 8.8: Aufnehmen des Local Interface in Standard Deployment Descriptor

Im WebLogic spezifischen Deployment Descriptor **weblogic-ejb-jar.xml** ist der Name des LocalHomeInterface zu deklarieren:

```
<weblogic-enterprise-bean>
 <ejb-name>ElementarRisikoBean</ejb-name>
 <entity-descriptor>
     ...
 </entity-descriptor>
 <jndi-name>
   de.adesso.finaleVariante.vertrag.ElementarRisikoHome
 </jndi-name>
 <local-jndi-name>
   de.adesso.finaleVariante.vertrag.ElementarRisikoLocalHome
 </local-jndi-name>
</weblogic-enterprise-bean>
```

Listing 8.9: Aufnehmen des Local Interface in spezifischen Deployment Descriptor

Diese Schritte müssen für alle Beans wiederholt werden, die per Local Interface ansprechbar sein sollen. Natürlich ist es möglich, dass manche Beans nur über ein Remote-

oder nur über ein Local Interface ansprechbar sind. Es müssen nicht zwangsläufig beide Interface-Typen für jede Bean zur Verfügung gestellt werden.

Im Quelltext selbst erhält man ein Local Interface als Rückgabewert eines `create` oder `find` auf einem LocalHomeInterface. Der Naming-Service liefert per Lookup das Local Home Interface, so wie der Lookup auch vom (Remote) Home Interface bekannt ist. Was in der Beschreibung verworren klingt, lässt sich einfach im Quellcode überblicken:

```
//1.
ElementarRisikoLocalHome   sstHome
  = (ElementarRisikoLocalHome) PortableRemoteObject.narrow(
    ctx.lookup(
      "de.adesso.vertrag.ElementarRisikoLocalHome"
    ),
    Class.forName(
      "de.adesso.vertrag.ElementarRisikoLocalHome"
    )
);

//2.
Iterator alleElementarRisiken
  = ( sstHome.findeAktuelleZuBuendel( getId().intValue(),
                                      DBHelper.MAX_DATE )
      ).iterator;
//3.
while(...){
    ElementarRisikoLocal elementarRisiko
      = (ElementarRisikoLocal) alleElementarRisiken.next();

    elementarRisiko.ermittleBeitrag(...);
}
```

Listing 8.10: Nutzung von Local Interfaces

Dieses Quelltextfragment stammt aus `RisikoBuendelBean.java`. Es zeigt den zuvor beschriebenen Ablauf:

1. Der erste Befehl ermittelt über den Namen des LocalHomeInterface und den Naming-Service eine Referenz auf das LocalHomeInterface.
2. Im zweiten Befehl wird auf dem LocalHomeInterface eine Find-Methode aufgerufen. Die Find-Methode liefert ein Menge von Local Interfaces als Rückgabewert.
3. Der dritte Teil des Quelltextfragments zeigt, wie eine `while`-Schleife über die Ergebnismenge der Suchanfrage iteriert. Die einzelnen Elemente der Treffermenge werden per `cast`-Operator in Local Interfaces für die `ElementarRisikoBean` verwandelt. Auf diesen Interfaces wird schließlich die oben deklarierte Methode `ermittleBeitrag` aufgerufen.

Wie an diesem Beispiel zu sehen ist, sind keine Neuerungen in der Programmierung mit Local Interfaces hinzugekommen. Alle Schritte sind bereits in ähnlicher Form von der Programmierung mit Remote Interfaces bekannt. Wesentlich bei der Benutzung von Local Interfaces ist die Designentscheidung, welche Beans lokal miteinander kommunizieren sollen.

Nachdem die Local Interfaces ausführlich vorgestellt worden sind, muss noch die *Nutzung proprietärer Mechanismen* als vierte Möglichkeit zur Vermeidung von Kommunikationskosten vorgestellt werden.

Um die Parameterübergabe per Referenz zu aktivieren, wird in **weblogic-ejb-jar.xml** das Attribut für `enable-call-by-reference` pro Bean auf true gesetzt:

```
<weblogic-enterprise-bean>
  <ejb-name>VertragBean</ejb-name>
  <!-- weitere Beschreibungselemente... -->
  <enable-call-by-reference>true</enable-call-by-reference>
</weblogic-enterprise-bean>
```

Listing 8.11: Deklaration der Parameterübergabe per Referenz

Der dargestellte Ausschnitt zeigt, wie für `VertragBean` die Parameterübergabe auf Übergabe per Referenz umgestellt wird. Die komplette Datei ist auf der beiliegenden CD-ROM zu finden.

Das von WebLogic genutzt Protokoll »T3« des Naming-Service wechselt automatisch den Übergabemodus der Parameter, falls die angesprochene Bean nicht im gleichen Server eingesetzt wird. Daher kann durch Setzen dieses Attributs auf `true` kein Fehler gemacht werden. Damit ist die Übergabe der Parameter per Referenz in wenigen Augenblicken aktiviert.

8.4.3 Messergebnis

Die folgend beschriebenen Messungen ermitteln den Performanzgewinn von Local Interfaces bzw. dem WebLogic-spezifischen T3-Protokoll. Hierzu werden drei verschiedene Varianten miteinander verglichen.

1. **Modifizierte EJB QL-Variante:** Diese Variante ist Basis der anderen beiden Varianten. Sie entspricht der in Unterkapitel 8.2 beschriebenen Umsetzung. Die Suchanfragen an `ElementarRisiko` und `RisikoBuendel` sind durch Indizes beschleunigt. In dieser Variante steht der Schalter `<enable-call-by-reference>` für alle Beans auf `FALSE`.

2. **T3 und modifizierte EJB QL-Variante:** Diese Variante entspricht der zuerst genannten Variante. Allein der Schalter `<enable-call-by-reference>` hat für alle Beans den Wert `TRUE`.

3. **Local Interface-Variante:** Basiert auf der zuerst genannten Variante. Der Schalter `<enable-call-by-reference>` hat für alle Beans den Wert `FALSE`. Die Kommunikation zwischen allen Beans des Vertrag-Packages wird über Local Interfaces abgewickelt.

Alle Varianten bearbeiteten 1000 Verträge. In der Tabelle in Abbildung 8.17 sind die gemessenen Werte der Varianten angegeben. An der processing time ist zu sehen, dass die Variante mit T3 bzw. mit Local Interfaces etwa 10 % schneller ist als die modifizierte EJB QL-Variante.

Methoden	EQL	T3	LocalInterfaces
		1000 // 120000	
Sucher	00:00:05.8	00:00:08.0	00:00:07.6
aendereSfKlasse	00:00:12.6	00:00:12.3	00:00:12.7
historisiereVertrag	00:01:12.4	00:00:58.4	00:00:56.4
findeSFKlasse	00:00:14.9	00:00:06.2	00:00:19.9
ermittleZuAbschlagStufen	00:01:11.2	00:00:56.0	00:00:54.8
ermittleSteuersatz	00:00:30.0	00:00:24.2	00:00:22.9
findeRisikoBuendel	00:00:31.9	00:00:22.2	00:00:23.1
historisiereRisikoBuendel	00:02:11.6	00:01:44.5	00:01:42.2
findeEinzelZuAbschlaege	00:01:31.8	00:01:11.5	00:01:13.5
findeElementarRisiken	00:03:01.9	00:02:41.2	00:02:45.6
historisiereElementarRisiko	00:06:41.9	00:05:11.3	00:05:05.9
ermittleMaxMindestBeitrag	00:07:33.9	00:07:33.7	00:07:25.7
ermittleBeitrag	00:07:34.2	00:07:33.2	00:07:26.6
berechnePraemie	00:00:42.2	00:00:30.8	00:00:28.6
processing time	**00:35:32.4**	**00:31:48.4**	**00:32:03.9**

Abbildung 8.17: Tabelle: Local Interfaces versus T3

Nicht alle gemessenen Methoden profitieren gleichermaßen von der vereinfachten Parameterübergabe. Deswegen ist es interessant, Methoden zu betrachten, bei denen »große« Datenmengen zu serialisieren sind. Im vorliegenden Beispiel sind das die hervorgehobenen Historisierungsfunktionen: »historisiereVertrag«, »historisiereRisikoBuendel« und »historisiereElementarRisiko«.

Bei der Umsetzung der Historisierungsfunktionen wurde der Entwurfsansatz mit Value-Objects gewählt. Ein Value-Object ist eine einfache Java-Klasse, die alle Attribute einer zugehörigen Bean enthält. Durch die Nutzung von Value-Objects wird vermieden, für das Lesen oder Setzen mehrerer Attribute der Bean mehrere Aufrufe über RMI (Remote Method Invocation) zu benötigen. Stattdessen wird *ein* Objekt erzeugt, das *alle* notwendigen Attribute enthält. Dieses Objekt lässt sich dann mit *einem* Aufruf über RMI übermitteln. Durch die Nutzung dieses Entwurfsmusters wird die Anzahl der RMI-Aufrufe reduziert, wobei der einzelne Aufruf ein größeres Datenvolumen zu transportieren hat (vgl. [46, S. 257ff]).

Das vergrößerte Datenvolumen bei RMI-Aufrufen bedeutet auch höhere Serialisierungskosten. Da bei den Historisierungsfunktionen also höhere Serialisierungskosten durch die Value-Objects zu erwarten sind, lässt sich die Einsparmöglichkeit bei entfallender Serialisierung an diesen Methoden besonders deutlich zeigen.

Allein bei der EJB QL-Variante müssen die benutzten Value-Objects bei der Parameterübergabe serialisiert und deserialisiert werden. Das entfällt bei den Varianten mit T3

und Local Interfaces. Die Darstellung der Werte als Balkendiagramm (Abbildung 8.18) verdeutlicht den Gewinn bezogen auf die serialisierungsintensiven Methoden.

Das Balkendiagramm zeigt nebeneinander drei Gruppen von Säulen. Jede Gruppe gehört zu einem Messpunkt: »historisiereVertrag«, »historisiereRisikoBuendel« und »historisiereElementarRisiko«. Säulen einer Farbe gehören zu einer Variante, wie in der Legende oben links zu sehen ist.

Am deutlichsten ist der Gewinn an Performanz bei der Historisierung der `Elementar-Risiko`-Bausteine zu sehen. Es wurde ca. 1 Minute und 30 Sekunden gegenüber der EJB QL-Variante gespart. Das sind etwa 25 % der Zeit für diese Methode (400 Sekunden gegenüber 300 Sekunden). Bei den anderen Methoden fällt das Verhältnis ähnlich aus, obgleich der absolute Gewinn weniger deutlich zu sehen ist. Der Unterschied zwischen der T3 und der Local Interfaces-Variante ist gering (kleiner 10 Sekunden).

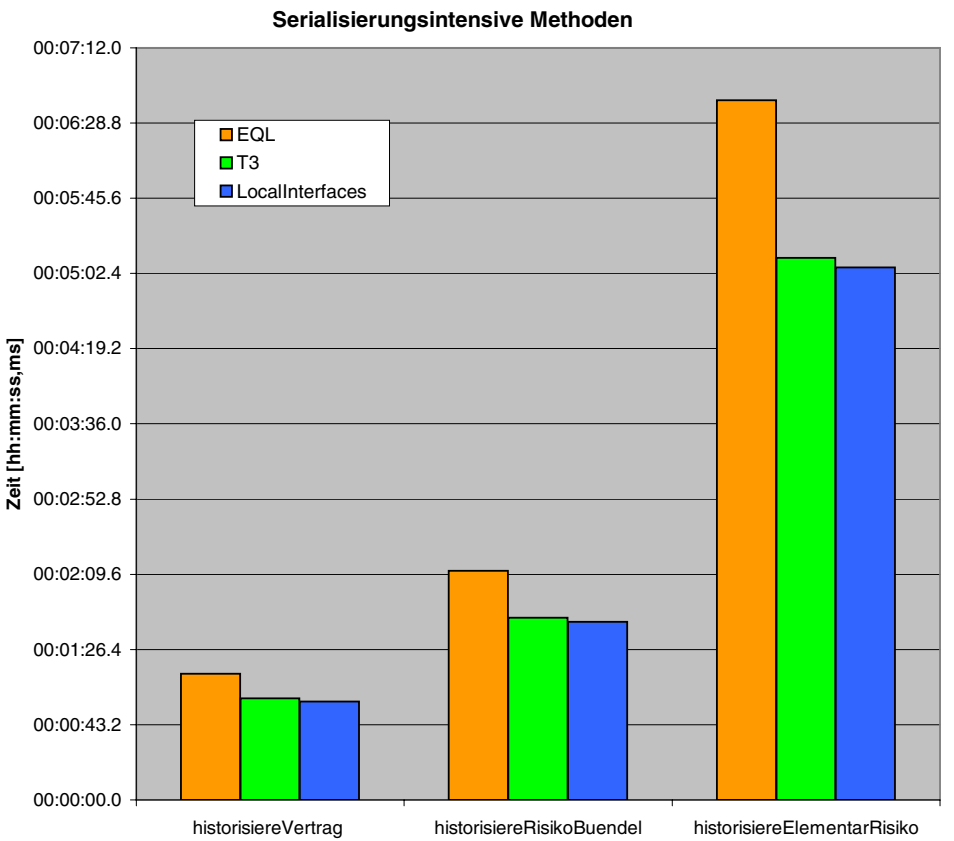


Abbildung 8.18: Balkendiagramm: Local Interfaces versus T3

8.4.4 Bewertung

Die Messungen zeigen, dass eine Variante, bei der Parameter per Referenz übergeben werden, schneller ist als eine Variante, bei der Parameter per RMI und damit per Serialisierung übergeben werden. Im dargestellten Beispiel macht das etwa 10 % der processing time aus. Bezogen auf Methoden, die umfangreiche Parameter serialisieren müssen, kann ein Performanzgewinn von etwa 25 % ermittelt werden.

Die Höhe des Performanzgewinns lässt sich *nicht* verallgemeinern. Dieser Wert ist abhängig von der Häufigkeit der Aufrufe zwischen Beans und der Art der Parameter, die dabei übergeben werden.

Anwendungen mit hohem Kommunikationsaufkommen gewinnen mehr durch diese Verbesserung als Anwendungen mit geringem Kommunikationsaufkommen (vgl. Antipattern »God« in [61]). Weiterhin lässt sich sagen, dass Anwendungen mit komplexen Übergabeparametern mehr gewinnen als Anwendungen, bei denen einfache, atomare Datentypen übergeben werden. Das können beispielsweise Anwendungen sein, die das Value-Object-Pattern [46] oder in Abwandlung dessen das bekannte Proxy-Pattern [26] benutzen.

Der Vergleich zwischen der Nutzung des T3-Protokolls und Local Interfaces zur Verbesserung der Kommunikation zeigt keinen relevanten Unterschied. Beide Verbesserungsansätze bringen einen ähnlichen Zugewinn. Dieses Messergebnis wird bestätigt von Nachrichten in den Foren des Applikationsserver-Herstellers [5].

Damit ist fraglich, ob sich der Aufwand lohnt, Local Interfaces zu implementieren. Während sich das T3-Protokoll in wenigen Augenblicken so konfigurieren lässt, dass die Parameter per Referenz übergeben werden, dauert die Umstellung einer bestehenden Komponente auf die Nutzung von Local Interfaces mit anschließendem Test zumindest ein paar Tage. Der Entwicklungsmehraufwand durch Local Interfaces ist eine Investition in die Portierbarkeit der Komponente. Eine Komponente, die fehlerfrei mit T3 arbeitet, wird auch fehlerfrei ohne T3 funktionieren — nur langsamer. Komponenten, die mit Local Interfaces umgesetzt sind, werden auch nach einer Portierung auf einen anderen Applikationsserver eine verbesserte Performanz haben. Einziger Vorteil der Local Interfaces ist also, dass sie Teil des EJB 2.0-Standards sind. Das T3-Protokoll ist es hingegen nicht.

Für Applikationsserver, die keine proprietäre Verbesserung der Parameterübergabe anbieten, stellt sich die zuvor genannte Frage nicht. Bei solchen Servern empfiehlt es sich, Local Interfaces in das Design der Anwendung aufzunehmen.

8.4.5 Design-Regel

Zur Erstellung von EJB-Anwendungen wird ein Komponentenschnitt durchgeführt. Ergebnis des Komponentenschnitts sind Mengen von zusammengehörigen Beans, die eine einzelne Komponente bilden. Kommunikation innerhalb der Komponente, d.h. Kommunikation von zusammengehörigen Beans, wird mit Local Interfaces abgewickelt. Kom-

munikation über Komponentengrenzen hinweg wird wie gewohnt per Remote Interfaces durchgeführt.

Bei Anwendungen auf Basis von *EJB 1.0 oder EJB 1.1* bleibt nur die Nutzung proprietärer und damit nicht portierbarer Möglichkeiten, die unnötige Serialisierung von Parametern zwischen Beans innerhalb einer JVM zu unterdrücken. Ein Beispiel dafür ist die beschriebene Nutzung des T3-Protokolls des Bea WebLogic 6.1.

8.5 Parallelisierung

Parallelisierung zur Beschleunigung der Abarbeitung einer Aufgabe ist in der Informatik weit verbreitet. Im folgenden Abschnitt wird erörtert, warum die nahe liegende Parallelisierung mit mehreren Session Beans nicht durchgeführt werden kann. Als Alternative wird der Entwurf mit »Message-Driven Beans«, dem in der EJB 2.0-Spezifikation neu hinzugekommenen Bean-Typ, erläutert und durchgeführt. Ausgehend von den Interaktionsdiagrammen in Abbildung 7.5 und Abbildung 8.5 diskutiert die Implementierungsvariante, an welchen Stellen des Geschäftsprozesses eine sinnvolle Parallelisierung möglich ist.

8.5.1 Literatur

Zum Thema parallele Abarbeitung, »concurrent processing«, findet sich in Artikeln zur EJB-Architektur kaum ein Verweis. In der Spezifikation wird darauf hingewiesen, dass es Ziel der Architektur ist, dem Bean-Entwickler »multi-threading« zu erleichtern [18, S. 29]. In der Tat koordiniert der Container mehrere Instanzen einer Bean, sodass sie zueinander parallel arbeiten können und mehrere Aufrufe an die gleiche Bean-Instanz in einzelnen Transaktionen isoliert werden (vgl. [16, S. 65ff; S. 103; S. 183ff]). Es scheint also nahe liegend, zur Abarbeitung auf mehrere Bean-Instanzen parallel zu verzweigen.

In einer nicht applikationsserverbasierten Java-Anwendung würden dazu mehrere Threads instanziiert werden. Der Prozess arbeitet in diesen Threads parallel. Ein Client des Applikationsservers ist eine Anwendung in dem beschriebenen Sinn. Im Client könnten mehrere Threads instanziiert werden und in jedem von ihnen könnte eine Bean auf dem Applikationsserver aufgerufen werden.

Genau dieses parallele Verzweigen durch Threads ist für Beans *innerhalb des Applikationsservers* nicht möglich: »The enterprise bean must not attempt to manage threads« [18, S. 493]. Der Grund dafür ist, dass durch Threads in der Bean dem Container die Möglichkeit genommen wird, die Laufzeitumgebung der Bean korrekt zu kontrollieren. Versuche, Threads in einer Bean zu instanziieren, werden vom Applikationsserver zur Laufzeit mit Exceptions quittiert.

Wenn die Bearbeitung ohne einen zusätzlichen Client umgesetzt werden soll, bieten sich noch »Message-Driven Beans« an [47]. Parallele Abarbeitung, »concurrent processing«, wird in der Spezifikation [18, S. 311] als wesentliches Ziel der Message-Driven Bean

beschrieben. Daher muss der Bean-Entwickler sich nicht um ihre Erzeugung und Koordination der Nebenläufigkeit kümmern.

»Message-Driven Beans« haben kein Home- oder RemoteInterface, über das sie erzeugt oder angesprochen werden. Sie werden in einem Instanzen-Pool vorgehalten und einer Warteschlange von Nachrichten zugewiesen. Per Container wird einer Instanz eine Nachricht zur Bearbeitung übergeben. Jede Instanz arbeitet in einem *eigenen* Thread. Stehen mehrere Nachrichten zur Bearbeitung an, so werden sie auf verschiedene Instanzen verteilt abhängig von der Größe des Instanzen-Pools. Überzählige Nachrichten verbleiben in der Warteschlange, bis eine Instanz sie übernehmen kann (vgl. Abschnitt 4.1).

Wesentlich ist dabei, dass der Container die Warteschlangensemantik gewährleistet und die Kontrolle der Nebenläufigkeit übernimmt. Der Bean-Entwickler muss lediglich das konkrete Muster implementieren, nach dem die Beans ihre Arbeit verrichten. Im Deployment Descriptor wird bestimmt, welcher Warteschlange die implementierten »Message-Driven Bean« zugehört. Dieser Warteschlange wird ein Arbeitsauftrag als Nachricht (`Message`) zugeschickt.

Die zuerst genannte Lösung, in einem Client *außerhalb* des Applikationsservers mehrere Threads zu erzeugen, ist schlecht zu warten und verstreut die Geschäftslogik über den Client und die Serverkomponenten. Load-Balancing ist im Client nachzuprogrammieren. Durch die Nutzung von »Message-Driven Beans« als Alternative zum ersten Ansatz wird dem Entwickler viel Arbeit bei der Koordination der Nebenläufigkeit abgenommen. Die Geschäftslogik wird vollständig auf dem Server implementiert. Daher wird der Ansatz mit Message-Driven Beans als Implementierungsvariante gewählt.

8.5.2 Übertragung und Entwurf

Bei der Übertragung des Ansatzes ist zuerst zu entscheiden, an welchen Stellen innerhalb der »Sfr-Umstufung« überhaupt verzweigt werden kann. Es werden voneinander unabhängige Teilschritte gesucht. Zur Analyse der Abarbeitung des Geschäftsprozesses werden die Sequenzdiagramme 7.5 und 8.5 herangezogen.

Bereits die einzelnen Geschäftsobjekte sind voneinander unabhängig. Auf das einzelne Geschäftsobjekt bezogen, kann weiterhin die Identifikation als Teilnehmer der Umstufung und die Umstufung als voneinander unabhängig betrachtet werden. Die Umstufung selbst besteht aus den Aufrufen von `stufeUm()` und `berechneBeitragNeu()`. Diese beiden Aufrufe müssen innerhalb der gleichen Transaktion stattfinden, sodass hier nicht parallelisiert werden kann. Innerhalb der Beitragsberechnung kann die Ermittlung der einzelnen Summanden des Beitrags parallelisiert werden. Im Folgenden werden die drei möglichen Verzweigungspunkte genauer diskutiert:

1. Identifikation der Teilnehmer
2. Umstufung
3. Beitragsberechnung der ElementarRisiken

Bei der *Identifikation der Teilnehmer* wird jede aktuelle Vertragsinstanz betrachtet. Falls diese den Suchkriterien genügt, wird sie in eine Ergebnismenge eingefügt. Eine parallelisierte Version des `Sucher` würde alle betrachteten Instanzen von `Vertrag` in den Applikationsserver laden. Für diesen Schritt bietet bereits die in Abschnitt 8.2 beschriebene Variante eine bessere Lösung, sodass hier darauf verzichtet wird, mit »Message-Driven Beans« zu parallelisieren.

Bei der »*Umstufung*« ist es nicht wichtig, in welcher Reihenfolge die Geschäftsobjekte bearbeitet werden. Welches Ergebnis die Umstufung eines einzelnen Vertrags hat, ist für die Umstufung der anderen Verträge ebenso unwesentlich. An dieser Stelle wird deshalb parallel auf mehrere Bearbeitungsstränge verzweigt.

Innerhalb eines einzelnen Vertrags ist es denkbar, bei der Berechnung der Beiträge auf Ebene des `RisikoBuendel` und bei der Berechnung der Beiträge auf Ebene des `ElementarRisiko` zu verzweigen. Die Ergebnisse dieser Berechnungen werden als Beitrag der nächsthöheren Vertragsebene addiert, z.B. ergeben alle Beiträge der `RisikoBuendel` in der Summe den Beitrag des Vertrags. Das Ergebnis der Berechnung muss also an zentraler (`Vertrag.setPraemie(...)`) Stelle abgelegt werden. An dieser Stelle bildet sich ein Flaschenhals. In die Message-Driven Beans muss zusätzliche Logik eingebaut werden, die so lange die Methode `setPraemie(...)` anfragt, bis die Instanz von `Vertrag` den Aufruf bearbeiten kann. Bis dahin wartet die Message-Driven Bean. Diese Wartezeit schmälert dann den Zugewinn durch die Parallelisierung. Daher wird innerhalb des einzelnen Vertrages nicht parallelisiert.

In Abwandlung des Sequenzdiagramms aus Abbildung 7.5 ergibt sich das Diagramm in Abbildung 8.19. Von einem ManagementClient wird ein `Sucher` erzeugt und danach ein `BatchRahmen`. Der `Batchrahmen` erhält im Unterschied zur vorangegangen Variante nur den `Sucher`. Über den Sucher ist der JNDI-Name der MessageQueue »messageQueue« bekannt, über die der Pool von parallel arbeitenden Message-Driven Beans angesprochen wird. Wird `BatchRahmen.start()` aufgerufen, so ermittelt der `Sucher` »sfrTeilnehmer« die Teilnehmer. Er übergibt sie als `Collection` von Remote Interfaces. Jedes Remote Interface wird als Message an die Message-Queue geschickt. Die Message-Queue übergibt die Messages einer Instanz der zugehörigen Message-Driven Bean. Die Message-Driven Bean übernimmt die weitere Verarbeitung. Dieser Sachverhalt ist im vorliegenden Diagramm verkürzt dargestellt. Der BatchRahmen kann die Message-Driven Bean nicht direkt mit: `onMessage(javax.jms.Message)` ansprechen. Wie oben erwähnt, übernimmt der Container die Verwaltung der Warteschlange und der Message-Driven Beans.

Die einzelne Bean stößt schließlich innerhalb einer neuen Transaktion auf dem `Vertrag`, dessen Remote Interface in der Message übermittelt wurde, die Methoden `stufeUm()` und `berechnePrämie()` an.

Damit ist die angestrebte Parallelisierung erreicht. Durch die »Message-Driven Beans« ist es möglich geworden, aus einer »SessionBean« parallel mehrere Instanzen von EntityBeans vom Typ Vertrag arbeiten zu lassen. Der Grad der Parallelität lässt sich durch die Anzahl der Message-Driven Beans konfigurieren.

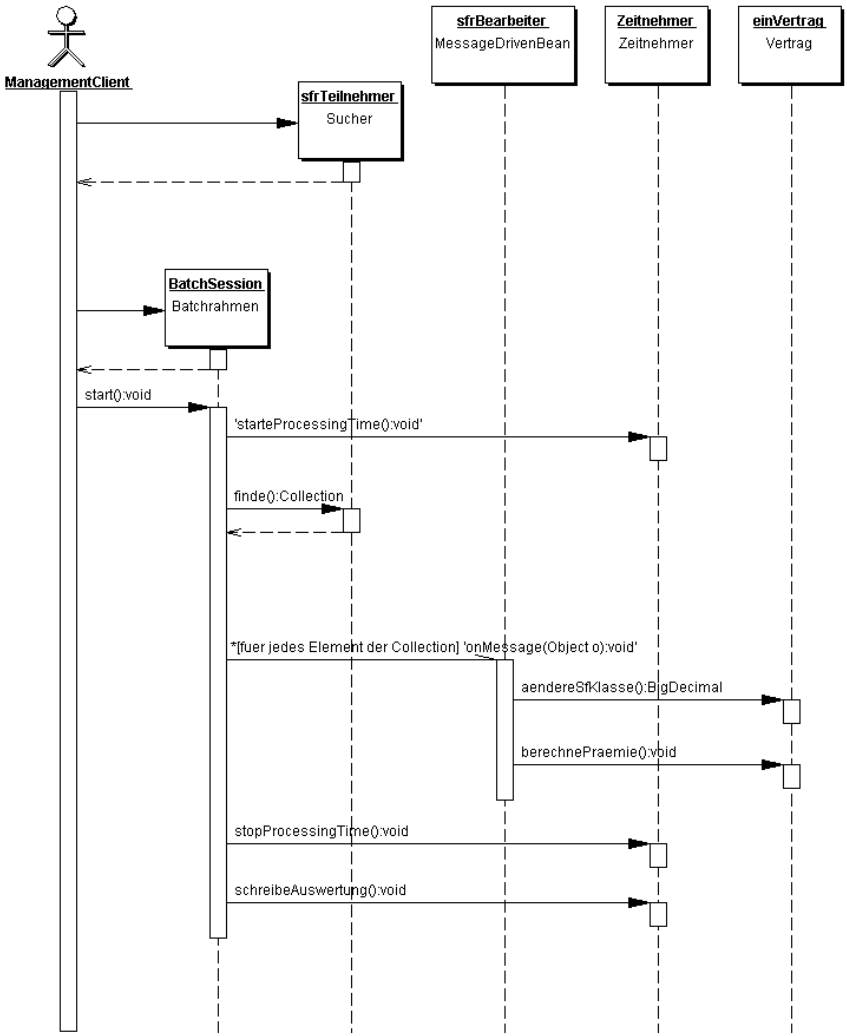


Abbildung 8.19: Sequenzdiagramm: Sfr-Umstufung mit `Sucher` und parallelisiertem `Bearbeiter`

Die praktische Umsetzung ist das Thema der folgenden Abschnitte. Zuerst wird im Applikationsserver eine MessageQueue erzeugt mit dem JNDI-Namen `de.adesso.mdb.vertrag.SfrUmstufungQueue`. Im Deployment Descriptor **ejb-jar.xml** wird ein Block zur Beschreibung der Message-Driven Bean angelegt:

```
<message-driven>
  <ejb-name>SfrUmstufungBearbeiterBean</ejb-name>
  <ejb-class>
    de.adesso.mdb.vertrag.SfrUmstufungBearbeiterBean
```

```
  </ejb-class>
  <transaction-type>Container</transaction-type>
  <acknowledge-mode>auto-acknowledge</acknowledge-mode>
  <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
  </message-driven-destination>
</message-driven>
```

Listing 8.12: Deklaration einer Message-Driven Bean im Deployment Descriptor

Dieser Block deklariert den Namen der Message-Driven Bean »SfrUmstufungBearbeiterBean«, durch den sie in den anderen DeploymentDescriptoren referenziert werden kann. Es wird festgelegt, welche Klasse die Funktionalität enthält (`de.adesso.mdb.vertrag.SfrUmstufungBearbeiterBean`) und in der darauf folgenden Zeile, dass der Container das Transaktionsmanagement übernimmt.

Schließlich legt der Deployment Descriptor fest, in welcher Art der Messageserver von einem Client über den Erhalt einer Nachricht informiert wird. Mit `auto-acknowledge` wird festgelegt, dass diese Information an den Message-Server automatisch erfolgt. Der letzte Parameter für `destination-type` gibt an, in welcher Art die Nachrichten verteilt werden: `javax.jms.Queue` bestimmt, dass eine Punkt-zu-Punkt-Verteilung gewählt wird, d.h. genau *ein* Empfänger wird benachrichtigt.

Der Deployment Descriptor **weblogic-ejb-jar.xml** stellt die Verbindung zwischen der im Applikationsserver angelegten Queue und dem im Deployment Descriptor **ejb-jar.xml** deklarierten Message-Driven Bean her.

```
<weblogic-enterprise-bean>
<ejb-name>
   SfrUmstufungBearbeiterBean</ejb-name>
<message-driven-descriptor>
  <pool>
   <max-beans-in-free-pool>
       4 </max-beans-in-free-pool>
   <initial-beans-in-free-pool>
       4 </initial-beans-in-free-pool>
  </pool>
  <destination-jndi-name>
      de.adesso.mdb.vertrag.SfrUmstufungQueue
  </destination-jndi-name>
</message-driven-descriptor>
<jndi-name>
  de.adesso.mdb.vertrag.SfrUmstufungBearbeiterBean
</jndi-name>
</weblogic-enterprise-bean>
```

Listing 8.13: Server-spezifische Deklaration einer Message-Driven Bean

`<ejb-name>SfrUmstufungBearbeiterBean</ejb-name>` stellt den Bezug zu der Deklaration in **ejb-jar.xml** her. Die Parameter `max-beans-in-free-pool` und `initial-beans-in-free-pool` geben an, wie viele Beans und damit auch Threads maximal bzw. anfänglich parallel arbeiten sollen. Die letzten beiden Einträge bezeichnen die JNDI-Namen, unter denen die Queue bzw. die verarbeitenden Message-Driven Beans zu finden sind.

Weitere Details zum Thema »Messaging« finden sich in [47]. Die Konfiguration des Applikationsservers zur Nutzung von Message-Driven Beans ist in [4] ausführlich dargestellt.

Die Dokumentation zum Applikationsserver stellt heraus, dass der erreichbare Grad an Parallelität abhängig ist von den beiden folgenden Faktoren:

1. **Isolierungsgrad der Transaktionen:** Durch die Isolierungsgrade wird angegeben, wie sehr die Daten einer Transaktion von *anderen* Transaktionen abgeschirmt werden. Da Suchanfragen länger dauern und meist auf mehr Daten zugreifen als andere Anfragen, halten Suchanfragen mehr und länger Daten abgeschirmt von anderen Transaktionen. Sie verzögern damit die anderen Anfragen. Wird der Isolierungsgrad der Suchanfrage gesenkt, sodass andere Anfragen die Daten der Suchanfrage lesen oder sogar ändern können, wird eine Verzögerung vermieden. Die Suchanfrage und die anderen Anfragen können gleichzeitig auf denselben Daten arbeiten [6, S. 214ff].

   Für die Beispielapplikation trifft das nur auf den Zugriff auf die `SfKlasse` zu, da viele Verträge auf die gleiche Instanz von `SfKlasse` verweisen. Bei einem hohen Serialisierungsgrad könnte diese Instanz von `SfKlasse` nur in einer Transaktion genutzt werden. Die anderen Abarbeitungsstränge müssten warten, bis diese Instanz zum Ende der Transaktion wieder freigegeben wird.

   Da die Instanzen von `SfKlasse` sich als Teil einer Produktdefinition nur sehr selten ändern, wird im Deployment Descriptor **weblogic-ejb-jar.xml** der schwächste Isolierungsgrad gewählt.

```
<transaction-isolation>
    <isolation-level>
        TRANSACTION_READ_UNCOMMITTED
    </isolation-level>
    <method>
        <description></description>
        <ejb-name>VertragBean</ejb-name>
        <method-name>stufeUm</method-name>
            <method-params>
                  <method-param>boolean</method-param>
            </method-params>
    </method>
</transaction-isolation>
```

Listing 8.14: Isolierungsgrad einer Transaktion

   Durch Setzten des Isolierungsgrads auf `TRANSACTION_READ_UNCOMMITTED` für die Methode `Vertrag.stufeUm` wird festgelegt, dass die genannte Methode auch Daten lesen

darf, die gerade an anderen Transaktionen teilnehmen. Weitere Details zum Thema Isolierungsgrade und Enterprise JavaBeans enthält [46, S. 232ff].

2. **Anzahl zur Verfügung stehender Datenbankverbindungen:** Für jede parallel stattfindende Transaktion muss eine eigene Verbindung zwischen dem Applikationsserver und der Datenbank existieren. Sind nicht genug gleichzeitige Verbindungen möglich, muss eine Transaktion warten, bis eine andere Transaktion beendet ist und eine Verbindung freigibt. Die maximale Anzahl von Datenbankverbindungen wird in der Konfiguration des Applikationsservers auf 100 gesetzt. Bezüglich der Datenbankverbindungen könnten vom Appliaktionsserver 100 *gleichzeitige* Transaktionen auf der Datenbank ausgeführt werden.

8.5.3 Messergebnis

Die folgende Messung vergleicht die modifizierte EJB QL-Variante mit verschiedenen Varianten, die Message-Driven Beans benutzen. Dabei unterscheiden sich letztere allein durch die *Anzahl* der Message-Driven Beans, die in einem Pool von Instanzen vorgehalten werden.

Diese Anzahl gibt nach den Vorüberlegungen auch an, auf wie viele parallele Abarbeitungsstränge die Anwendung verzweigt. Die Anzahl der Message-Driven-Bean-Instanzen wird im Deployment Descriptor **weblogic-ejb-jar.xml** mit dem Parameter `<max-beans-in-free-pool>` eingestellt.

Abbildung 8.20 zeigt eine Tabelle, in der zu jeder Variante die processing time angegeben ist. Als erste Variante ist die EJB QL-Variante dargestellt. Darunter ist mit »1 MDB« die Variante dargestellt, bei der genau eine Message-Driven Bean die Verträge bearbeitet. »2 MDB« bezeichnet die Variante, bei der genau zwei Message-Driven Beans die Verträge parallel verarbeiten usw.

Variante 1000 // 120000	processing time
EQL	00:35:32.4
1 MDB	00:33:12.0
2 MDB	00:32:22.0
3 MDB	00:33:20.0
4 MDB	00:33:40.0
5 MDB	00:33:16.0
10 MDB	00:34:48.0

Abbildung 8.20: Tabelle: processing time in Abhängigkeit von der Anzahl Message-Driven Beans

Alle Varianten haben eine processing time zwischen 33 und 36 Minuten. Das widerspricht den erwarteten Ergebnissen. Durch die Parallelisierung sollte sich die Bearbeitungszeit drastisch verringern. Im Idealfall sollte sich die processing time durch die Anzahl der

Message-Driven Beans teilen. Bei zwei Instanzen ergibt sich die halbe processing time, bei drei Instanzen ein Drittel der Ausgangszeit und so fort. Für die durchgeführten Messungen erreichen alle Varianten nur die Zeit einer sequenziellen Abarbeitung.

Wie im vorangegangenen Abschnitt erwähnt, sind bei der Implementierung auch Datenbankverbindungen und die Sperrsituation auf den Zeilen berücksichtigt worden, sodass es an diesen Stellen zu keinen Performanzengpässen gekommen ist.

Über den Windows-NT Taskmanager können weitere Messgrößen ermittelt werden, die bei der Beantwortung helfen, warum kaum eine Performanzverbesserung stattgefunden hat. Die Auslastungskurve des Prozessors gibt ein Indiz dafür ab, dass die Hardware des Testrechners keine parallele Verarbeitung des Geschäftsprozesses unterstützt. Abbildung 8.21 zeigt eine repräsentative Lastkurve des Prozessors bei sequenzieller Abarbeitung. Die Last des Prozessors liegt zwischen 80 % und 95 %.

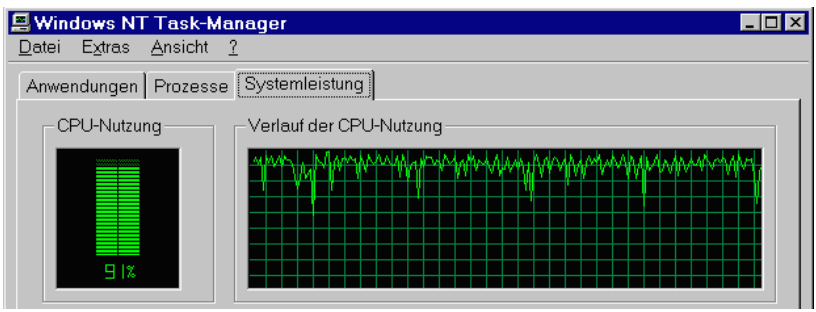


Abbildung 8.21: NT Taskmanager: Lastkurve des Prozessors für die EJB QL-Variante

Davon erzeugt der Applikationsserver etwa 75–90 % und die Datenbank etwa 5–15 %. Diese Information kann ebenfalls im Taskmanager abgelesen werden, wie in Abbildung 8.22 zu sehen ist.

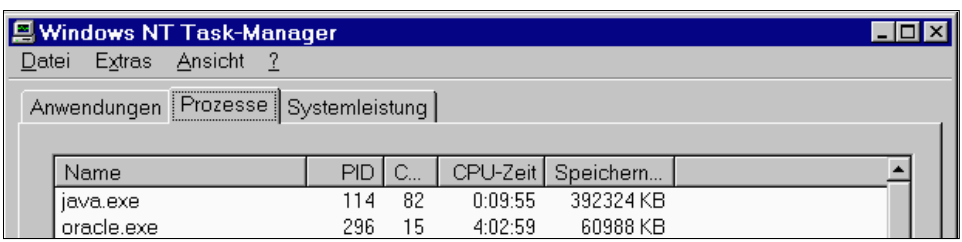


Abbildung 8.22: NT Taskmanager: anteilige Auslastung des Prozessors für die EJB QL-Variante

Der Prozessor ist bereits bei sequenzieller Abarbeitung fast ausgelastet. Abbildung 8.23 zeigt die Lastkurve bei Einsatz von 2 Message-Driven Beans.

Während die Lastkurve für die EJB QL-Variante gebirgig aussieht, ergibt sich eine gerade Linie bei 100 % als Lastkurve für die Variante mit 2 Message-Driven Beans. Der Prozessor ist also völlig ausgelastet. Im Bereich der Performanz-Modellierung wird dann

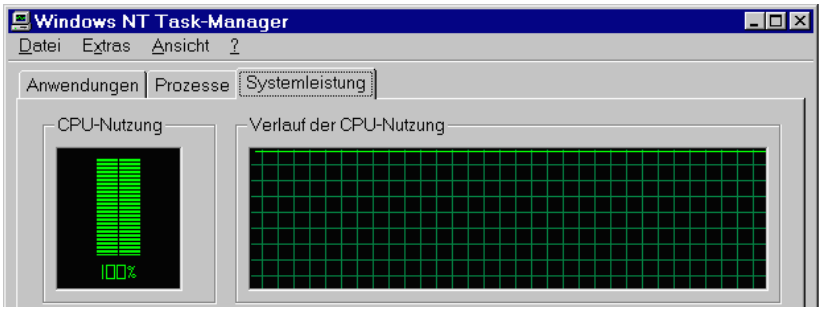


Abbildung 8.23: NTTaskmanager:Auslastung des Prozessors für die EJB QL-Variante

von *gesättigten* Ressourcen gesprochen. Weitere Verbesserungen sind nur erreichbar, falls die Sättigung aufgelöst oder vermieden werden kann [40]. Das kann beispielsweise durch eine Steigerung der Kapazität der Ressource passieren.

Die Lastkurve ist damit ein Indiz dafür, dass die Hardware an der unerwartet schlechten Performanz schuld ist.

8.5.4 Bewertung

Das Ergebnis der Messungen zeigt für das vorliegende Beispiel keine wesentliche Verbesserung der Performanz. Der Grund für dieses enttäuschende Ergebnis ist die unzureichende Hardware, auf der die Messläufe durchgeführt werden. Um diese Vermutung zu bestätigen, müssten alle Messungen auf einem leistungsfähigeren Rechner mit mehreren Prozessoren oder auf einem Verbund von Rechnern wiederholt werden. Der Praxisbericht [67] unterstreicht unabhängig von den hier behandelten Message-Driven Beans, dass die Verteilung von Datenbank und Applikationsserver auf verschiedene Rechner besonders wichtig ist für die Performanz. Ein weiteres Indiz für die unzureichende Hardware gibt die finale Variante in Abschnitt 8.7. In dieser Variante ist es gelungen, die Prozessorlast durch andere Verbesserungen zu mindern. In der finalen Variante fallen daher die Verbesserungen durch die Message-Driven Beans deutlicher aus.

Durch den Entwurf werden keine anderen Geschäftsprozesse negativ beeinflusst. Der Aufwand, Message-Driven Beans nachträglich zu implementieren, hat für dieses Beispiel wenige Tage gedauert. Bei der Nutzung von Message-Driven Beans sollte zusätzliche Zeit für umfangreiche Testläufe vorgesehen werden. Diese Testläufe sind notwendig, um die richtige Kombination von Konfigurationsparametern für eine optimale Nutzung von Message-Driven Beans zu ermitteln.

Vermutlich verbessern Message-Driven Beans die Skalierbarkeit einer Anwendung, soweit ausreichend Hardware zu Verfügung steht.

8.5.5 Design-Regel

Message-Driven Beans können benutzt werden, um serverseitig auf parallele Verarbeitungsstränge zu verzweigen. Diese Beans sind *allein* für die Verzweigung auf mehrere

Stränge zuständig. Die Geschäftslogik verbleibt weiterhin in den Entity Beans und Session Beans.

Für *EJB 1.0- und 1.1*-basierte Anwendungen kann nur außerhalb der Beans parallelisiert werden. Bereits clientseitig sind mehrere Threads zu öffnen, die serverseitig Beans ansprechen. Load-Balacing und Warteschlangensemantik sind ebenfalls Client-seitig auszuprogrammieren.

8.6 Arbeitsaufteilung zwischen Online- und Batchbetrieb

Bevor ein Geschäftsobjekt in einem Batchlauf bearbeitet werden kann, muss es in das System eingegeben worden sein. Das geschieht in den allermeisten Fällen durch einen Geschäftsvorfall, bei dem ein Sachbearbeiter die notwendigen Daten eingibt.

Je nach Auslastung des Systems im Online-Betrieb kann nach Arbeitsschritten gesucht werden, die aus dem Batchlauf in den Online-Geschäftsvorfall vorgezogen werden können. Durch den Online-Geschäftsvorfall wird das Geschäftsobjekt so weit wie möglich für den Batchlauf vorbereitet.

Häufig werden als Ergebnis dieser Vorbereitung zusätzliche Attribute gespeichert, deren Wert aus anderen Attributen des gleichen oder eines assoziierten Objekts zu ermitteln ist. Aus der reinen Sicht der Datenmodellierung sind solche transienten Attribute überflüssig, da sie jederzeit aus den anderen Attributen ermittelt werden können. Damit ist die Arbeitsverteilung zwischen Online- und Batchverarbeitung in diesen Fällen als Denormalisierung, d.h. Einführung kontrollierter Redundanz, zu verstehen.

Abschnitt 8.6 diskutiert, in welchen Fällen kontrollierte Redundanz angewendet werden kann, und führt dies beispielhaft vor. Die Veränderungen an den Beans wie auch an der Datenbank werden erläutert.

8.6.1 Literatur

Die Frage, welche Arbeitsschritte aus den Batchprozessen in die Online-Prozesse verlagert werden können, ist eine Frage des Designs der Anwendung. Um die zu verlagernden Schritte zu identifizieren, müssen daher die Sequenzdiagramme zu den einzelnen Methoden, die im Batchlauf benutzt werden, im Detail analysiert werden. Wird ein Schritt ausgelagert, so ist das Ergebnis dieses Schritts in einem oder mehreren *redundanten* Attributen zu speichern, bis der Batchprozess auf dieses Ergebnis zugreift. Ein Attribut eines Objekts wird als redundant bezeichnet, wenn es funktional abhängig ist von anderen Attributen. Das heißt es gibt eine Möglichkeit, aus anderen Attributen den Wert des redundanten Attributs herzuleiten. Ändert sich eines der anderen Attribute, so muss sich gemäss der funktionalen Abhängigkeit auch das redundante Attribut ändern. Daher ist bei der Nutzung kontrollierter Redundanz die Änderungshäufigkeit der Attribute zu berücksichtigen, auf denen der Wert des redundanten Attributs beruht. Nur wenn sich

die Daten selten ändern, von denen das redundante Attribut abhängt, lohnt es sich, einen Arbeitsschritt vorzuverlagern.

Technisch entspricht das der Nutzung »kontrollierter Redundanz« [22, S. 435ff]. Es werden zur Beschleunigung zusätzliche Attribute gespeichert, die funktional von anderen Attributen der gleichen Instanz abhängig sind. Bei deutlicheren Ausprägungen dieses Konzepts werden Spalten einer Tabelle (d.h. Attribute eines Objekts) in einer anderen Tabelle als direkte Kopie wiederholt. Dadurch kann bei manchen Funktionen der Zugriff auf mehrere Tabellen eingespart werden.

Die Abhängigkeit zwischen den Spalten muss dabei berücksichtigt werden. Bei einer Veränderung der ursprünglichen Spalte müssen auch alle Kopien der Spalte aktualisiert werden. Damit erklärt sich auch, warum so etwas nur auf Daten mit einer geringen Änderungshäufigkeit eingesetzt werden sollte. Ändern sich die dem redundanten Attribut (der redundanten Spalte) zugrunde liegenden Daten, so ist dafür zu sorgen, dass das zusätzlich gespeicherte, redundante Attribut ebenfalls verändert wird. Der folgende Abschnitt 8.6.2 klärt diesen Sachverhalt anhand des Beispielgeschäftsprozesses.

Neben der Möglichkeit, Arbeitsschritte aus dem Batchlauf in einen Online-Geschäftsvorfall zu verlagern, gibt es auch die Möglichkeit, Arbeitsschritte in *vorbereitende* Batchläufe auszulagern und schließlich die dritte Möglichkeit, einen einzelnen Lauf in mehrere kleine Läufe herunterzubrechen. Die letzte Möglichkeit wird vor allem benutzt, wenn die Zeitscheiben, die für Batchläufe genutzt werden können, zu gering sind [28]. Die Schwierigkeit bei der Nutzung redundanter Attribute wurde bereits erläutert. Die Schwierigkeit bei der Nutzung veränderter Zeitrhythmen besteht in der organisatorischen Einbettung des Systems in die vorhandenen Randsysteme. Die vorhandenen Randsysteme (z.B. Druckstraße, Inkasso-Schnittstelle) müssen nun wöchentlich beliefert werden. Das kann mit Belieferungszeiten anderer Bestandsführungen kollidieren, wenn Bestandsführungen verschiedener Sparten dasselbe Randsystem beliefern müssen.

8.6.2 Übertragung und Entwurf

Bei der Diskussion, wie die Beispielanwendung sinnvoll kontrollierte Redundanz einsetzen kann, wird zuerst die Funktionaliät des `Sucher` betrachet. Im nächsten Schritt werden die Methodenaufrufe behandelt, die in den Kontext der `Bearbeiter`-Funktionalität gehören. Die Unterscheidung von `Sucher` und `Bearbeiter`-Funktionalität wird in Abschnitt 7.3 beschrieben.

Zur Analyse der `Sucher`-Funktionalität sei nochmals das Sequenzdiagramm der Basisvariante in Abbildung 8.4 betrachtet. Innerhalb der Methode `pruefeAutomatischeSfr–Umstufung()` werden fünf Abfragen ausgeführt.

1. **Aktuelle Gültigkeit:** Zuerst wird auf aktuelle Gültigkeit geprüft. Dieser Schritt vergleicht lediglich das aktuelle Datum mit dem Attribut `hGueltigBis` des Vertrags, sodass hier keine Ersparnis erreicht werden kann.

2. **Hauptfälligkeit:** Im zweiten Schritt wird die nächste Hauptfälligkeit errechnet, um festzustellen, ob das Geschäftsobjekt im Monat des Batchlaufs hauptfällig ist. Dieses

Attribut wird von vielen Geschäftsvorfällen genutzt und ist während der Abrechnung zur Hauptfälligkeit einfach zu aktualisieren. Der zu erwartende Nutzen für den Beispielgeschäftsvorfall ist gering. Da der Anpassungsaufwand jedoch ebenfalls gering ist, wird die nächste Hauptfälligkeit als zusätzliches Attribut gespeichert. Es wird bei Neuanlage des Vertrags gesetzt und ab dann mit jedem Abrechnungslauf zur Hauptfälligkeit aktualisiert. Dieses Attribut ist insofern redundant, als es direkt von dem Attribut `hGueltigAb` abhängig ist. Die erste Hauptfälligkeit wird eine Versicherungsperiode, zumeist ein Jahr, nach In-Kraft-Treten des Vertrags `hGueltigAb` erreicht. Alle weiteren Hauptfälligkeiten folgen periodisch. Ein Vertrag mit einer Laufzeit von fünf Jahren hat also vier Hauptfälligkeiten.

3. **Automatische Umstufbarkeit:** Schritt drei des `Suchers` überprüft allein das Attribut `Vertrag.automatischeSfrUmstufung`, sodass hier keine Verbesserung zu erreichen ist.

4. **Letzte Sfr-Umstufung:** Schritt vier des `Suchers` überprüft allein das Attribut `Vertrag.letzteSfrUmstufung`; auch hier ist keine Verbesserung zu erreichen.

5. **Assoziierte `SfKlasse`:** Im letzten Schritt wird die Hochstufungsklasse von der aktuell assoziierten `SfKlasse` ermittelt. Ist sie vorhanden, d.h. ungleich `null`, so kann dem Vertrag eine höhere Stufe zugewiesen werden. Er hat noch nicht die beste `SfKlasse` erreicht. Hier lässt sich der Zugriff auf die `SfKlasse` sparen, indem eine zusätzliche Referenz auf die Hochstufungsklasse der aktuellen `SfKlasse` gespeichert wird. Die Arbeitsschritte des Suchers benötigen dann nur noch Attribute des Vertrags. Da Rabattstaffeln als Teil der Produktdefinition aus rechtlichen Gründen nicht geändert werden dürfen, sobald Verträge auf ihnen beruhen, ist keine weitere Logik zusätzlich zu implementieren. Einzig, wenn der Vertrag eine neue `SfKlasse` zugewiesen bekommt, ist gleichzeitig die Referenz auf die neue Hochstufungsklasse zu aktualisieren. Damit ist gezeigt, dass die Änderungshäufigkeit dieses Attributs der Nutzung als redundantem Attribut nicht widerspricht.

Das aktualisierte Sequenzdiagramm ist in Abbildung 8.24 dargestellt. Es entspricht weitestgehend der Darstellung in Abbildung 8.4, sodass nur die Abweichungen erwähnt werden.

Die Methode `ermittleHauptFälligkeit()` wurde durch die Methode `getHauptFälligkeit()` ersetzt, da die Hauptfälligkeit als zusätzliches Attribut gespeichert wird. Die Methode `ermittleHochstufungsKlasse()` wurde durch `getHochstufungsKlasse()` ersetzt. Die Referenz zur Hochstufungsklasse verweist jetzt direkt zur Hochstufungsklasse. Dadurch muss auf keine anderen Instanzen als den Vertrag selbst zugegriffen werden.

Die Veränderung der `Sucher`-Funktionalität ist damit abgeschlossen. Nun wird die *`Bearbeiter`-Funktionalität* betrachtet. Sie enthält die Umstufung mit anschließender Beitragsneuberechnung. Ausgangsbasis ist die Art der Bearbeitung anhand des Sequenzdiagramms in Abbildung 8.5.

Die Methode `stufeUm()` muss angepasst werden, da zusätzlich das Attribut für die Hochstufungsklasse aktualisiert werden muss. Die Gründe dafür liegen in der veränderten

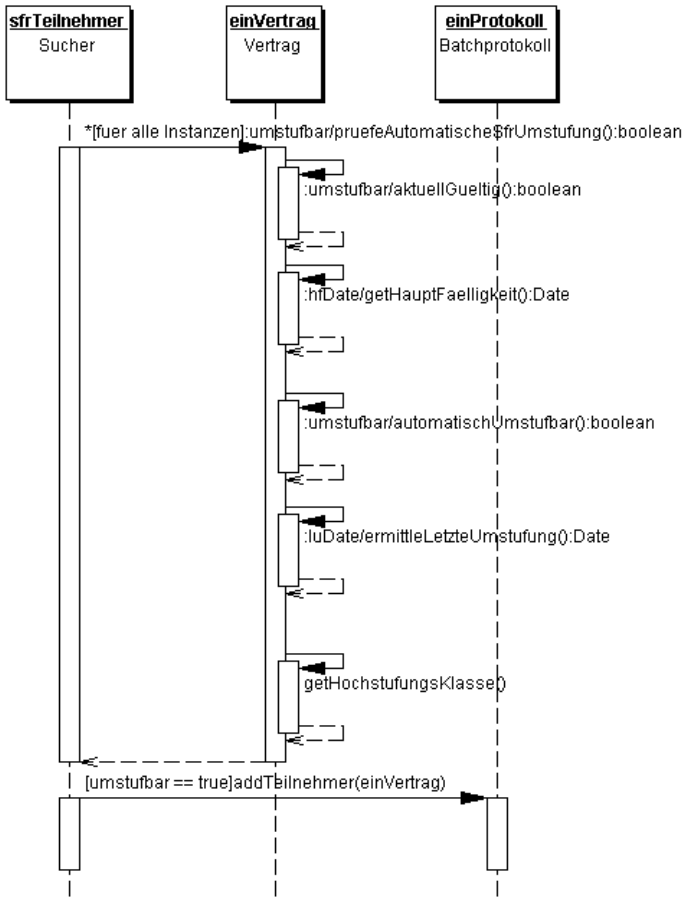


Abbildung 8.24: Sequenzdiagramm: Identifizieren der beteiligten Geschäftsobjekte mit Redundanz

`Sucher`-Funktion, die zuvor beschrieben wurde. Die Abbildung 8.25 zeigt weiterhin einen einzigen Zugriff auf die `SfKlasse`. Diesmal jedoch, um das Attribut für die Referenz auf die Hochstufungsklasse zu aktualisieren. Die Verbesserung des `Sucher` hat damit keine Verschlechterung für die weitere Bearbeitung nach sich gezogen.

In der Methode `berechneBeitragNeu()` finden Zugriffe auf die Tabellen von `ZuAbschlag–Stufe` und `EinzelZuAbschlag` statt. Stünde die genaue Anzahl assoziierter Instanzen fest, sodass einem `RisikoBuendel` stets genau zwei `EinzelZuAbschlag`-Instanzen assoziiert wären, so könnte man die benötigten Spalten von dort der Vertragstabelle hinzufügen. In [37] und [12] kann die beschriebene Art, aggregiertes Objekt und aggregierende Bestandteile in eine *einzelne* Tabelle abzubilden, ausführlich nachgelesen werden.

Das Beispielsystem soll jedoch eine variable Anzahl von `ZuAbschlägen` verarbeiten können, sodass an dieser Stelle der Zugriff auf weitere Tabellen nicht eingespart werden kann.

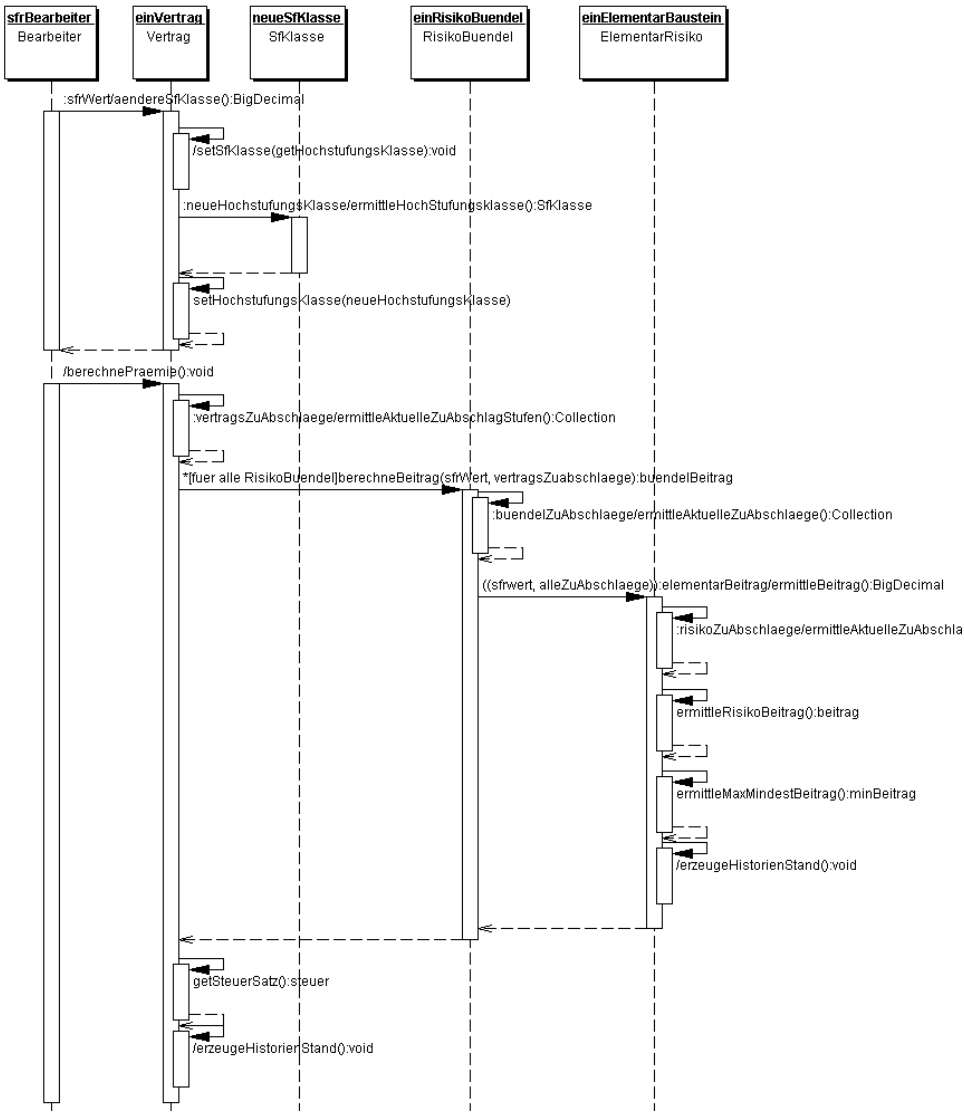


Abbildung 8.25: Sequenzdiagramm: Umstufung mit Redundanz

Der Zugriff auf die `TarifBean` in `ermittleBeitrag()` und `ermittleMaxMindestBeitrag()` kann eingespart werden. Die Attribute `beitrag` und `maxMindestBeitrag` enthalten dann das Ergebnis der Errechnung des Beitrags aus dem Tarif. Die Ermittlung des Beitrags aus dem Tarif kann durch einen vorbereitenden Batchlauf passieren oder direkt bei der Anlage in einem Online-Geschäftsvorfall stattfinden.

Um endgültig zu entscheiden, ob diese Attribute redundant gespeichert werden sollen, muss die Änderungshäufigkeit der zugrunde liegenden Attribute bedacht werden. Die Attribute `beitrag` und `maxMindestBeitrag` sind funktional abhängig vom Tarifwerk und

von den Risikomerkmalen, die im `ElementarRisiko` gespeichert sind. Der tarifliche Beitrag kann sich daher auf zweierlei Arten ändern.

1. **Regulierungsbogen:** Dies tritt auf eine Art ein, wenn sich das versicherte Risiko ändert. Damit erhält ein Risikomerkmal einen anderen Wert. Der Versicherungsnehmer hat dies seiner Versicherung durch einen Prämienregulierungsbogen mitzuteilen, woraufhin der Vertrag auf Basis des gleichen Tarifs und geänderter Risikomerkmale neu berechnet wird.

2. **Tarifanpassung:** Die zweite Art, nach der sich Beiträge ändern können, ist die »Tarifanpassung«. Abhängig von gewissen Größen wie Inflation wird von unabhängiger Stelle ein Faktor bestimmt, um den alle Versicherungen ihre Tarife anheben können. Unabhängig von der Hauptfälligkeit eines Vertrags werden dann die Vertragsbeiträge einmal pro Jahr nach der Tarifanpassung neu ermittelt.

Die vom Tarif und den Risikomerkmalen abhängigen Attribute `beitrag` und `maxMindest–Beitrag` werden also nur gezielt von wenigen Geschäftsprozessen beeinflusst, sodass sie redundant in zusätzlichen Attributen der `ElementarRisiko`-Instanzen gespeichert werden könnten. Um das Datenmodell an dieser Stelle nicht ändern zu müssen, wird für diese Variante unterstellt, dass es einen vorbereitenden Batchlauf gibt, der die Attribute `beitrag` und `maxMindestBeitrag` mit den unmodifizierten Wert aus dem Tarif belegt.

In Abbildung 8.25 ist kein Zugriff auf einen Tarif dargestellt: `getBeitrag()` und `getMaxMindestBeitrag()` werden allein auf dem `ElementarRisiko` aufgerufen. Andere Geschäftsprozesse sind dafür verantwortlich, dass in diesen Attributen der aktuell gültige Wert steht.

Es lässt sich der Zugriff auf die `Steuersatz`-Tabelle vermeiden. Diese Tabelle ist laut Abschnitt 6.1 entstanden, weil an zentraler Stelle in einem Schlüsselverzeichnis der Steuersatz der so genannten »Versicherungsteuer« gepflegt werden soll. Anstelle des ständigen Zugriffs auf dieses Schlüsselverzeichnis per Wrapper wird nun die Steuer als weiteres Attribut im Vertrag gespeichert. Der Zugriff auf das Schlüsselverzeichnis erfolgt nur noch einmal bei Anlage des Vertrags, um den aktuell gültigen Steuersatz im Vertrag zu speichern. Damit ist der Zugriff auf eine weitere Tabelle eingespart. Die Kosten für diese Einsparung werden durch einen weiteren Batchlauf erzeugt, der alle Verträge verändert, sobald sich die Versicherungsteuer ändert. Wie in Abbildung 8.25 zu sehen ist, wird `getSteuerSatz()` nur noch auf dem Vertrag aufgerufen.

8.6.3 Messergebnis

Zur Bestimmung der verbesserten processing time wird die Variante mit *kontrollierter Redundanz* mit der modifizierten EJB QL-Variante verglichen. Die Tabelle in Abbildung 8.26 stellt die Messwerte einander gegenüber. Beide Varianten bearbeiten 1000 Verträge.

An der processing time wird deutlich, dass die Verbesserung wesentlich ist. Die Variante mit kontrollierter Redundanz braucht weniger als zwei Drittel der processing time, der EJB QL-Variante.

Methoden	EQL 1000 // 120000	Redundanz 1000 // 120000
Sucher	00:05.8	00:02.2
aendereSfKlasse	00:12.6	00:23.0
historisiereVertrag	01:12.4	01:20.6
findeSFKlasse	00:14.9	00:09.4
ermittleZuAbschlagStufen	01:11.2	01:10.5
ermittleSteuersatz	00:30.0	00:00.0
findeRisikoBuendel	00:31.9	00:30.3
historisiereRisikoBuendel	02:11.6	02:09.3
findeEinzelZuAbschlaege	01:31.8	01:29.5
findeElementarRisiken	03:01.9	03:03.0
historisiereElementarRisiko	06:41.9	06:35.2
ermittleMaxMindestBeitrag	07:33.9	00:00.1
ermittleBeitrag	07:34.2	00:00.1
berechnePraemie	00:42.2	00:43.0
processing time	**35:32.4**	**19:30.2**

Abbildung 8.26: Tabelle: Kontrollierte Redundanz versus EJB QL-Variante

Anhand des Säulendiagramms in Abbildung 8.27 lässt sich deutlicher sehen, welche der eingangs diskutierten Verbesserungen die Reduzierung der processing time bewirkt haben.

Die Verbesserung des `Suchers` ist zuerst diskutiert worden. Die Tabelle mit Messwerten legt nahe, dass sich die Zeit für den `Sucher` halbiert hat (5 Sekunden gegenüber 2 Sekunden). Anhand des Säulendiagramms wird jedoch sichtbar, dass diese Verbesserung kaum ins Gewicht fällt gegenüber den anderen gemessenen Zeiten bzw. außerhalb der Messgenauigkeit liegt (vgl. Abschnitt 7.2).

Die zweite Verbesserung betrifft »ermittleSteuersatz«. Dieser Schritt konnte eliminiert werden, was zu einer Einsparung von absolut 30 Sekunden an einer processing time von ca. 35 Minuten (= 2100 Sekunden) führt. Prozentual wurde damit eine Verbesserung von ca. 1,5 % erreicht.

Die wesentliche Verbesserung wurde bei der Vermeidung des Zugriffs auf den Tarif erreicht. Die Messpunkte »ermittleMaxMindestBeitrag« und »ermittleBeitrag« machen mit zusammen etwa 15 Minuten einen Anteil von ca. 40 % der processing time der EJB QL-Variante aus.

8.6.4 Bewertung

Die Variante zeigt deutlich die Wichtigkeit von Messungen, *bevor* Änderungsaufwände investiert werden. Unsaubere Entwicklungsprozesse sind oft dadurch gekennzeichnet, dass der örtliche Guru mit seinem »Gefühl« für die Performanz einer zu entwickelnden Applikation Verbesserungen an den Stellen einbringt, an denen er ein Problem *vermutet*.

So wurde beispielsweise die Zeit für den `Sucher` halbiert und die Zeit zur Ermittlung des Steuersatzes auf 0 gesenkt. Das ist für sich betrachtet ein beachtlicher Erfolg, gemessen an der Verbesserung, die im Beispielgeschäftsprozess erreicht wurde, jedoch vergeudete

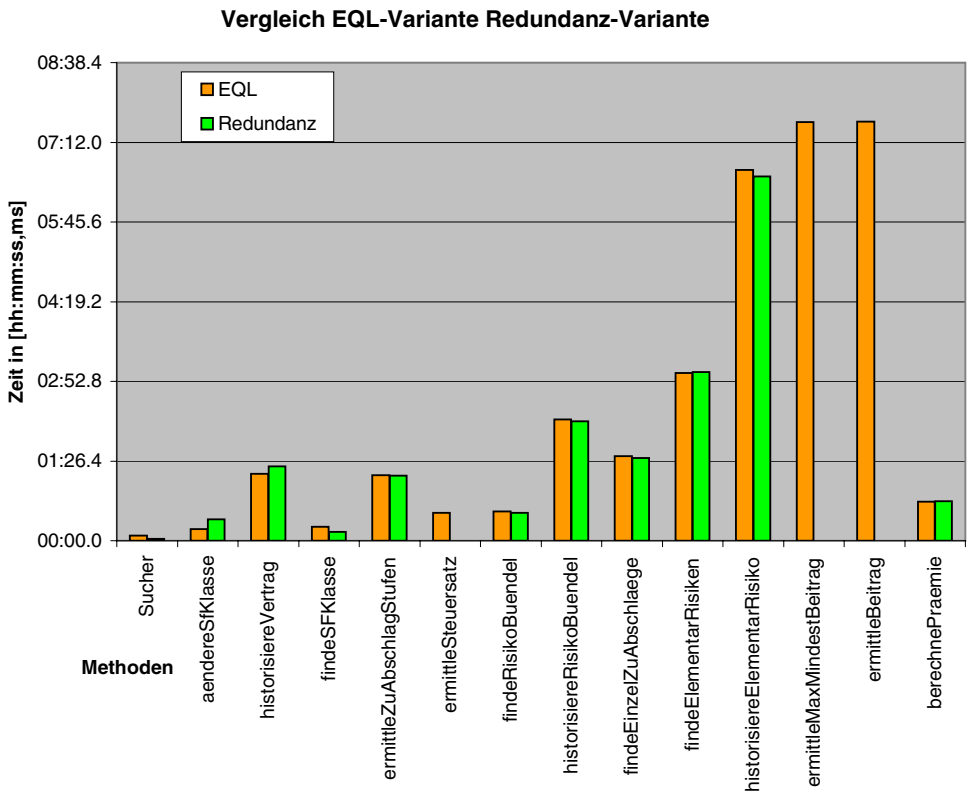


Abbildung 8.27: Säulendiagramm: Redundanz versus EJB QL

Entwicklungszeit. Mit den ersten beiden Veränderungen ist eher eine Verschlechterung des Systems eingetreten, da zusätzliche Logik notwendig ist, um die Integrität der Daten sicherzustellen, und die Wartbarkeit durch die Nutzung von Redundanzen abgenommen hat.

Der überflüssige Aufwand wäre vermieden worden, wenn anhand einer Messung die Anteile von »Sucher« und »ermittleSteuersatz« an der processing time ermittelt worden wären. Diese beiden Verbesserungen sind dennoch im Rahmen dieses Buchs durchgeführt worden, um Einsatzmöglichkeiten der Nutzung kontrollierter Redundanz aufzuzeigen.

Die Vermeidung des Zugriffs auf den Tarif ist dagegen sehr erfolgreich gewesen. Der einzelne Aufruf von Tarif dauert etwa 60 Millisekunden. Da es zwei Tarifaufrufe pro `ElementarRisiko` gibt und 8 ElementarRisiken dem Beispielvertrag assoziiert sind, summieren sich die Kosten der Aufrufe schnell zu einem großen Anteil der processing time. Daran zeigt sich die Wichtigkeit, eine realistische Workload zu benutzen. Haben die wirklichen Verträge des produktiven Systems nur zwei `ElementarRisko`-Instanzen, so würde auch dieser Änderungsaufwand fraglich.

Wie eingangs erwähnt, nimmt durch die hier beschriebene Nutzung von kontrollierter Redundanz die Performanz der Online-Geschäftsvorfälle und auch mancher Batchläufe ab. Bezogen auf die Gesamtperformanz des Systems werden die Kosten nur *umverteilt*. Ein System, das sowohl im Online- als auch im Batchverhalten zu langsam ist, kann durch kontrollierte Redundanz kaum profitieren. Die verschlechterte Wartbarkeit eines Systems ist ebenfalls kritisch bei der Nutzung kontrollierter Redundanz zu bedenken. Durch Redundanzen werden zusätzliche Abhängigkeiten im System geschaffen, die bei Änderungen zu berücksichtigen sind.

Der Änderungsaufand für die Nutzung kontrollierter Redundanz ist hoch. Neben dem Geschäftsprozess, der verbessert werden soll, müssen weitere Methoden implementiert oder angepasst werden, um die Integrität der Daten sicherzustellen. Die Integrität ist anschließend durch umfangreiche Tests nachzuweisen. Durch die Tests soll ausgeschlossen werden, dass es Geschäftsvorfälle gibt, die die Integrität der Daten verletzen.

8.6.5 Design-Regel

Bei Daten mit einer geringen, genau bestimmbaren Änderungshäufigkeit können Arbeitsschritte des Batchlaufs in anderen Geschäftsprozessen vorweggenommen werden. Durch zusätzliche, redundante Attribute werden Ergebnisse der vorweggenommenen Arbeitsschritte gespeichert.

Durch Wiederholung von Attributen einer Klasse als Kopie der Attribute einer anderen Klasse können Zugriffe auf andere Instanzen vermieden werden. Auf Ebene der Datenbanktabellen enthält eine Tabelle kopierte Spalten einer anderen Tabelle, sodass der Zugriff auf die weitere Tabelle eingespart wird. Die neu geschaffenen Abhängigkeiten sind sorgfältig zu dokumentieren.

8.7 Kombination der Varianten

In dieser abschließenden Variante werden alle zuvor dargestellten Varianten kombiniert. Der Entwurf wird wie bei der Basisvariante Schritt für Schritt erläutert. Zur Motivation der einzelnen Design-Entscheidungen wird auf die vorangegangenen Kapitel verwiesen.

Begonnen wird mit dem Entwurf der statischen Struktur, darauf aufbauend wird der dynamische Ablauf diskutiert, ähnlich wie es in der Beschreibung der Basisvariante geschehen ist.

Die Basisvariante hat eindrücklich gezeigt, wie katastrophal sich die Vergeudung von Datenbankzugriffen auf die Performanz auswirken kann. Daher ist es Ziel der kombinierten Variante, möglichst wenige Transaktionen und Datenbankverbindungen aufzubauen. Das wird durch eine Bündelung mehrerer Geschäftsobjekte zu einer Transaktion erreicht. Sämtliche Änderungen, die durch die Transaktion vorgenommen werden, werden erst am Ende der Transaktion in einem *einzigen* Datenbankzugriff übertragen. Durch Message-Driven Beans wird vermieden, dass der Applikationsserver ruht, während die Änderungen in die Datenbank gespielt werden.

Die processing time der kombinierten Variante gibt Aufschluss darüber, wie viel Performanz *insgesamt* durch die dargestellten Verbesserungen zu gewinnen ist.

8.7.1 Entwurf der persistenten Struktur

Ziel beim Entwurf der persistenten Struktur ist es, dass bei der Abarbeitung des Batchlaufs möglichst wenig Tabellen angesprochen werden. An dieser Stelle fließen die Ergebnisse aus Unterkapitel 8.6 ein.

Das folgende Klassendiagramm (siehe Abbildung 8.28) zeigt alle persistenten Klassen des Geschäftsobjekts »KFZ-Versicherung«. Um die Übersichtlichkeit zu bewahren, sind nur die Attribute der Klassen, nicht aber die Methoden, dargestellt. Die Klassen `VertragsBuendel`, `ZuAbschlagStufe`, `Selbstbehalt`, `EinzelZuAbschlag` und `Deckung` verbleiben gegenüber der Basisvariante unverändert und sind daher nur mit Namen dargestellt.

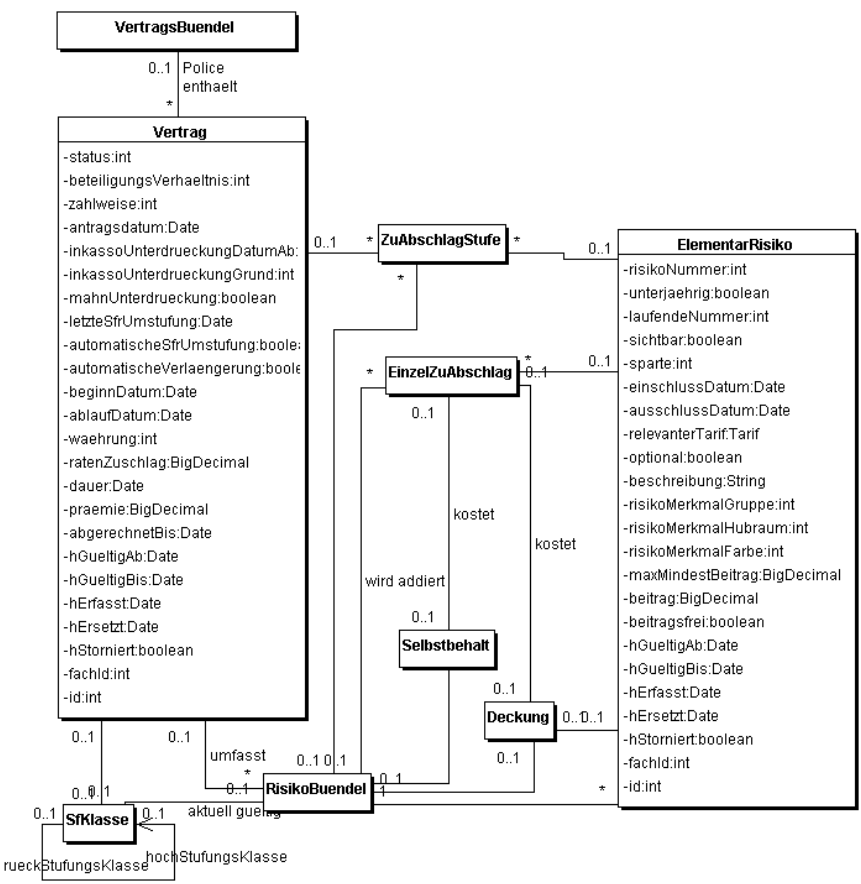


Abbildung 8.28: Klassendiagramm: Kombination aller Verbesserungen

Die Klassen `Steuersatz` und `SfKlasse` *könnten* als eigenständige Klasse entfallen. Ihre Attribute würden in `RisikoBuendel` und `Vertrag` aufgehen. Der `Vertrag` würde die beiden Attribute `prozentsatz` und `abweichungsGrund` von `Steuersatz` erhalten. Da sich im Unterkapitel 8.6 jedoch gezeigt hat, dass diese Veränderungen im Vergleich zum Aufwand wenig Performanzgewinn bewirken, wird auf sie verzichtet. Der Zugriff auf den Tarif soll jedoch vermieden werden, sodass die Attribute `ElementarRisiko.beitrag` und `ElementarRisiko.maxMindestBeitrag` anfangs mit den unmodifizierten Werten aus dem Tarif belegt werden.

Auf Ebene der Datenbank werden die Tabellen partitioniert (wie in Unterkapitel 8.3 beschrieben). Da der Sucher des Vertrags keinen grossen Einfluss auf die processing time hat, wird in Abweichung von Unterkapitel 8.3 auch der Vertrag wie alle anderen Tabellen nach dem Historisierungsattribut `hGueltigBis` partitioniert. Die Suche auf Verträgen wird dadurch beschleunigt. Es entfallen jedoch die zusätzlichen Aufwände ein weiteres, eigentlich redundantes Attribut einzupflegen.

Zusätzlich sollen alle Beans im gleichen Container eingesetzt werden, sodass die Parameter zwischen ihnen per Referenz übergeben werden können, wie in Kapitel 8.4 beschrieben. Alle Beans einer Komponete kommunizieren daher untereinander per Local Interface. Der Entwicklungsaufwand für die Local Interfaces ist eine Investition in Portierbarkeit und Modularität. Stünde der Zeitdruck bei der Entwicklung im Vordergrund, so wäre auf das proprietäre T3-Protokoll zurückgegriffen worden, um unnötige Serialisierung von Parametern zu vermeiden.

8.7.2 Entwurf der Interaktion

Der dynamische Ablauf wird im Sequenzdiagramm in Abbildung 8.29 dargestellt. Anfangs wird wie gewohnt ein `Sucher` erzeugt. Er enthält als statisches Attribut den Namen der Message-Queue. Mit diesem Namen wird über den Naming-Service die Message-Queue gefunden (JNDI-Lookup). Der `Sucher` wird bei der Erzeugung einer `Batchrahmen`-Session Bean übergeben.

Wird die Methode `start()` aufgerufen, so werden über den `Sucher` alle teilnehmenden Verträge gefunden. Die Suchanfrage ist auf Ebene der Datenbank mit einem geeigneten Index unterstützt. Die Methode `findeSfrUmstufungsTeilnehmer()` nutzt EJB QL als Anfragesprache, wie es in Unterkapitel 8.2 beschrieben ist. Dadurch wird das Laden überflüssiger Bean-Instanzen vermieden.

In Unterkapitel 8.5 wird parallel auf mehrere Message-Driven Beans verzweigt. An jede Message-Driven Bean wird eine Referenz auf einen zu bearbeitenden Vertrag übermittelt. Innerhalb der Bearbeitungschritte, die durch die Message-Driven Bean auf dem Vertrag aufgerufen werden, wird jede Attributänderung direkt mit einem Datenbankzugriff gespeichert. Diese Schritte wurden im Folgenden leicht modifiziert, um Datenbankzugriffe einzusparen.

Durch Konfiguration der Beans im Deployment Descriptor ist es möglich, *alle* Änderungen mit *einem* Datenbankzugriff zum Ende der Transaktion zu übertragen (»write

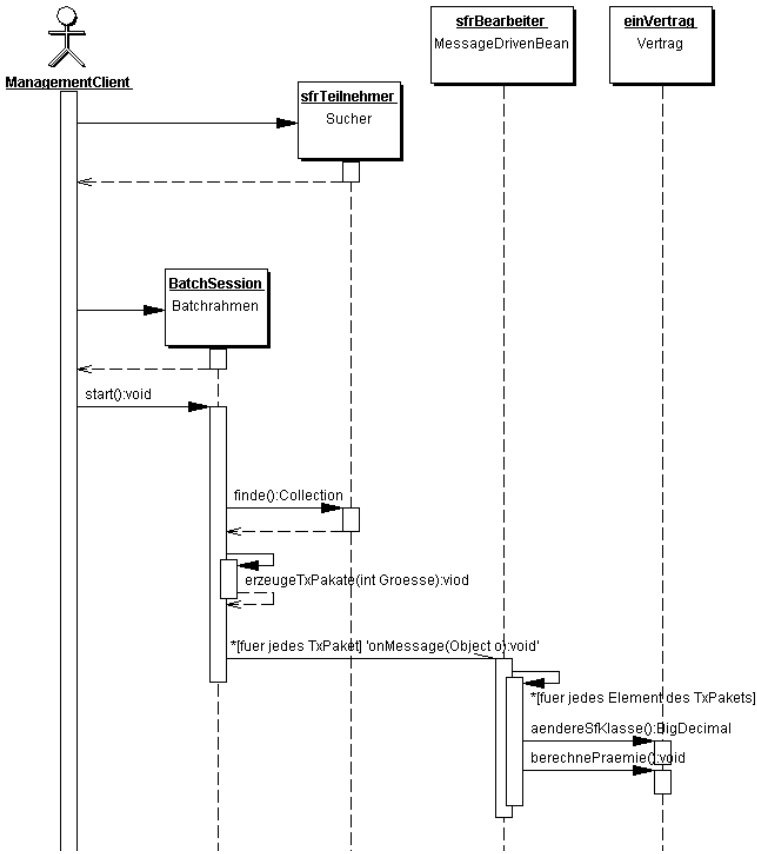


Abbildung 8.29: Sequenzdiagramm: Kombination aller Verbesserungen

on commit«). Dadurch werden mehrere Datenbankzugriffe für `UPDATE`-Operationen eingespart. Pro Geschäftsobjekt findet genau ein Zugriff zum Aktualisieren und Einfügen neuer Instanzen statt.

Dieses Verhalten wird pro Bean mit dem folgenden Eintrag im Deployment Descriptor Weblogic-ejb-jar.xml erreicht:

```
<entity-descriptor>
  ...
  <persistence>
    <delay-updates-until-end-of-tx>
        True
    </delay-updates-until-end-of-tx>
  </persistence>
  ...
</entity-descriptor>
```

Listing 8.15: Deklaration des Update-Zeitpunkts einer Transaktion

In Abänderung zu Unterkapitel 8.5 wird das Vorgehen auf mehrere Geschäftsobjekte ausgeweitet. Dazu werden aus der Menge der zu bearbeitenden Geschäftsobjekte Pakete einer definierten Größe zusammengestellt (siehe Methode `Batchrahmen.erzeugeTxPakete (int groesse)`). Alle Geschäftsobjekte eines Pakets »teilen« sich eine einzelne Transaktion. Ist die Paketgröße z.B. 100, so werden die Einfüge- und Aktualisierungsfunktionen für 100 Geschäftsobjekte mit nur einem Datenbankzugriff übermittelt.

Die Message-Driven Beans rufen pro Geschäftsobjekt in dem übergebenen Paket die Bearbeitungsfunktionalität auf (siehe `Vertrag.aendereSfKlasse()` und `Vertrag.berechne Praemie()`). Die Message-Driven Beans haben allein den Zweck, eine Verzweigung auf parallele Bearbeitungsstränge zu ermöglichen. Sie beinhalten keine Geschäftslogik. Die Geschäftslogik liegt in der `Vertrag`-Bean.

Für die Berechnung der Prämie werden die redundanten Attribute genutzt, wie in Kapitel 8.6 beschrieben. Dadurch wird der Zugriff auf das Tarifsystem vermieden und die einzelne Anfragen beschleunigt.

Der Grad der Parallelität der Verarbeitung kann von außen durch die Anzahl der Message-Driven Beans beeinflusst werden. Dieser Konfigurationsparameter wird im Deployment Descriptor gesetzt. Die Größe der Pakete, d.h. die Anzahl der Geschäftsobjekte, die zu einem Paket gehören, ist ebenfalls im Deployment Descriptor konfigurierbar. Er ist als `env–entry` zu `BatchrahmenBean` gespeichert. Der Sucher speichert als statisches Attribut den Schlüssel, mit dem die Paketgröße aus dem Deployment Descriptor ausgelesen werden kann. Auf diese Art kann für jeden Batchlauf individuell eingestellt werden, mit welchem Grad an Parallelität er arbeitet. Die Paketgröße wird von der *Robustheit* des Batchlaufs bestimmt. Scheitert die Bearbeitung des 99. Geschäftsobjekts eines Pakets, so werden auch die Änderungen an den erfolgreich zuvor bearbeiteten 98 Geschäftsobjekten rückgängig gemacht (atomare Ausführung der Transaktion). Dieser Fall lässt sich jedoch durch spezielle Fehlerbehandlungskonzepte vermeiden.

8.7.3 Messergebnis

Im Folgenden werden mehrere Messungen dargestellt, die jeweils 1000 Geschäftsobjekte bearbeitet haben. In der finalen Variante kann die Anzahl der Message-Driven Beans und die Anzahl der Geschäftsobjekte, die sich eine Transaktion teilen, variiert werden. Diese Parameter können abhängig vom Geschäftsprozess und der Hardware die Performanz verbessern oder bei falscher Konfiguration sogar verschlechtern.

Im Folgenden wird die Wahl der Konfigurationsparameter für die finale Variante erörtert. Die gemessene processing time für verschiedene Parameter ist in der Tabelle in Abbildung 8.30 dargestellt.

Die Tabelle zeigt in der ersten Zeile des grauen Bereichs den Namen der Variante, in der zweiten Zeile die Anzahl der Message-Driven Beans zur parallelen Abarbeitung und in der dritten Zeile die Anzahl der Geschäftsobjekte, die sich eine Transaktion teilen. Insgesamt sind 11 verschiedene Kombinationen von Parametern und processing times dargestellt. Als Vergleich sind die processing times der Varianten »EQL (mit Index)« und

Variantenname	EQL	Redundanz	Final	Final
Anzahl Message-Driven Beans	-	-	1	1
Geschäftsobjekte pro tx	-	-	1	100
processing time	35:32.4	19:30.2	20:02.0	14:47.0

Variantenname	Final	Final	Final	Final
Anzahl Message-Driven Beans	1	1	2	2
Geschäftsobjekte pro tx	200	500	1	100
processing time	14:26.0	13:45.0	11:43.0	10:55.0

Variantenname	Final	Final	Final	Final	Final
Anzahl Message-Driven Beans	3	3	10	10	20
Geschäftsobjekte pro tx	10	100	10	33	10
processing time	10:50.0	10:40.0	10:10.0	14:32.0	11:38.0

Abbildung 8.30: Tabelle: processing time der finalen Variante für verschiedene Parameter

»Redundanz« dargestellt. Aus Darstellungsgründen wurde die Tabelle selbst in mehrere Zeilen aufgeteilt. Beispielsweise dauert die processing time 10 Minuten und 50 Sekunden bei Verwendung von drei Message-Driven Beans und zehn Geschäftsobjekten, die innerhalb derselben Transaktion bearbeitet werden.

Wie in Abschnitt 7.2 beschrieben wurde, sind die Bestandteile der processing time nur zusätzliche Werte, um gezielte Verbesserungen vornehmen zu können. Sie wurden daher in der Tabelle ausgelassen.

Abbildung 8.31 stellt die Werte in einem Säulendiagramm dar. Die Säule am linken Rand ist als oberste in der Legende aufgeführt. Die Namenskürzel in der Legende geben die Parameter für die Anzahl der Message-Driven Beans und die Anzahl der Geschäftsobjekte einer Transaktion wieder. Am längsten von den Konfigurationsvarianten der finalen Variante dauert mit ca. 20 Minuten die processing time bei einem Message-Driven Bean (sequenzielle Verarbeitung) und nur einem Geschäftsobjekt, das pro Transaktion bearbeitet wird. Sie entspricht von der Art der Abarbeitung der Variante mit kontrollierter Redundanz. Die zusätzliche Zeit gegenüber der Variante mit Redundanz ergibt sich wahrscheinlich durch das Serialisieren und Deserialisieren der `Messages`. Werden 100 Geschäftsobjekte innerhalb derselben Transaktion bearbeitet, sinkt die processing time auf etwa 15 Minuten. Weitere Erhöhungen der Geschäftsobjekte pro Transaktion bringen keinen wesentlichen Fortschritt.

In Abbildung 8.32 ist eine typische Lastkurve für die Variante mit einer Message-Driven Bean und 100 Geschäftsobjekten pro Transaktion dargestellt. Deutlich sind zwei tiefe Einbrüche in der Prozessorlast zu sehen. In dieser Zeit werden die Dateien der Datenbank auf der Festplatte aktualisiert. Es liegt nahe, mit weiteren parallelen Verarbeitungssträngen die Leerlaufzeit des Prozessors zu nutzen. Wie an dem Säulendiagramm zu sehen ist, werden die Ressourcen für die Variante mit 10 Message-Driven Beans und 10 Geschäftsobjekten am besten genutzt: processing time ca. 10 Minuten. Eine Erhöhung der Anzahl der Message-Driven Beans oder der Geschäftsobjekte pro Transaktion bringt Verschlechterung. So dauert die Nutzung von 20 Message-Driven Beans etwa 12 Minuten. Werden die Geschäftsobjekte pro Transaktion auf 33 erhöht, dauert die Abarbeitung etwa 15 Minuten.

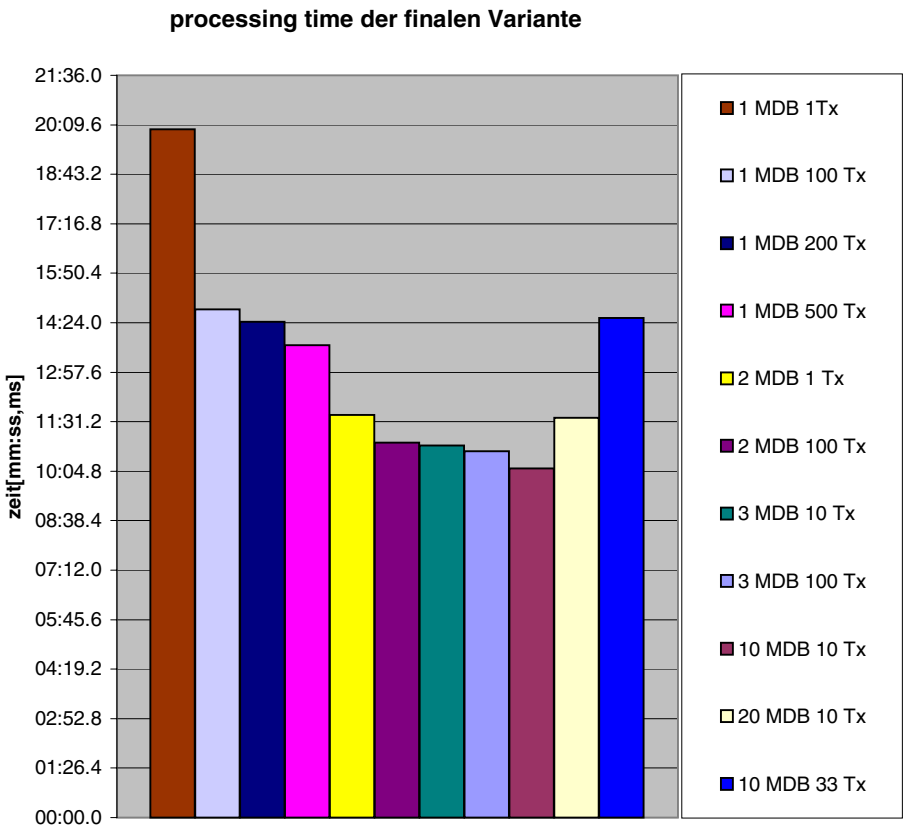


Abbildung 8.31: Tabelle: processing time der finalen Variante für verschiedene Parameter

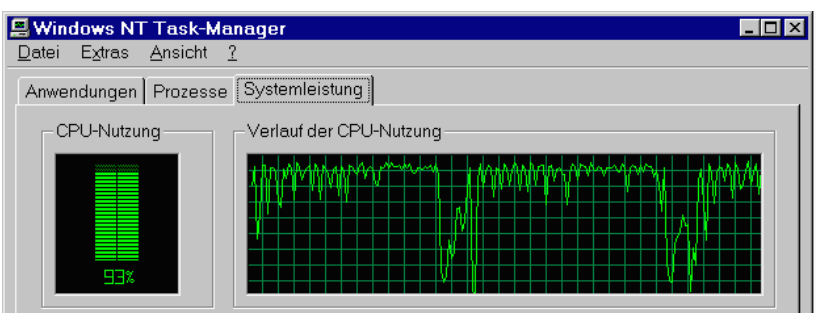


Abbildung 8.32: Lastkurve bei 1 Message-Driven Bean und 100 Geschäftsobjekten

8.7.4 Bewertung

Im Vergleich zu den vorangegangen Varianten hat die finale Variante eine deutlich geringere processing time. Die Basisvariante lässt es kaum zu, 1000 Geschäftsobjekte zu bearbeiten. Allein für 10 Geschäftsobjekte dauert die processing time fast einen Tag. Die mit Indexen beschleunigte EJB QL-Variante benötigte nur noch 35 Minuten. Durch die Kombination der Varianten, mit dem Ziel möglichst wenige Datenbankzugriffe durchzuführen, ist es gelungen, die processing time auf etwa 10 Minuten zu senken.

Wesentlich zu diesem Ergebnis hat die Möglichkeit des Applikationsservers beigetragen, die Datenbank durch nur einen Zugriff am Ende der Transaktion zu aktualisieren. Da die EJB 2.0-Spezifikation keine Aktualisierungsintervalle vorschreibt, verstößt diese Verbesserung nicht gegen den Standard. Gleichwohl wird diese Möglichkeit nicht vom Standard erzwungen, sodass nicht alle Applikationsserver sie bieten. Eine Anwendung, die für verzögertes Aktualisieren programmiert worden ist, ist weiterhin auf andere Applikationsserver portierbar.

Die dargestellten Parameterkombinationen sind stichprobenartig gewählt worden, um ihre Wirkung auf die processing time zu verdeutlichen. Es ist denkbar, dass bei einer systematischen Messung aller Parameterkombinationen eine Zeit unter 10 Minuten erreichbar ist. Die ermittelten Zeiten und Parameterkombinationen sind nicht übertragbar auf eine andere Hardware-Umgebeung. Allein die Verteilung von Applikationsserver und Datenbank auf verschiedene Rechner würde deutlich andere Lastkurven erzeugen. Durch den Einsatz weiterer Prozessoren sollte sich auch die Wirkung der Parallelisierung verbessern.

8.7.5 Design-Regel

Die Design-Regel zur finalen Variante ergibt sich aus den Design-Regeln vorangegangener Varianten. Ein allgemeines Design für Massendatenverarbeitung mit EJB 2.0 enthält Kapitel 9. Es ist unabhängig vom Beispielgeschäftsprozess und beinhaltet die Erkenntnisse aller Varianten dieses Buchs. Dieses Design ist das Ergebnis des Buchs und bildet daher ein eigenes ausführliches Kapitel.

9 Design-Regeln für performante EJB-Anwendungen

Das folgende Kapitel fasst die Ergebnisse des Buchs zusammen. Ziel des Buchs ist die Ermittlung eines performanten Designs für Geschäftsprozesse mit EJB 2.0. Dazu sind durch Prototypenerstellung verschiedene Designentscheidungen bezüglich ihres Einflusses auf die Performanz bewertet worden.

Als Ergebnis wird ein Entwurf vorgestellt, der performante Umsetzung von Geschäftsprozessen auf Basis von EJB 2.0 ermöglicht. Dieser Entwurf beinhaltet die Performanzverbesserungen der verschiedenen Prototypen (Implementierungsvarianten) und genügt den in der Einleitung zu Kapitel 7 gestellten Anforderungen:

1. Verwässerung des EJB 2.0-Standards vermeiden
2. Wiederverwendbarkeit und Modularität
3. Verallgemeinerbarkeit des Batchkonzepts

Ansatz für performante Geschäftsprozesse ist es, parallele Verarbeitung einzusetzen und mehrere Geschäftsobjekte in einer einzelnen Transaktion zu bündeln. Datenbanknahe Massendatenverarbeitung erreicht ihre hohe Geschwindigkeit, indem *Mengen* von Instanzen aufeinander angewendet werden. Da Enterprise JavaBeans den Ansatz der objektorientierten Kapselung verfolgen, können sie nur *Instanz pro Instanz* bearbeitet werden. Andernfalls wird die objektorientierte Kapselung aufgebrochen. Diese durch das Programmierparadigma erzwungene Geschwindigkeitseinbuße wird durch Parallelisierung abgefangen. Der dargestellte Parallelisierungsansatz ist nur auf EJB 2.0-Plattformen möglich.

9.1 Statisches Modell

Bei dem *Entwurf der statischen Struktur* wird nach der objektorientierten Analyse ein Komponentenschnitt durchgeführt (vgl. Unterkapitel 8.4.2). Mehrere Beans bilden eine Komponente. Die Kommunikation innerhalb der Komponente wird per Local Interfaces abgewickelt. Die Kommunikation *zwischen* Komponenten erfolgt per Remote Interfaces. Abbildung 9.1 zeigt diesen Zusammenhang schematisch. Dargestellt sind zwei Komponenten `Komponente1` und `Komponente2`. `Komponente1` enthält `BeanA`, `BeanB` und `BeanC`. `BeanC` verfügt nur über Local Interfaces und ist daher nur innerhalb der Komponente ansprechbar, beispielsweise von `BeanB`. `BeanB` verfügt über Local und Remote Interfaces, sodass ein gewisser Teil von `BeanBs` Funktionalität nur innerhalb der Komponente angesprochen werden kann. Ein anderer Teil der Funktionalität ist auch von außen ansprechbar. In den Local und Remote Interfaces sind also nicht zwangsläufig die gleichen Funktionen ansprechbar. `BeanA` besitzt nur Remote Interfaces, sodass sie nur von außen angesprochen werden kann. Sie kann beispielsweise an `BeanB` delegieren, um Funktionalität nach außen anzubieten.

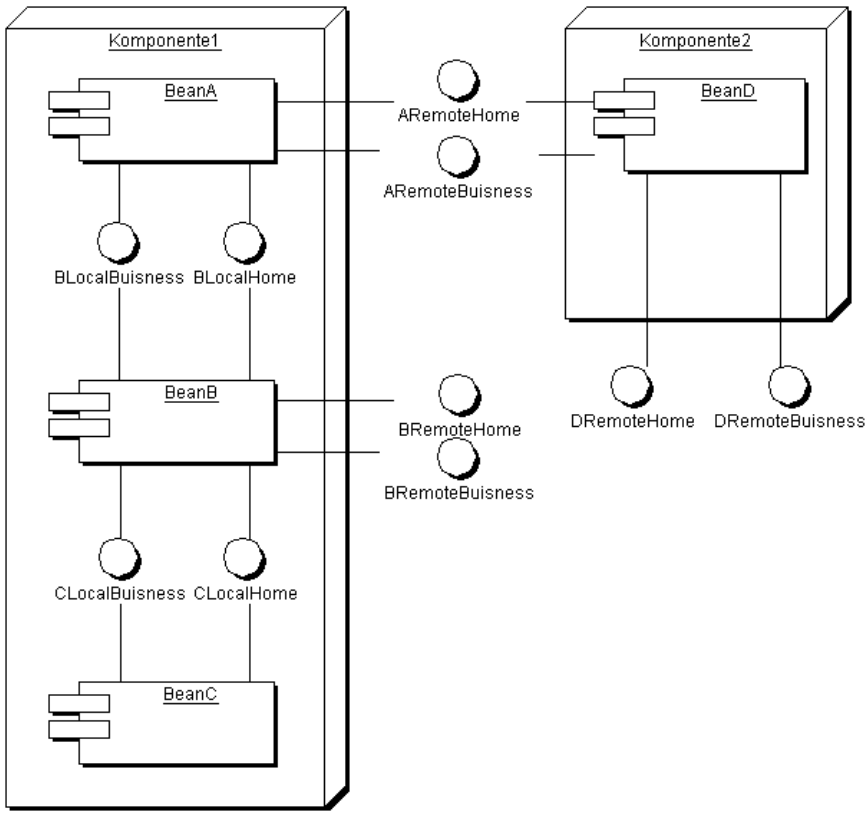


Abbildung 9.1: Komponentenschnitt und Schnittstellen

Durch diesen Ansatz wird eine bessere Kaspelung erreicht als bei Komponenten, die nach EJB 1.0 oder 1.1 implementiert worden sind. Es kann explizit unterschieden werden, welche Funktionalität nur innerhalb der Komponente zur Verfügung steht. Ändern sich die Local Interfaces, so reicht es, die Komponente allein neu zu übersetzen. Aufrufende Komponenten müssen nicht nochmals übersetzt werden, da sie auf die unveränderten Remote Interfaces zugreifen. Um parallele Entwicklung mehrerer Komponenten zu ermöglichen, genügt es, fremden Komponenten die eigenen Remote Interfaces zur Verfügung zu stellen. Davon unberührt kann die interne Struktur der Komponente und die interne Interaktion über Local Interfaces nachträglich verändert werden.

Neben der in den vorangegangenen Kapiteln nachgewiesenen Performanzverbesserung wird durch diesen Design-Ansatz die Modularität der Komponenten verbessert.

Auf Ebene der Datenbank sind die Anfragen durch geeignete Indexe zu beschleunigen. Dazu fertigt der Applikationsserver oder die Datenbank ein Protokoll der Anfragen an. Anhand dieses Protokolls können häufig gestellte oder lange dauernde Anfragen ermittelt werden. Da diese Technik weit verbreitet ist, bedarf sie keiner weiteren Erklärung.

Auf der Ebene der Datenbank können gewisse Datensätze als mögliche Ergebnisse einer Suchanfrage a priori durch Partitionierung ausgeschlossen werden. Partitionierung wird ohnehin eingesetzt, um die Wartbarkeit einer Datenbank zu erhöhen, da umfangreiche Datentabellen so in einzelne Wartungseinheiten zerlegt werden können. Wird als Partitionierungskriterium ein *fachlich bedeutsamer* Schlüssel gewählt und dieser Schlüssel bei kritischen Suchanfragen mitbenutzt, beschränkt die Datenbank die Suchanfrage nur auf einzelne Partitionen. Es muss also nicht zwangsläufig der ganze Datenbestand stets aufs Neue untersucht werden.

Während der Entwurfsphase müssen bereits performanzkritische Anwendungsfälle identifiziert werden. Ist der Entwurf zu diesen Anwendungsfällen bis auf Sequenzdiagramme verfeinert worden, so kann analysiert werden, auf welche Instanzen zugegriffen werden muss. Nehmen viele verschiedene Entity Bean-Typen an dem Anwendungsfall teil, so bedeutet das, dass viele Tabellen angesprochen werden müssen. Die Anzahl der Zugriffe auf andere Tabellen kann durch kontrollierte Redundanz vermindert werden. Eine Bean erhält dann zusätzliche Attribute, die sich aus den Attributen anderer Beans ableiten lassen, sodass sich der Zugriff auf die anderen Beans einsparen lässt. Um den Aufwand der Denormalisierung zum Performanzgewinn in Relation setzen zu können, ist es zwingend notwendig, mit ersten Prototypen des *normalisierten* Entwurfs Zeitmessungen durchzuführen. Durch die Denormalisierung verschlechtert sich meist die Wartbarkeit des Systems und die Performanz anderer Anwendungsfälle, sodass Nutzen und Nachteil detailliert ermittelt und verglichen werden müssen.

9.2 Interaktion

Ziel beim *Entwurf der Interaktion* ist es, so wenig wie möglich Datenbankzugriffe zu erzeugen, ohne die Integrität der Daten zu verletzen. Die Möglichkeiten zur Performanzsteigerung der Interaktion zwischen Beans ergeben sich daraus wie folgt.

Wird eine Menge bestimmter Bean-Instanzen gesucht, so darf dazu nicht, wie in Java-Programmen üblich, über verschiedene Relationen iteriert werden. Das Iterieren über Relationen führt zwangsläufig dazu, dass alle an der Relation teilnehmenden Beans geladen werden. Wenn anschließend alle geladenen Beans benötigt werden, ist das ein korrektes Vorgehen. Wird allerdings nur eine kleine Teilmenge der Beans benötigt, vergeudet dieses Vorgehen teure Datenbankzugriffe. Die meisten Beans werden in diesem Fall geladen, um festzustellen, dass sie gewisse Suchkriterien *nicht* erfüllen. Dieser Fehler tritt ebenfalls auf, wenn über mehrere Relationen traversiert wird. Werden beispielsweise zu einem `Vertrag` alle beitragsfreien `ElementarRisiko`-Instanzen gesucht (vgl. Abbildung 6.1), so ist zu vermeiden, dass durch das Traversieren der Relationen auch alle `RisikoBuendel` geladen werden. Wird dieses Traversieren von Relationen zur Umsetzung einer Suchanfrage konsequent benutzt, ergibt sich eine *um mehrere Größenordnungen verringerte* Performanz.

Durch Nutzung der Enterprise JavaBean Query Language kann dieser Fehler vermieden werden, ohne die Schichtentrennung zur Persistenzschicht zu zerstören. Prinzipiell werden nur die Beans in den Applikationsserver geladen, die wirklich bearbeitet werden.

Weiterhin kann das Laden ganzer Beans vermieden werden, wenn nur ein einzelnes Attribut benötigt wird. Der Rückgabewert einer EJB QL-Anfrage kann auch eine Menge einzelner Attribute sein (vgl. Abschnitt 4.4.2). Auch hier wird das Laden der *ganzen* Bean vermieden.

Das Aktualisieren der Datenbank sollte nicht pro verändertem Attribut oder pro neu angelegter Instanz vorgenommen werden. Soweit der Applikationsserver diese Möglichkeit unterstützt, sollten die Änderungen an persistenten Daten erst im Applikationsserver gesammelt und dann in *einem* Datenbankzugriff am Ende der Transaktion eingespielt werden (write on commit). Dieses Verhalten ist kein Teil der Spezifikation, sodass es nicht von allen Applikationsservern unterstützt wird. Die Art der Aktualisierung des persistenten Speichers wird von der Spezifikation offen gelassen, sodass eine Verwendung dieser Möglichkeit auch nicht gegen die Spezifikation verstößt. Das Nutzen dieser Möglichkeit setzt voraus, dass die Anwendung entsprechend implementiert wurde. Die finale Variante liefert ein Beispiel dafür.

Speziell bei Massendatenverarbeitungsprozessen bietet es sich an, die Bearbeitung mehrerer Geschäftsobjekte zu einer einzelnen Transaktion zu bündeln. Aber auch Online-Anwendungen können von dieser Möglichkeit, die Performanz zu verbessern, profitieren.

Neben den Transaktionsgrenzen muss der Isolierungsgrad der Transaktionen bedacht werden, um einen möglichst hohen Durchsatz erreichen zu können. Auf Daten, bei denen eine Änderung organisatorisch ausgeschlossen werden kann, kann der Isolierungsgrad soweit reduziert werden, dass mehrere Transaktionen sie gleichzeitig lesen können. Beispielsweise ist es gesetztlich verboten, die Rabatte für einen abgeschlossenen Vertrag nachträglich zu ändern. Damit ist organisatorisch ausgeschlossen, dass sich die Werte einer `SfKlasse` während des Batchlaufs ändern können. Sie kann daher von mehreren Transaktionen gleichzeit benutzt werden. Sind exklusive Sperren für die Daten notwendig, so ist darauf zu achten, dass sie möglichst nur für kurze Zeit gehalten werden.

Datenbanken sind für eine hohe Anzahl gleichzeitiger Anfragen ausgelegt. Durch Message-Driven Beans wird innerhalb eines Geschäftsprozesses auf dem Applikationsserver parallel verzweigt. Massendatenverarbeitung besteht damit aus einer möglichst hohen Anzahl paralleler Bearbeitungen von einzelnen Geschäftsobjekten. Für das einzelne Geschäftsobjekt bleibt die objektorientierte Kapselung erhalten. Die fachliche Logik der Bearbeitung ist in Methoden der EnitityBeans implementiert und die Message-Driven Beans übernehmen allein die parallele Verzweigung.

Den Message-Driven Beans müssen nicht zwangsläufig einzelne Geschäftsobjekte übergeben werden. Wird den Beans eine Menge von Geschäftsobjekten übergeben, so werden alle Geschäftsobjekte innerhalb des Transaktionskontextes der Methode `MessageDrivenBean.onMessage` bearbeitet. Erst zum Ende dieser einzelnen Transaktion werden alle veränderten Geschäftsobjekte in der Datenbank aktualisiert, um Datenbankzugriffe zu sparen (Bündelung von Geschäftsobjekten).

Besitzt die Betriebsabteilung für die Batchanwendung genug Spielraum und Flexibilität, so kann ein einzelner Batchlauf in mehrere kleine Batchläufe mit weniger Geschäfts-

objekten heruntergebrochen werden. Dadurch wird auf organisatorische Weise das Zeitlimit für den Batchlauf umgangen.

9.3 Einhaltung der Randbedingungen

Nachdem das Design für performante Massendatenverarbeitung mit EJB 2.0 vorgestellt worden ist, bleibt zu überprüfen, ob die anfangs gestellten Forderungen eingelöst werden.

Alle vorgestellten Verbesserungen beziehen sich auf Möglichkeiten des Enterprise JavaBean 2.0-Standards oder übertragen bekannte Verbesserungskonzepte aus der Anwendungsentwicklung für Datenbanken. Es sind keine eigenen Klassen entwickelt worden, um die Kommunikation zwischen Beans zu beschleunigen oder Beans in einem Pufferspeicher vorzuhalten. Das vorgestellte Design ist also ohne zusätzliche Infrastruktur-Klassen auf jedem EJB 2.0-zertifizierten Applikationsserver einsetzbar. Die Verwässerung des Standards wurde vermieden.

Die eingeforderte Wiederverwendbarkeit bezieht sich auf die Funktionalität der Batchverarbeitung. Da die implementierte Funktionalität sich in den Methoden der Beans befindet, ist sie auch durch Online-Anwendungen als Aufruf auf einer einzelnen Instanz nutzbar. Die Modularität der Anwendung ist gegenüber einer Applikation auf Basis von EJB 1.1 erhöht worden. Durch Local Interfaces kann klar unterschieden werden zwischen Funktionalität, die über die Komponentenschnittstelle von außen ansprechbar ist, und Funktionalität, die nur innerhalb der Komponente benutzt werden soll. Das verbessert die Austauschbarkeit von Komponenten und ermöglicht die parallele Entwicklung von Komponenten.

Die vorgestellten Verbesserungen sind nicht von der Anwendungsdomäne abhängig und auch nicht vom beispielhaft beschriebenen Geschäftsprozess. Sie lassen sich daher auch auf andere Massendatenverarbeitungsprozesse übertragen. Die gestellten Randbedingungen sind damit eingehalten worden.

10 Ausblick

Mit den diskutierten Design-Regeln zur Entwicklung von performanten Batches in EJB-basierten Anwendungen haben wir gezeigt, dass EJB 2.0 eine Grundlage auch für Anwendungen sein kann, in denen große Datenbestände bearbeitet werden. Die Auswirkungen der Beachtung dieser Regeln wurden explizit quantifiziert, Verbesserungen wurden messbar gemacht. Dennoch bleibt eine gewisse Skepsis: bedeuten die gemessenen Verbesserungen denn nun tatsächlich, dass performante EJB 2.0-Anwendungen möglich sind? Oder zeigen die gemessenen Verbesserungen womöglich nur, dass man von katastrophal langsamen Anwendungen zu schnelleren, aber immer noch inakzeptabel langsamen Anwendungen kommen kann?

Unsere Antwort ist eindeutig: *Performante Anwendungen auf der Basis von EJB sind möglich.* Diese sehr abstrakte Aussage wird allerdings erst richtig nützlich, wenn wir im Auge behalten, unter welchen Umständen aus der Performanzmöglichkeit auch Realität wird. Um dies zu beurteilen, wagen wir einen Blick auf eine Reihe von Projekten, in denen wir an der Entwicklung performanter, EJB-basierter Anwendungen beteiligt waren:

- Es wurde ein im Internet angebotenes Reservierungssystem für ein international tätiges Schulungsunternehmen realisiert. Die kritischen Batchläufe waren Provisionsabrechnungen und Monatsende-Abrechnungen. Beide wurden durch den Einsatz dedizierter Batch-Zugriffsfunktionen schnell genug.
- Es wurde eine online-Banking und Brokerage-Lösung für den Direktableger einer großen Bank gebaut. Es wurden knapp eine Million Kundenkonten verwaltet. Die Ermittlung von Statistiken über alle Konten wurden in Form von Batches durchgeführt. Initiale Performanzprobleme konnten durch parallele Bearbeitung in mehreren Threads gelöst werden.
- Es wurde ein Industriekunden-Bestandsführungssystem für eine Versicherung gebaut. Durch Analyse der log-Dateien, Anpassen der Algorithmen zur Verminderung der Datenbankanfragen und der Nutzung mehrerer Threads wurden sowohl die Batchläufe als auch die Dialogteile schnell genug.

In zwei der genannten Projekte wurden die in diesem Buch vorgestellten Design-Regeln nicht von Anfang an berücksichtigt, sodass zahlreiche nachträgliche Performanzverbesserungen vorgenommen werden mussten. Nachträgliche Verbesserungen haben sich dabei als noch mühseliger, risikoreicher und teurer herausgestellt als ohnehin schon vermutet. Die konsequente Anwendung der erörterten Design-Regeln von Projektbeginn an ist daher von grundlegender Bedeutung für den Projekterfolg.

In den angesprochenen Entwicklungsprojekten hat sich auch gezeigt, dass die Performanz von Batches nur eines von zwei drängenden Performanzproblemen ist. Das andere ist die Performanz von Dialoganwendungen. In den genannten Projekten wurde deutlich, dass es ohne den performanten Zugriff auf große Datenmengen kaum möglich ist, performante Dialoganwendungen zu erstellen. Die Beachtung der Design-Regeln zur

Verbesserung der Performanz der Batchanteile ist damit eine notwendige, nicht jedoch hinreichende Bedingung für performante Dialoganteile.

Dialoganwendungen (bzw. Online-Anwendungen) profitieren von den vorgestellten Design-Regeln, da diese Regeln die Performanz der Geschäftslogik und Persistenzschicht einer Applikationsserver-Anwendung verbessern. Abbildung 3.5 in Abschnitt 3.2 zeigt die Anwendungsschichtung einer typischen applikationsserver-basierten Anwendung. An dieser Abbildung wird deutlich, dass die Performanz einer interaktiven Anwendung sich im Zusammenspiel von drei Schichten ergibt:

1. Laufzeitverhalten des Client (Java-Anwendung)
2. Übertragung zwischen Client und Applikationsserver (schwarze Pfeile)
3. Geschäftslogik und Persistenz(Applikationsserver mit Geschäftsobjekten)

Die ersten beiden Schichten wurden in diesem Buch explizit nicht betrachtet. Sie enthalten eigene, sehr spezifische Problemstellungen, wie sich in der umfangreichen Literatur zur Performanz dieser Schichten zeigt. Die Schicht der Geschäftslogik muss die beiden zuvor beschriebenen Schichten bedienen. In ihr greifen — bei interaktiven Anwendungen mit etwas veränderter Gewichtung — die Design-Regeln, die in diesem Buch ermittelt worden sind. Die Bündelung von mehreren Geschäftsobjekten in einer Transaktion wird geringere Bedeutung haben. Dies ist durch die Verarbeitungslogik interaktiver Anwendungen bedingt. Hingegen sind die Verbesserungen der Anfragen durch die EJB-QL und die Analyse der Serverlogs, die Verminderung der Serialisierungskosten durch Local Interfaces, die Verminderung der Datenbankzugriffe durch write-on-commit und die Parallelisierung bzw. asynchrone Verarbeitung von Arbeitsschritten mit Message-Driven Beans wesentliche Möglichkeiten, um Client-Anfragen performant bearbeiten zu können. In dieser Hinsicht sind die erarbeiteten Design-Regeln auch grundlegend für performante, interaktive Anwendungen.

Eine offene Frage bleibt die nach der Verallgemeinerbarkeit der vorgestellten Regeln und Ergebnisse. Unsere Validierungen in der Praxis beschränken sich auf klassische Informationssysteme. Ein typisches Merkmal dieser Systeme ist, dass die arithmetische Komplexität selten über die der Zinsrechnung hinausgeht. Im Wesentlichen geht es um Auskunftsfähigkeit (welcher Kunde hat welche Verträge und wie oft gab es Kontakte mit ihm und Ähnliches) und um einfache Werteänderungen (Adressänderungen, Tarifänderungen, usw.). Komplexität in der Anwendungslogik entsteht über Plausibilitäten, die sich über viele Objekte erstrecken, über Anfragen, die auf (viele) Objekte (vieler) Typen zugreifen, über die Menge der zu verwaltenden Daten und über komplizierte Anforderungen an die Darstellung von Objekten auf Bildschirmen und Schriftstücken. Hierbei geht es ohne weiteres um Hunderttausende (manchmal auch Millionen) von Objekten, die in verschiedenen Historienständen verwaltet werden müssen. Wenn es für eine solche Zahl von Objekten dann auch nur um vergleichsweise einfache Operationen geht (Objekt anschauen, ggf. einen Wert ändern und eine neue Rechnung stellen), dann hat man es im Wesentlichen nicht mit einem funktionalen Problem zu tun, sondern damit, einfache Algorithmen so performant zu gestalten, dass sie in Form eines Batches im dafür vorgesehenen Zeitfenster durchgeführt werden können. Dieses Problem lässt sich

mit EJB 2.0 und den diskutierten Regeln bewältigen. Ob diese Regeln ausreichen, um andere Arten von Anwendungen zu entwickeln, lässt sich auf der Grundlage der durchgeführten Experimente nicht beantworten. Anforderungen, die an sicherheitskritische oder Realzeit-Anwendungen gestellt werden, scheinen auf den ersten Blick über das Anforderungsniveau für Informationssysteme hinauszugehen. Ein offenes Ende bleibt der Entwurf von Anwendungen, die ohne programmiertechnische Anpassungen für parallele Rechensysteme skalieren. Die Untersuchungen bezüglich des Einsatzes mehrerer, parallel zu betreibender Message-Driven Beans vermittelt erste Eindrücke, die jedoch durch den systematischen Einsatz auf Multi-Prozessorsystemen erhärtet werden müssen.

Das letztendliche Ergebnis dieses Buches geht nicht nur aus den dokumentierten Verbesserungspotenzialen hervor. Es besteht vielmehr darin, dass es in der industriellen Praxis gelingen kann, performante EJB 2.0-Anwendungen zu bauen, wenn es gelingt, die genannten Design-Regeln durchgängig anzuwenden. Dass dies gelingen kann, soll aber nicht heißen, dass es einfach ist. Die konsequente Anwendung der genannten Regeln kann nur gelingen, wenn die Maßnahmen und Aktivitäten zur Prüfung der potenziellen Performanz in alle Prozesse der Softwareentwicklung integriert sind.

Prozesse des Projektmanagements müssen darauf ausgerichtet sein, Budgets und vor allem Fristen so zu kalkulieren, dass auch die Aktivitäten des Software Performance Engineering durchgeführt werden können. Das Qualitätsmanagement muss konstruktiv und analytisch darauf hinwirken, dass alle Artefakte auch unter Performanzaspekten erstellt und geprüft werden. Typischerweise kommt hierbei insbesondere den konstruktiven Anteilen eine besondere Bedeutung zu, denn oft muss davon ausgegangen werden, dass die Entwickler weder mit den Notwendigkeiten eines begleitenden Performance Engineering vertraut sind noch die konkreten Design-Regeln für EJB-basierte Anwendungen kennen. Die Prozesse der eigentlichen Softwareentwicklung selbst sind von den in diesem Buch entwickelten Design-Regeln besonders betroffen. In genau diesen Prozessen müssen sie zur Anwendung gebracht werden, und zwar nicht als lästiges Anhängsel, sondern als integraler Bestandteil. Schließlich sind auch die Prozesse des Konfigurationsmanagements betroffen, denn Konfigurationseinheiten können nur dann zu Versionen und Releases zusammengebunden werden, wenn die Performanzanforderungen erfüllt sind. Was aber passiert, wenn dies nicht der Fall ist, wenn das neue Release aber nötig ist, um produktions- oder testverhindernde Fehler zu beseitigen? Auch aus dieser Richtung entsteht folglich die Notwendigkeit zu kleinteiligem Konfigurationsmanagement und flexibel integrierbaren Konfigurationseinheiten.

Zusammenfassend lässt sich die Botschaft dieses Buches damit folgendermaßen bewerten: traditionelle Informationssysteme lassen sich auf der Grundlage von Enterprise JavaBeans so entwickeln, dass sie den typischen Performanzanforderungen genügen. Da bei der Entwicklung übliche Fehlerquellen durch den Einsatz der Services von EJB-Applikationsservern verringert werden können, kann die Entwicklung insgesamt sehr effizient gestaltet werden. Mit anderen Worten: Enterprise JavaBeans sind eine geeignete, mittlerweile hinreichend reife Technologie für die effiziente Entwicklung performanter Anwendungen.

A Technische Plattform

Folgende Hard- und Software wurde bei der Umsetzung der Implementierungsvarianten eingesetzt.

Hardware:

- Einzelprozessor IBM-PC, Pentium III, 550 MHz
- 512 MB Arbeitsspeicher
- 1 SCSI-Festplatte

Software:

- TogetherJ 5.0 (TogetherSoft)
- Powerdesigner 6.1 (Sybase)
- T.O.A.D. VI (Quest Software)
- Weblogic Server 6.1 (BEA)
- Oracle 8.1.7 (Oracle)
- UltraEdit 8.2 (IDM Computer Solutions)
- Windows NT 4, Service Pack 6 (Microsoft)
- LaTeX

B Ausführen des Beispielcodes

Auf der beiliegenden CD-ROM befinden sich der Quelltext und die lauffähigen Prototypen zu den im Buch dargestellten Implementierungsvarianten. Um die Implementierungsvarianten starten zu können, ist eine Installation des Applikationsservers BEA WebLogic 6.1 und ein Zugriff auf eine Oracle-Datenbank (Oracle 8.1.7) mit dem Bezeichner »Batch« erforderlich.

Zuerst ist im Datenbanksystem ein Nutzer mit Administratorrechten einzurichten. Er wird benutzt, um das Datenbankschema für die verschiedenen Varianten anzulegen, die Testdaten einzuspielen und den Connection-Pool des Applikationsservers zu erstellen. Der Nutzer heißt »batchuser« und hat das Passwort »batchuser«. Im Unterverzeichnis Implementierung befindet sich das Skript createBatchuser.cmd, mit dem ein solcher Nutzer angelegt werden kann. Das Skript TestDBConnection.cmd im gleichen Unterverzeichnis kann genutzt werden, um die Datenbankverbindung für diesen Nutzer zu testen. Dieses Skript nutzt intern das mit dem Applikationsserver ausgelieferte Utility `dbping`. Eine Referenz zu diesem Utility befindet sich auf den Web-Seiten der Firma BEA.

Ist durch die vorangegangenen Schritte sichergestellt, dass eine Verbindung zur Datenbank aufgebaut werden kann, so wird als nächster Schritt eine neue *Domain* auf dem Applikationsserver eingerichtet. Sie bekommt den Bezeichner »Batch« und nutzt einen Connection-Pool auf der oben genannten Datenbank mit dem Nutzer/Passwort: `batchuser/batchuser`. Die Einrichtung einer *Domain* und ihrer Datenbankverbindung kann ebenfalls auf den Web-Seiten des Applikationsserver-Herstellers nachgelesen werden. Das Unterverzeichnis ServerKonfiguration enthält als Hilfestellung die Domain im zip-Format, mit der die Implementierungsvarianten auf dem Testrechner betrieben worden sind. Wenn das Archiv Batch.zip auf der Festplatte entpackt ist, können die darin enthaltenen Konfigurationsdateien zum Abgleich der erstellten Konfiguration der Domain benutzt werden. Wichtig sind vor allem folgende Dateien:

- **setEnv.cmd:** Diese Datei setzt die zum Server-Start notwendigen Umgebungsparameter: `Path`, `Classpath`, Stammverzeichnis der Serverinstallation `WL_HOME` und Verzeichnis der Java-Installation `JAVA_HOME`.
- **startWebLogic.cmd:** Diese Batch-Datei dient dem Starten des Applikationsservers. In ihr werden die Parameter für die Laufzeitumgebung, in der der Server arbeitet, eingestellt: u.a. Heapsize, Art der Garbage-Collection. Außerdem kann hier das Passwort eingetragen werden, mit dem der Server gestartet wird, sodass es nicht stets aufs Neue eingegeben werden muss.
- **config.xml:** Diese Konfigurationsdatei enthält Information über alle Anwendungen, die in dieser Domain zum Einsatz kommen, die Verbindungsinformation für die Datenbank, Einstellung des Logging und die Konfiguration des integrierten JMS-Servers. Es ist empfehlenswert, diese Datei über die mit dem Applikationsserver ausgelieferte Management-Console zu bearbeiten.

Diese Dateien gehören zum Lieferumfang des Applikationsservers und sind allein bezüglich der Parameter angepasst worden. Eine weitere, modifizierte Variante der Datei

`setEnv.cmd` befindet sich auf der CD im Verzeichnis Implementierung. Sie bestimmt alle Umgebungsvariablen, die zur Übersetztung des Quelltextes der Implementierungsvarianten notwendig sind.

Nachdem die Domain erfolgreich eingerichtet worden ist, können nun die einzelnen Varianten übersetzt und ausgeführt werden. Sie befinden sich in den Unterverzeichnissen zu dem Verzeichnis Implementierung. Abbildung B.1 zeigt dieses Verzeichnis und die Unterverzeichnisse zu einer der Varianten.

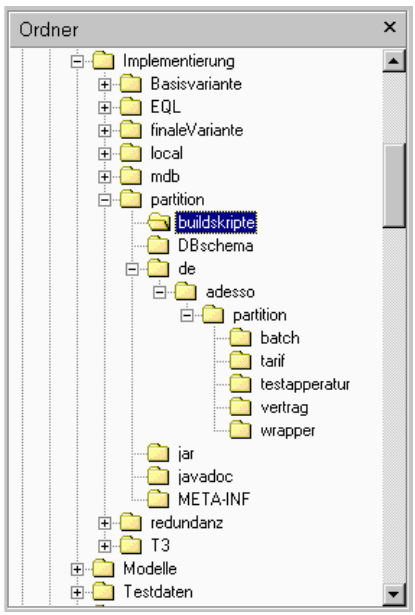


Abbildung B.1: Verzeichnisstruktur der Implementierungsvarianten

Alle Varianten haben die gleiche Verzeichnisstruktur. Die Unterverzeichnisse haben den folgenden Zweck:

- **buildskripte:** Dieses Verzeichnis enthält eine Datei zum Übersetzen der Quelltexte (build.cmd). Sie nutzt zum Übersetzen den `javac`-Compiler von SUN. Für schnellere Übersetzungsläufe kann auch der `jikes`-Compiler von IBM benutzt werden. Das Skript kopiert die fertige `jar`-Datei in ein Unterverzeichnis des Applikationsservers, von wo diese je nach Einstellung sofort gestartet wird. Weiterhin enthält das Unterverzeichnis Dateien zum Erstellen der JavaDoc-Dokumentation (makeImplementierungsDocs.cmd) und zum Starten des Testlaufs (runClient.cmd). Alle genannten Skripte setzen voraus, dass die Pfade in der Datei setEnv.cmd im Verzeichnis Implementierung korrekt gesetzt sind. Bevor der Testlauf gestartet werden kann, sind die Testdaten in der Datenbank anzulegen.
- **DBschema:** In diesem Verzeichnis liegen mehrere SQL-Skripte. Über sie werden `Sequences` zur Erzeugung der Primärschlüssel und die Tabellen zur Speicherung der Beans angelegt. Manche Varianten legen zusätzlich einen Index an, wo es sinnvoll

erscheint. Die Datei **DBAnlegen.cmd** kann genutzt werden, um automatisch die Datenbank auf den nächsten Testlauf vorzubereiten. Dazu werden neben der Anlage der Tabellen auch die Testdaten aus dem Unterverzeichnis **Testdaten** importiert (siehe Abbildung B.1).

- **de:** Dieses Unterverzeichnis ist die Wurzel der package-Struktur, in der sich der Quelltext befindet. Die einzelnen Packages enthalten Klassen für den folgenden Zweck:
  - **batch:** Im package `batch` liegt die Session Bean, die den Batchrahmen ausmacht, die `BatchException` und die Interface-Definitionen für `Sucher` und `Bearbeiter` (vgl. Unterkapitel 7.3). Die Klassen sind ausführlich im Abschnitt 7.3 beschrieben worden.
  - **tarif:** Die `Tarif`-Bean ist ein Dummy, um einen Zugriff auf Tarifinformation für die Beitragsberechnung zu simulieren. Für die Testläufe ist sie so konfiguriert worden, dass ein Zugriff auf sie etwa 60 Millisekunden dauert. Dieser Wert kann durch Einstellungen im Deployment Descriptor variiert werden.
  - **testapparatur:** In diesem package sind alle Klassen enthalten, die zum Starten und Messen einer Variante benötigt werden. `Zeitnehmer` und `Stoppuhr` dienen der Messung (vgl. Abschnitt 7.2). `ManagementClient` wird zum Starten des Batchlaufs angewendet. Die Klasse `Testdatenbestand` dient dem initialen Anlegen der Testdaten (vgl. 7.1). Ist die Datenbank einmalig mit den Testdaten gefüllt, so empfiehlt es sich, die Testdaten mit dem Oracle-Utility `EXP` zu sichern und für weitere Messläufe immer wieder mit `IMP` einzuspielen. Die Klasse `Testdatenbestand` sollte also nur benutzt werden, wenn die Struktur der Testdaten grundsätzlich verändert werden soll. Für Messläufe mit den im Buch beschriebenen Testdaten liegen im Verzeichnis **Testdaten** Dateien für den Import per `IMP` vor.
  - **vertrag:** Dieses Package enthält die Entity Beans, die die Komponente Vertag ausmachen. In diesem package sind ebenfalls die Geschäftsvorfall-spezifischen Klassen `SfrUmstufungSucher` und `SfrUmstufungBearbeiter` enthalten.
  - **wrapper:** In diesem package sind Klassen enthalten, die einige grundlegende Dienste für die Anwendung bereitstellen. `DBHelper` enthält Funktionalität für den Umgang mit Datumswerten und ermöglicht den Zugriff auf `Sequences` zur Erzeugung von Primärschlüsseln. Der zweite Service wird mittlerweile auch direkt vom Applikationsserver angeboten. Der `Logger` delegiert Funktionalität zum Mitprotokollieren von Meldungen und Fehlern an den Logging-Mechanismus des Applikationsservers. Der `ContextHelper` erleichert den JNDI-Zugriff auf Home Interfaces. Damit in der Client-seitigen JVM für den JNDI-Lookup ein `InitialContext` mit `new InitialContext()` angelegt werden kann, muss die Datei **jndi.properties** im Classpath gefunden werden. Das ist möglich, indem sie beispielsweise mit in die jar-Datei gepackt wird (siehe `build.cmd`).
- **jar:** In diesem Unterverzeichnis sind die lauffähigen `jar`-Dateien der Anwendung enthalten. Sie können durch Kopieren in die eingerichtete Domain (siehe oben) sofort in Betrieb genommen werden, ohne dass der Quelltext übersetzt werden muss. Alle Varianten können parallel deployed werden.

- **javadoc:** Dieses Unterverzeichnis enthält die Schnittstellendokumentation im HTML-Format.
- **META-INF:** In diesem Unterverzeichnis befinden sich die Deployment Descriptoren zu den Varianten. Das sind pro Variante die folgenden drei xml-Dateien:
  - **ejb-jar.xml:** ist Teil des EJB 2.0-Standards und enthält die dort vorgeschriebenen Strukturen, wie das abstrakte Schema der Bean oder die Transaktionsattribute. Diese Datei ist auf andere EJB-2.0-kompatible Applikationsserver ohne Änderungen portierbar.
  - **weblogic-cmp-rdbms-jar.xml:** enthält Information zur Umsetzung der Container-Managed Persistence in WebLogic 6.1. Laut Spezifikation wird die Verwendung eines proprietären Deployment Descriptors für CMP vorgeschrieben. Der Inhalt und die Struktur des Deployment Descriptors sind damit applikationsserver-spezifisch.
  - **weblogic-ejb-jar.xml:** enhält für den WebLogic Applikationsserver spezifische Deployment-Information. Das sind u.a. Optionen zum Tuning der Applikation. Damit ist auch diese Datei nicht auf andere Applikationsserver portierbar.

  Nähere Information zu den genannten Dateien ist auf den Web-Seiten der Firma BEA bzw. Sun nachzulesen.

Im Folgenden wird dargestellt, welche Variante sich in welchem Unterverzeichnis befindet und worin auf Ebene des Quelltextes die Besonderheiten der Variante liegen:

- **Basisvariante:** Die Basisvariante befindet sich im gleichnamigen Unterverzeichnis. Sie ist am wenigsten ausgearbeitet und sollte nur mit kleinen Datenmengen ausprobiert werden.
- **EJB QL-Variante:** Diese Variante befindet sich im Unterverzeichnis **EQL**. Sie wird in Abschnitt 8.2 beschrieben. Der Unterschied zur Basisvariante wird vor allem in dem Deployment Descriptor **ejb-jar.xml** deutlich. Dort sind die Datenbankanfragen in EJB QL enthalten.
- **Partitionierte Variante:** Diese Variante unterscheidet sich von der Basisvariante durch die Partitionierung der Datenbanktabellen. Sie ist in Abschnitt 8.3 beschrieben. Die Partitionen werden bei der Anlage des Datenbankschemas durch die Datei **createDB_Partitioniert.sql** erstellt. Das SQL-Skript wird durch die Datei **DBAnlegen.cmd** aufgerufen.
- **Local Interfaces-Variante:** Diese Variante basiert auf der EJB QL-Variante. Der Unterschied wird deutlich durch die zusätzlichen Java-Dateien für die Local Interfaces und die zusätzlichen Einträge in den Deployment Descriptoren. Sie ist ausführlich beschrieben in Abschnitt 8.4. Durch Verwendung von Local Interfaces können Serialisierungskosten reduziert werden.
- **T3-Variante:** Diese Variante basiert ebenfalls auf der EJB QL-Variante. Sie befindet sich im Unterverzeichnis **T3** und wird im Abschnitt 8.4 behandelt. Das T3-Protokoll ist ein BEA-spezifischer Ansatz zur Einsparung von Serialisierungskosten. Der Unterschied zeigt sich in der Belegung des Parameters `enable-call-by-reference` im

Deployment Descriptor weblogic-ejb-jar.xml. Bei der Grundeinstellung des Applikationsservers wird stets das T3-Protokoll genutzt, sodass es in den zuvor beschriebenen Varianten explizit abgeschaltet werden musste, um den Zugewinn an Performance messen zu können.

- **Message-Driven Beans-Variante:** Diese Variante wird in Abschnitt 8.5 erläutert. Sie befindet sich im Unterverzeichnis mdb. Um sie ausführen zu können, muss eine Message-Queue `SfrUmstufungQueue` auf dem Server eingerichtet sein. Die Variante unterscheidet sich von den anderen Varianten dadurch, dass im Package vertrag die Klasse `SfrBearbeiter` als Message-Driven Bean umgesetzt ist. Der `Batchrahmen` fungiert als Sender der Messages, die von dem `SfrBearbeiter` konsumiert werden.
- **Redundanz-Variante:** Diese Variante befindet sich im Unterverzeichnis Redundanz. Bei dieser Variante wird der Zugriff auf den Tarif vermieden. Sie ist ausführlich in Abschnitt 8.6 beschrieben.
- **Finale Variante:** Diese Variante vereinigt alle vorgestellten Lösungen. Sie ist am weitesten ausgearbeitet, sodass in ihr Beispiele für alle Neuerungen des EJB-Standards gefunden werden können. Insbesondere benutzt sie »Write-On-Commit« und demonstriert, wie die Bearbeitung mehrerer Geschäftsobjekte zu einer Transaktion gebündelt werden. Sie nutzt eine Message-Queue `FinaleSfrUmstufungQueue`, die dazu auf dem JMSServer eingerichtet werden muss.

Fragen, Kritik und Anmerkungen zu den Quelltextbeispielen können Sie an Schneider@adesso.de richten.

Literaturverzeichnis

[1] Jochen Alt, Thomas Greb und Marc Volz: *Bewährungsprobe: Einsatz von „Enterprise Java Beans" in einem hochverfügbaren Reservierungssystem*; *OBJEKTspektrum*; 6, S. 51–55; 2000.

[2] Arief und Speirs: *A UML Tool for an Automatic Generation of Simulation Programms*; in *Proceedings Second International Workshop on Software and Performance WOSP 2000*; S. 71–76; 2000.

[3] Bea: *Performance Tuning Your JDBC Application*; 2000; edocs.beasys.com/wls/docs60/jdbc/performance.htm.

[4] Bea: *Online Dokumentation zum Weblogic Server 6.1*; 2001; edocs.beasys.com/wls/docs61/.

[5] *Newsforum der Firma Bea*; 2001; http://newsgroups.bea.com/.

[6] Philip A. Bernstein und Eric Newcomer: *Principles of Transaction Processing*; The Morgan Kaufmann series in data management systems; Morgan Kaufmann; 2. Aufl.; 1997.

[7] Thorsten Bludau: *Portierung von EJB-basierten Anwendungen*; Cand Scient Thesis; University of Dortmund, Department of Informatics, LS 10; Dortmund; 2001.

[8] Michael J. Carey, David J. DeWitt und Jeffrey F. Naughton: *The OO7 Benchmark*; *SIGMOD Record (ACM Special Interest Group on Management of Data)*; 22(2), S. 12–21; 1993.

[9] Susan Cheung und Vlada Matena: *Java Transaction API*; Sun Microsystems, Incorporation; Mountain View, California; 1999.

[10] Lawrence Chung und Brian A. Nixon: *Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach*; in *Proceedings of the 17th International Conference on Software Engineering*; S. 180–188; New York; 1995; ACM Press.

[11] Paul C. Clements: *Coming Attractions in Software Architecture, Technical Report No. CMU/SEI-96-TR-008*; Techn. Ber.; Carnegie Mellon University; Pittsburgh, Pennsylvania; 1996.

[12] Jens Coldewey und Wolfgang Keller: *Objektorientierte Datenintegration*; *OBJEKTspektrum*; 4, S. 20–28; 1996.

[13] Vittorio Cortellessa und Raffaela Mirandola: *Deriving a Queueing Network based Performance Model from UML Diagramms*; in *Proceedings Second International Workshop on Software and Performance WOSP 2000*; S. 58–70; 2000.

[14] Jason Crowe und Mark Johnson: *Measuring the Performance of ARM 3.0 for Java*; 2001; regions.cmg.org/regions/cmgarmw/CMG00_paper_final.pdf Stand 08.2001.

[15] Asbjorn Sigfred Danielsen: *An Evaluation of Two Strategies on Object Persistence*; Cand Scient Thesis; University of Oslo, Department of Informatics; Oslo; 1999.

[16] Linda G. DeMichiel, Sanjeev Krishnan und L. Ümit Yalcinalp: *Enterprise JavaBeans Specification*; 2000; Proposed Final Draft 1, 23. Oktober 2000.

[17] Linda G. DeMichiel, Sanjeev Krishnan und L. Ümit Yalcinalp: *Enterprise JavaBeans Specification*; 2001; Proposed Final Draft 2, 23. April 2001.

[18] Linda G. DeMichiel, Sanjeev Krishnan und L. Ümit Yalcinalp: *Enterprise JavaBeans Specification*; 2001; Final Release, 22. August 2001.

[19] Debabrata Dey, Veda C. Storey und Terrence M. Barron: *Improving database design through the analysis of relationships*; *TODS*; 24(4), S. 453–486; 1999.

[20] *EJB White Paper*; 2000; http://www.java.sun.com/products/ejb/white_paper.html.

[21] Domenico Ferrari: *Measurement and Tuning of Computer Systems*; Prentice-Hall; Englewood Cliffs; 1. Aufl.; 1983.

[22] Candace C. Fleming und Barbara von Halle: *Handbook of relational Database Design*; Addison Wesley Longman GmbH; Reading; 1. Aufl.; 1989.

[23] Martin Fowler und Kendall Scott: *UML konzentriert*; Addison Wesley Longman GmbH; Reading; 2. Aufl.; 2000.

[24] Juergen Friedrichs und Henri Jubin: *Enterprise JavaBeans by Example*; Prentice Hall; Upper Saddle River, New Jersey; 1. Aufl.; 2000.

[25] Frank von Fürstenwerth und Alfons Weiß: *Versicherungsalphabet*; Verlag Versicherungswirtschaft; Karlsruhe; 9. Aufl.; 1997.

[26] Erich Gamma: *Entwurfsmuster*; Addison-Wesley; Bonn [u.a.]; 1. Aufl.; 1997.

[27] Jonathan Gennick: *ORACLE SQL*Plus*; O'Reilly; Sebastopol; 1. Aufl.; 1999.

[28] Jürgen Glag: *Design Guidelines for Large-Scale Batch*; 1999; Vortrag auf der IDUG International DB2 User Group, 1999, www.glag-consult.de.

[29] Frank Griffel: *Componentware. Konzepte und Techniken eines Softwareparadigmas*; dpunkt-Verlag; Heidelberg; 1. Aufl.; 1998.

[30] Volker Gruhn und Andreas Thiel: *Komponentenmodelle: DCOM, JavaBeans, EnterpriseJavaBeans, CORBA*; Addison-Wesley; München; 2000.

[31] Wolfgang Hahn, Fridtjof Toennissen und Andreas Wittkowski: *Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken*; *Informatik Spektrum*; 18, S. 143–151; 1995.

[32] Fried Hoeben: *Using UML Models for Performance Calculation*; in *Proceedings Second International Workshop on Software and Performance WOSP 2000*; S. 77–82; 2000.

[33] Malte Hülder: *Erfassung der zeitabhängigen Entwicklung von Geschäftsobjekten im Kontext der Historisierung von Versicherungsverträgen mit Enterprise JavaBeans*; Cand Scient Thesis; University of Dortmund, Department of Informatics, LS 10; Dortmund; 2001.

[34] Raj Jain: *The Art of Computer Systems Performance Analysis: Techniques for experimental Design, Measurement, Simulation and Modeling*; Wiley; Sebastopol; 1. Aufl.; 1991.

[35] Tyler Jewell: *What's wrong with the EJB 2 specification?*; 2001; www.onjava.com/pub/a/onjava/2001/02/28/ejb.html.

[36] Ralph E. Johnson, Quince D. Wilson und Joseph W. Yoder: *Connecting Buisness Objects to Relational Databases*; Techn. Ber.; Department of Computer Science University of Illinois; Urbana; 1998.

[37] Wolfgang Keller und Jens Coldewey: *Accessing Relational Databases*; in *Technical Report WUCS 97-07*. Washington University Press; 1997.

[38] Georg Koch und Kevin Loney: *ORACLE8 The Complete Reference*; Osborne/McGraw-Hill; Berkeley; 1. Aufl.; 1997.

[39] Craig Larman: *Enterprise JavaBeans 201 — The Aggregate Entity Pattern*; 2000; www.sdmagazine.com/articles/2000/0004/0004c/0004c.htm.

[40] Christoph Lindemann: *Performance Modelling with Deterministic and Stochastic Petri Nets*; John Wiley & Sons; New York; 1. Aufl.; 1998.

[41] Catalina Llado und Peter Harrison: *Performance Evaluation of an Enterprise JavaBean Server Implementation*; in *Proceedings Second International Workshop on Software and Performance WOSP 2000*; S. 180–188; 2000.

[42] Vlada Matena und Beth Stearns: *Applying Enterprise JavaBeans*; The Java Series; Longman Higher Education; Upper Saddle River, New Jersey; 1. Aufl.; 2001.

[43] Heinrich C. Mayr: *Datenbanktechnologie*; Vorlesungsunterlagen; 2000.

[44] Merseguer, Campos und Mena: *A Pattern-Based Approach to Model Software Performance*; in *Proceedings Second International Workshop on Software and Performance WOSP 2000*; S. 137–142; 2000.

[45] Vito Minetti: *Performance Evaluation of a Batch-Time Sharing Computer System using a Trace Driven Model*; in *Modeling and Performance Evaluation of Computer Systems*; Amsterdam; 1976; North-Holland Publishing Company.

[46] Richard Monson-Haefel: *Enterprise JavaBeans*; O'Reilly; Sebastopol; 1. Aufl.; 1999.

[47] Richard Monson-Haefel und David A. Chappel: *Java Message Service*; O'Reilly; Sebastopol; 1. Aufl.; 2001.

[48] Dave Muirhead, Anita Osterhaug et al.: *iCommerce Design Issues and solutions*; Techn. Ber.; Gemstone Systems, Inc.; Beaverton, Oregan; 2000.

[49] Richard J. Niemiec: *ORACLE Performance Tuning*; Osborne/McGraw-Hill; Berkeley; 1. Aufl.; 1999.

[50] Jan Nowitzky: *Partitionierungstechniken in Datenbanksystemen*; *Informatik Spektrum*; 24, S. 345–355; 2001.

[51] Bernd Oestereich: *Objektorientierte Softwareentwicklung*; Oldenbourg Verlag; München, Wien; 4. Aufl.; 1999.

[52] Anthony Ralston, Edwin Reilly und David Hemmendinger (Hg.): *Encyclopedia of Computer Science*; Grove's Dictionaries Inc; New York; 4. Aufl.; 2000.

[53] Claus Rautenstrauch und Andre Scholz: *Vom Performanz Tuning zum Software Performanz Engineering am Beispiel datenbankbasierter Anwendungssysteme*; *Informatik Spektrum*; 22, S. 261–275; 1999.

[54] Georg Reese: *Database Programming with JDBC and Java*; O'Reilly; Sebastopol; 1. Aufl.; 1997.

[55] Mario Schkolnick und Paolo Tiberio: *Estimating the Cost of Updates in a Relational Database*; *TODS*; 10(2), S. 163–179; 1985.

[56] Andreas Schmietendorf, Andre Scholz und Claus Rautenstrauch: *Evaluating the Performance Engineering Process*; in *Proceedings Second International Workshop on Software and Performance WOSP 2000*; S. 89–95; 2000.

[57] Dennis Shasha: *Database Tuning: a Principle Approch*; Prentice-Hall; Englewood Cliffs; 1. Aufl.; 1992.

[58] Dennis Shasha: *Tuning Time Series Queries in Finance: Case Studies and Recommendations*; *Data Engineering Bulletin*; 22(2), S. 40–46; 1999.

[59] Dennis Shasha, François Llirbat et al.: *Transaction Chopping: Algorithms and Performance Studies*; *TODS*; 20(3), S. 325–363; 1995.

[60] Jack Shirazi: *Java Performance Tuning*; O'Reilly; Sebastopol; 1. Aufl.; 2000.

[61] Connie Smith und Lloyd Williams: *Software Performance AntiPattern*; in *Proceedings Second International Workshop on Software and Performance WOSP 2000*; S. 127–136; 2000.

[62] Martin Steppler und Bernhard Walke: *Performance Analysis of Communication Systems Formally Specified in SDL*; in *Proceedings of the First International Workshop on Software and Performance WOSP 98*. ACM Press; 2000.

[63] *Enterprise JavaBeans Specification*; 1998.

[64] Clemens Szyperski: *Component software*; ACM Press book; Addison-Wesley; Harlow [u.a.]; 1. Aufl.; 1997.

[65] Clemens Szyperski: *Emerging Component Software Technologies — a Strategic Comparison*; *Software — Concepts & Tools*; (19); 1998.

[66] Heidi Thorpe: *Oracle8i Tuning and Administration*; Addison-Wesley; Harlow [u.a.]; 2. Aufl.; 2001.

[67] Matthias Ulrich: *Einstellungsfrage*; *ix Magazin für professionelle Informationstechnik*; (9), S. 129–133; 2001.

[68] GDV VAA Arbeitskreis: *Das Objektorientierte fachliche Referenzmodell*; 1999; www.gdv.de/vaa.

[69] GDV VAA Arbeitskreis: *Technische Beschreibung der pVAA*; 1999; www.gdv.de/vaa.

[70] Gerhard Wagner: *Reifeprüfung: Ein Warenwirtschaftssystem mit Enterprise JavaBeans*; *Java Spektrum*; 6, S. 46–50; 2000.

[71] Gio Wiederhold: *Dateiorganisation in Datenbanken*; McGraw-Hill; Hamburg, New York; 1. Aufl.; 1989.

[72] Lloyd G. Williams und Connie U. Smith: *Performance Evaluation of Software Architectures*; in *Proceedings of the First International Workshop on Software and Performance WOSP 98*. ACM Press; 2000.

[73] *Proceedings of the Second International Workshop on Software and Performance WOSP 2000*. ACM Press; 2000.

[74] *Proceedings of the First International Workshop on Software and Performance WOSP 98*. ACM Press; 2000.

Abbildungsverzeichnis

Listingverzeichnis

Index

Symbole

A

B

C

D

E

G

H

I

J

K

L

M

O

P

R

S

T

V

W

Z