Diplomarbeit

Migration von J2EE-Anwendungen

Universität Hamburg
Fakultät MIN, Department Informatik
Zentrum für Architektur und Gestaltung von IT-Systemen

9. Dezember 2008

Henning Stein
Am Kreideberg 32
21339 Lüneburg

Matrikelnummer: 5510612
Studiengang: Wirtschaftsinformatik
E-Mail: 2stein@informatik.uni-hamburg.de

Erstbetreuung: Dr. Axel Schmolitzky
Zweitbetreuung: Prof. Dr. Hartmut Stadtler

Danksagungen

Ich danke Dr. Axel Schmolitzky dafür, dass er sich bereit erklärt hat, der Erstgutachter dieser Arbeit zu sein. Außerdem bedanke ich mich bei ihm für viele Gespräche und Feedback das mir bei der Erstellung dieser Arbeit sehr geholfen hat.

Prof. Dr. Hartmut Stadtler danke ich dafür, dass er sich bereit erklärt hat, Zweitgutachter dieser Arbeit zu sein.

Christian Seipl danke ich für die Erklärung des Bullwhip-Effekts, des Beer Distribution Game sowie für die Dokumentation, die er mir überlassen hat.

Lukas Ciostek danke ich dafür, dass er sich die Zeit genommen hat, mir zu erklären, wie seine Projektgruppe die Migration des Beer Distribution Game durchgeführt hat und mit mir die Unterschiede zwischen unseren Migrationsansätzen diskutiert hat.

Meinem Chef, Ernst Schorning, danke ich dafür, dass er es mir ermöglicht hat, während der Schlussphase dieser Diplomarbeit genug Zeit für diese aufbringen zu können.

Schließlich danke ich Carolin Hamann, dass sie mich während der Diplomarbeit unterstützt hat, mich zur Arbeit angetrieben hat und die für sie größtenteils unverständliche Arbeit Korrektur gelesen hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

BDG	Beer Distribution Game
BMP	Bean-managed persistence
CMP	Container-managed persistence
CMR	Container-managed relationship
CSC	Client Side Container
EIS	Enterprise Information System
EJB	Enterprise JavaBean
EJB QL	Enterprise JavaBeans Query Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IIOP	Internet Inter-ORB Protocol
IOC	Inversion of Control
J2EE	Java 2 Platform, Enterprise Edition
J2SE	Java 2 Platform, Standard Edition
Java EE	Java Platform, Enterprise Edition
Java IDL	Java Interface Definition Language
Java ME	Java Platform, Micro Edition
Java SE	Java Platform, Standard Edition
JCA	Java Connector Architecture
JCP	Java Community Process
JDBC	Java Database Connectivity
JNDI	Java Naming and Directory Interface
JPA	Java Persistence API
JPE	Java Platform for the Enterprise
JPQL	Java Persistence query language
JSF	JavaServer Faces
JSP	JavaServer Pages
JSR	Java Specification Request
JVM	Java Virtual Machine
LCOM	Lack of Cohesion in Methods
LDAP	Lightweight Directory Access Protocol
LOC	Lines of Code
O/R Mapper	objekt-relationaler Mapper
POJI	Plain Old Java Interfaces

POJO	Plain Old Java Object
PU	Persistence Unit
RMI	Remote Method Invocation
RMI-IIOP	Remote Method Invocation over Internet Inter-ORB Protocol
SPI	Java Authentication Service Provider Interface for Containers
WMC	Weighted Methods per Class
XHTML	Extensible HyperText Markup Language
XML	Extensible Markup Language

1 Einleitung

J2EE (*Java 2 Platform, Enterprise Edition*) ist eine Plattform zum Entwickeln von Unternehmensanwendungen. In der aktuellen, fünften Version dieser Plattform hat ihr Hersteller, Sun Microsystems, einige tiefgreifende Veränderungen vorgenommen. Diese wurden mit der Begründung durchgeführt, dass sich dadurch die Komplexität der auf der Plattform basierenden Anwendungen reduzieren ließe (vgl. [Sun08c]).

Die Verbreitung von J2EE lässt sich nur schätzen, da die Hersteller kommerzieller Server die Anzahl der verkauften Lizenzen nicht veröffentlichen. Eine Studie von Forrester erwähnt jedoch, dass etwa 18000 Kunden einen Applikationsserver von IBM und 32000 Kunden einen von Oracle nutzen (vgl. [RGF$^+$]). Ein Applikationsserver wird für J2EE benötigt, kann aber auch für andere Zwecke eingesetzt werden.

Glassfish, ein kostenlos verfügbarer Applikationsserver, wurde laut Herstellerangaben über einen Zeitraum von etwas mehr als drei Jahren etwa 14 Millionen Mal heruntergeladen (vgl. [PL08]).

Insgesamt gibt vermutlich mehrere 10000 Installationen von J2EE-kompatiblen Applikationsservern und auch dementsprechend viele Anwendungen.

Um eine einfache Weiterentwicklung oder Wartung von J2EE-Anwendungen zu ermöglichen, könnte es sinnvoll sein, diese auf die aktuelle Version zu migrieren, um von den Neuerungen profitieren zu können.

1.1 Ziele dieser Arbeit

Ein Ziel dieser Arbeit ist, zu klären, ob und wie sich eine solche Migration durchführen lässt. Außerdem soll geklärt werden, ob eine solche Migration auf die aktuelle Version von J2EE tatsächlich eine Verringerung der Komplexität zur Folge hat. Dazu muss auch bestimmt werden, wie sich die Komplexität von J2EE-Anwendungen messen lässt.

1.2 Aufbau dieser Arbeit

In Kapitel 2 wird der Begriff der Migration definiert und Gründe für Migrationen sowie verschiedene Migrationsansätze dargestellt.

In Kapitel 3 werden die Grundlagen von J2EE und Java EE dargestellt und auf die Besonderheiten der Migration von J2EE-Anwendungen eingegangen.

In Kapitel 4 wird anhand eines Beispiels die Migration einer J2EE-Anwendung dargestellt.

In Kapitel 5 wird anhand von Code-Metriken beurteilt, ob die Migration zu einem weniger komplexen und damit besser wartbaren System geführt hat.

Kapitel 6 enthält das Resümee der Arbeit sowie einen Ausblick.

2 Migration

In diesem Kapitel soll der Begriff der Migration geklärt werden. Außerdem wird auf verschiedene Gründe für eine Migration eingegangen. Es werden Migrationsstrategien und Migrationsprozesse vorgestellt.

2.1 Der Begriff Migration

Das Wort *Migration* stammt aus dem lateinischen und bedeutet soviel wie „(Aus)wanderung“ (vgl. [Wei03]).

In der Literatur werden Migrationen häufig als die Veränderung eines bestehenden Systems (*Altsystem* oder Legacy System) definiert, so dass es mit einer anderen technologischen Basis lauffähig ist (vgl. [FH07, Hil07, SHT05]). Das durch die Migration entstehende System wird in dieser Arbeit *Neusystem* oder *Zielsystem* genannt.

Diese Definitionen beinhalten meistens nur implizit die funktionale Identität zwischen dem Altsystem und dem Zielsystem.

Für diese Arbeit wird die folgende Definition verwandt, die diese Forderung explizit enthält und auch angibt, wie ihre Erfüllung überprüft werden kann:

> Migration bezeichnet die Überführung eines Softwaresystems in eine andere Zielumgebung. Migrationen sind rein technische Transformationen mit einer klaren Anforderungsdefinition. Das zu migrierende Altsystem beschreibt eindeutig die Systemfunktionalität, deren Erhalt nach erfolgreicher Migration durch Regressionstests überprüft werden kann. [GW05, S. 1]

Der Begriff Migration wird in der Literatur vor allem für die Modernisierung von Anwendungen angewandt, die geschäftskritisch sind, die aus mehreren Millionen Zeilen Code bestehen und die eine hohe Verfügbarkeit aufweisen müssen (vgl. [BS95, Sne99]).

Allerdings kann auch jedes andere Software-System, das Zeichen von Software-Alterung (s. unten) zeigt, als Altsystem bezeichnet werden und für eine Migration in Frage kommen.

In [GW05] wird die Migration der gesamten Anwendung als *Software-Migration* bezeichnet und in folgende Teilmigrationen aufgeteilt:

Hardware-Migration bezeichnet den Wechsel der Hardware, z. B. von einem Mainframe auf Rechner der x86-Architektur.

Migration der Laufzeitumgebung bezeichnet den Wechsel der Systemsoftware z. B. eine Migration von Windows zu Linux.

Architektur-Migration bezeichnet den Wechsel von einer Architektur zu einer anderen, z. B. von einer monolithischen Architektur zu einer Schichten-Architektur.

Migration der Entwicklungsumgebung bezeichnet den Wechsel von der Sprache, in der die Anwendung geschrieben wird, zu einer anderen. Z. B. von Cobol zu C++.

2.2 Gründe für Migrationen

Eine Migration kann aus verschiedenen Gründen durchgeführt werden. Dazu zählen folgende vier Punkte:

- Wartbarkeit und Erweiterbarkeit des Altsystems sind gering. Dadurch ist es nicht, oder nur mit großem Aufwand, an sich ändernde Geschäftsprozesse anpassbar (vgl. [Sne99, BS95, GW05, BLWG99]).
- Qualifiziertes Personal, das sich mit dem veralteten System auskennt, ist schwer zu finden (vgl. [Sne99]).
- Das Altsystem verursacht hohe Kosten (vgl. [Sne99, BS95, BLWG99]).
- Die mangelnde Qualität der Schnittstellen des Altsystems erschweren es, es an andere Systeme anzuschließen (vgl. [BLWG99]).

Diese Punkte lassen sich unter dem Begriff *Softwarealterung* (Software Aging) zusammenfassen, für die David Parnas in [Par94] folgende Gründe angibt:

Mangelnde Anpassung an neue Anforderungen führt dazu, dass eine Anwendung nicht mehr zeitgemäß ist. Ein Programm, das vor 20 Jahren geschrieben und nie geändert wurde, würde heute als veraltet gelten.

Anpassungen an neue Anforderungen führen dazu, dass Änderungen von Software-Entwicklern durchgeführt werden, die möglicherweise das ursprüngliche Design des Programms nicht vollständig nachvollzogen haben und somit dieses Design zerstören. Danach verstehen weder die ursprünglichen Entwickler das System, da sie die Änderungen nicht nachvollziehen können, noch die späteren Entwickler; diese haben das System nie verstanden.

Ein Softwareprodukt veraltet also durch mangelnde Anpassung oder wird durch ständige Anpassungen immer komplexer und damit schlechter wartbar.

Als Folgen davon zählt Parnas folgende drei Punkte auf (vgl. [Par94]):

- Schlechte Erweiterbarkeit der Software
- Verschlechterte Zeit-/Platzperformance durch die verfallende Struktur
- Fehleranfälligkeit

Als Möglichkeit zur Verhinderung von Software-Alterung sieht Parnas Dokumentation und *Design for Change*, worunter er versteht, dass der Software-Entwickler bereits beim Gestalten des Systems auf später mögliche Änderungen achten und das System im Hinblick auf diese entwerfen soll (vgl. [Par94]).

Um bereits vorhandene Software am Altern zu hindern, schlägt Parnas Nachdokumentation und Nachmodularisierung vor. Es soll also nachträglich das getan werden, was schon beim Gestalten der Software hätte getan werden müssen (vgl. [Par94]).

Außerdem schlägt er vor, sich Gedanken über die Entwicklung des Systems nach dem ersten Release zu machen, einschließlich eines Plans, wie und wann die Software endgültig abgelöst werden kann (vgl. [Par94]).

Eine mögliche Ablösung von Altsystemen kann eine Migration sein. Dabei können verschiedene Ansätze verfolgt werden, die in den folgenden Abschnitten dargestellt werden.

Das Ziel einer Migration ist wartungsfreundliche, dem aktuellen Stand der Technik entsprechende Software (vgl. [BS95]).

2.3 Migrationsstrategien

Folgende drei Migrationsstrategien sind denkbar (vgl. [Sne99, GW05, BLWG99]):

Neuentwicklung bedeutet, dass das Zielsystem vollständig neu entwickelt wird. Das Altsystem dient lediglich als Quelle für die Spezifikation und die Daten.

Kapselung bedeutet, dass das Altsystem beibehalten wird und das Neusystem seine Funktionalität bereitstellt, indem es auf das Altsystem zugreift. Neue Anforderungen werden dann im Neusystem implementiert. Dies entspricht einer Anwendung des Entwurfsmusters *Adapter* aus [GHJV96] auf Modul- oder Systemebene. In [BLWG99] wird diese Möglichkeit als kurzfristige Lösung angesehen, die langfristig zwei schwer wartbare Systeme hervorbringt.

Konversion bedeutet, dass das Altsystem von einer Programmiersprache in eine andere umgewandelt wird. Dabei kann die Umwandlung evtl. auch teilautomatisiert ablaufen.

2.4 Migrationsprozesse

In der Literatur wird häufig davon ausgegangen, dass eine Mainframe-basierte Anwendung zu migrieren ist. Dabei ist zu beachten, dass nicht notwendigerweise relationale Datenbanken ge-

nutzt werden, sondern z. B. Dateien, die im Dateisystem liegen. Daher ist es häufig nicht möglich, mit mehreren Programmen gleichzeitig auf die selben Daten zuzugreifen. Eines der zentralen Probleme, das die im Folgenden dargestellten Migrationsprozesse lösen müssen, ist daher die Verfügbarkeit des Altsystems zu garantieren, während das Neusystem mit den nötigen Daten versorgt wird (vgl. [BS95]).

2.4.1 Der Cold-Turkey-Ansatz

Der *Cold-Turkey-Ansatz* (auch *Big-Bang-Migration* (vgl. [BLWG99])) versucht, das Neusystem mit modernen Mitteln unter Nutzung des in [Roy70] dargestellten Wasserfall-Modells herzustellen.

Es wird, während das Altsystem weiter genutzt wird, das Neusystem komplett hergestellt. Dabei wird – verkürzt dargestellt – erst das Altsystem vollständig analysiert, das Neusystem vollständig hergestellt und dann die Daten übertragen. Daraufhin übernimmt das Neusystem sämtliche Aufgaben des Altsystems, das dann nicht mehr benötigt wird (vgl. [BS95]).

Laut [BS95] führt dies u. a. zu den folgenden Problemen.

Fehlende Spezifikationen

Die einzige Dokumentation für ein Altsystem ist typischerweise der Quelltext. Die Entwickler des Altsystems sind häufig nicht mehr verfügbar und die Dokumentation nicht vorhanden, lückenhaft oder veraltet. Auch der im Altsystem verwandte Programmierstil wird nicht dem aktuellen Stand der Technik entsprechen. Die Gewinnung einer Spezifikation durch das Analysieren des Codes ist in diesem Fall komplex und teuer.

Undokumentierte Abhängigkeiten

Im Laufe der Zeit können Fremdsysteme an das Altsystem angeschlossen worden sein, die auf das Altsystem zugreifen. Diese Abhängigkeiten sind im Altsystem nicht dokumentiert, müssen aber trotzdem entdeckt und beachtet werden, da sonst die abhängigen Systeme ausfallen würden.

Altsysteme können zu groß sein

Altsysteme müssen häufig eine nahezu 100%ige Verfügbarkeit aufweisen. Allerdings kann das Übertragen der Daten aus dem Altsystem bei einem großen Datenbestand mehrere Tage oder Wochen dauern. Außerdem müssen die Daten für gewöhnlich noch für das neue System aufbereitet werden. Kann nicht für die Zeit, die die Datenübertragung dauern wird, auf das Altsystem verzichtet werden, kann der Cold-Turkey-Ansatz nicht genutzt werden.

Die Schwierigkeit große Projekte zu managen

Altsysteme sind typischerweise sehr große Systeme. Die Entwicklung des neuen Systems wird daher nur mit einer großen Anzahl Entwickler in vertretbarer Zeit gelingen. Große Projekte sind allerdings schwierig handzuhaben und scheitern daher häufig.

Analysis-Paralysis

Eine Migration nach dem Cold-Turkey-Ansatz kann erst dann beginnen, wenn das Altsystem vollständig analysiert ist. Da das Altsystem aber komplex und kompliziert ist, wird immer weiter analysiert, da nicht sichergestellt werden kann, dass das gesamte System verstanden wurde. Da die Analyse nie abgeschlossen wird, kann auch die Migration nicht beginnen.

Insgesamt wird der Cold-Turkey-Ansatz als sehr starr in Bezug auf die Anforderungen an das neue System und als mit einem hohen Risiko behaftet angesehen (vgl. [BS95]). Als Alternative wurde der im nächsten Abschnitt vorgestellte Chicken-Little-Ansatz entwickelt (vgl. [BS95]).

2.4.2 Der Chicken-Little-Ansatz

Der *Chicken-Little-Ansatz* ist ein iterativer Ansatz, der die oben aufgezählten Risiken des Cold-Turkey-Ansatzes vermeiden soll. Das Altsystem bleibt dabei im Einsatz und wird Stück für Stück durch das neue System abgelöst. Durch diesen iterativen Ansatz, sollen die oben geschilderten Probleme in kleine, überschaubare und handhabbare Teilprobleme zerlegt werden, die dann einzeln gelöst werden können (vgl. [BS95]).

In [BS95] wird, ähnlich wie Kent Beck es in [Bec04] tut, argumentiert, dass wenn ein Schritt der Migration fehlschlägt nur dieser Schritt wiederholt werden muss, und nicht das gesamte Projekt fehlschlägt.

Die Probleme verschwinden durch diesen Ansatz nicht, aber die durch sie entstehenden Risiken für das Migrationsprojekt werden überschaubarer.

Gateways

Das iterative Vorgehen soll durch *Gateways* ermöglicht werden. Dabei handelt es sich um Software, die die Anfragen der Benutzer auf das Altsystem und das Zielsystem aufteilt (vgl. [BS95]).

Um Gateways nutzen zu können, teilt der Chicken-Little-Ansatz das Altsystem in folgende drei Schichten ein (vgl. Abbildung 1: [BS95]).

- Schnittstellenschicht
- Anwendungsschicht
- Datenbankschicht

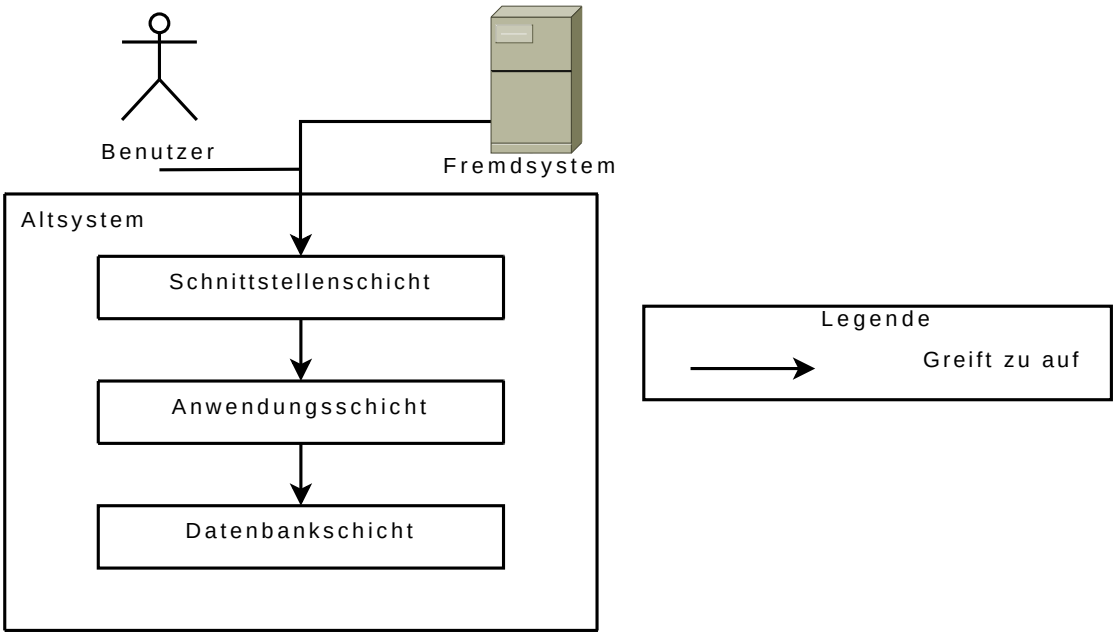

Abbildung 1: Schichten des Altsystems

Gateways sind Programme, die Aufrufe auf das Altsystem und das Zielsystem aufteilen. Dabei können die Gateways zwischen den Benutzern bzw. Fremdsystemen und der Schnittstellenschicht sein oder zwischen zwei Schichten des Altsystems. In Abbildung 2 ist beispielhaft ein Datenbank-Gateway dargestellt. Wo genau Gateways eingesetzt werden, hängt vor allem davon ab, wie gut das Altsystem modularisierbar ist (vgl. [BS95]).

Schritte des Chicken-Little-Ansatzes

Der Chicken-Little-Ansatz beinhaltet 11 Schritte, in denen die Migration durchgeführt wird. Diese Schritte hängen nur teilweise voneinander ab und können auch parallel durchgeführt werden. Die Abhängigkeit der Schritte ist in Abbildung 3 nachvollziehbar.

Diese Schritte werden in jedem Inkrement ausgeführt. Es können auch Schritte ausgelassen werden (vgl. [BS95]).

1. Das Altsystem inkrementell analysieren

Es muss, wie beim Cold-Turkey-Ansatz, das Altsystem analysiert werden, um die Spezifikation des neuen Systems zu erhalten.

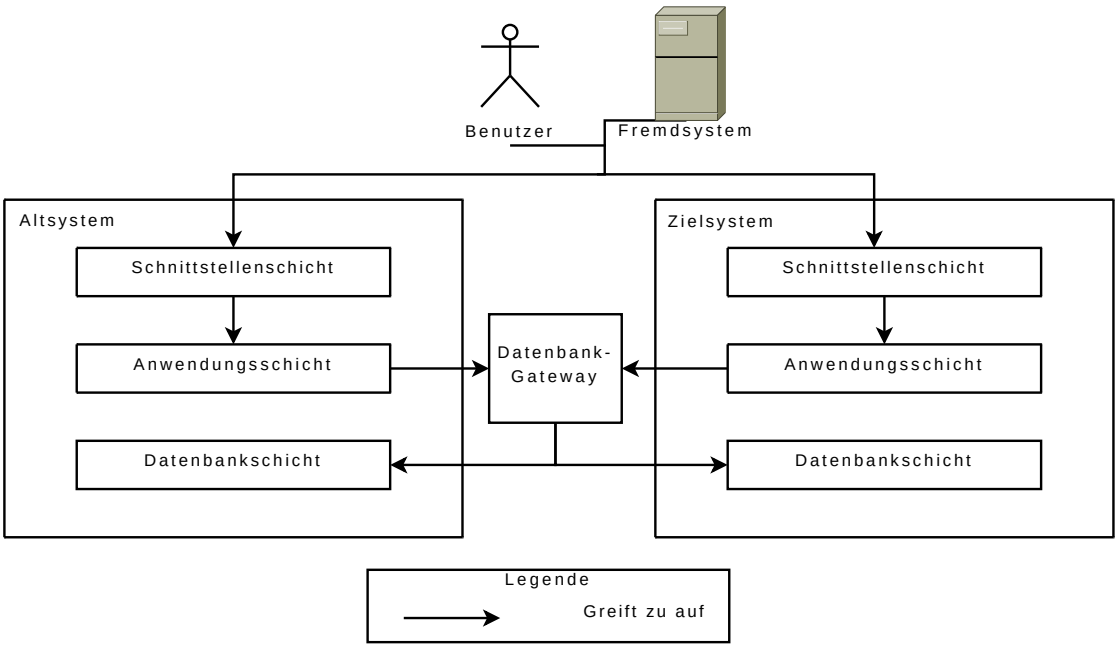

Abbildung 2: Beispiel eines Gateways

2. Das Altsystem inkrementell zerlegen

Das Altsystem soll in einzelne Module zerlegt werden. Dazu muss es eventuell refaktorisiert werden. Das Ziel ist eine schwache Kopplung der Module, so dass sie einzeln durch das neue System ersetzt werden können.

3. Inkrementell die Zielschnittstellen entwerfen

Die Schnittstellen werden entworfen und es wird entschieden, ob ein Gateway für die Schnittstellenschicht benötigt wird.

4. Inkrementell das Zielsystem entwerfen

Es kann das alte System nachgebaut werden oder ähnliche Geschäftsprozesse im Zielsystem entworfen werden.

5. Inkrementell die Zieldatenbank entwerfen

Das Schema für die Datenbank des Zielsystems entwerfen.

6. Die Zielumgebung inkrementell installieren

Die Hardware und die Software, die für den Betrieb des Neusystems benötigt werden, installieren.

7. Benötigte Gateways inkrementell erzeugen und installieren

Im Verlauf der Migrationen werden möglicherweise verschiedene Gateways benötigt. Diese können gekauft oder selbst hergestellt werden.

8. Die Datenbank des Altsystems inkrementell migrieren

Teile der Daten des Altsystems können in das neue System übernommen werden. Dies kann, unter Einsatz eines entsprechenden Gateways, inkrementell geschehen. Dabei müssen die Daten aus der alten Datenbank geladen, gesäubert, konvertiert und in der neuen Datenbank gespeichert werden.

9. Das Altsystem inkrementell migrieren

Entsprechend den Anforderungen an die Anwendung, werden einzelne Module migriert. Kriterien für die Reihenfolge der Migration können sein:

- Einfache Migrierbarkeit
- Verursachte Kosten
- Bedeutung des Moduls für die Anwender

10. Die Benutzungsschnittstellen des Altsystems inkrementell migrieren

Die Benutzungsschnittstelle wird auf die neue Version umgestellt. Um zu verhindern, dass die Benutzer zwischen der neuen und der alten Benutzungsschnittstelle wechseln müssen, kann ein Gateway eingesetzt werden.

11. Inkrementell auf das Zielsystem umsteigen

Das Zielsystem kann benutzt werden. Dabei können erst einzelne Benutzer umsteigen, um das System zu testen. Nach und nach können alle Benutzer auf das Zielsystem umsteigen.

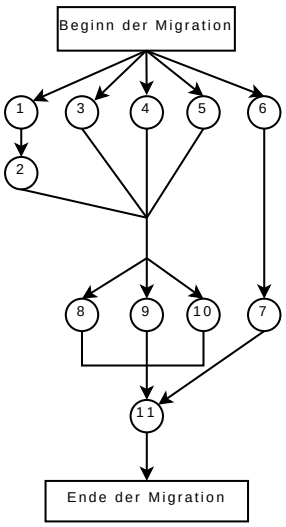

Abbildung 3: Schritte des Chicken-Little-Ansatzes

2.4.3 Der Butterfly-Ansatz

In [BLWG99] wird die Benutzung von Gateways, um eine Migration inkrementell durchführen zu können, kritisiert. So ist ein inkrementelles Umsteigen auf das Zielsystem unter Risikogesichtspunkten zwar zu befürworten, allerdings erhöht die Benutzung von Gateways die Komplexität und damit das Risiko des Migrationsprojekts zu scheitern. Weitere Kritikpunkte an Gateways umfassen, dass Gateways

- keine Unterstützung für Transaktionsmanagement bieten. Daher kann die Konsistenz der Daten zwischen dem Altsystem und dem Zielsystem nicht sichergestellt werden.
- keine Möglichkeit bieten, zwischen Unterschieden in Struktur und Repräsentation der beiden Datebankschemata zu vermitteln
- schwer zu bauen und zu betreiben sind.

Im Zusammenhang mit dem *Chicken-Little-Ansatz* werden in [BLWG99] die Datenbank-Gateways kritisiert. Ein Datenbank-Gateway muss für jeden Datenbankzugriff entscheiden, ob der Zugriff auf die Datenbank des Altsystems, des Zielsystems oder auf beide erfolgen muss. Ist Letzteres der Fall, muss ein Schreibzugriff durch ein 2-Phasen-Commit-Protokoll durchgeführt werden. Die Konsistenz der Daten beim Schreiben in heterogenen Datenbanken zu garantieren ist ein komplexes Problem und daher nur mit viel Aufwand lösbar (vgl. [BS95]).

Der in [BLWG99] dargestellte *Butterfly-Ansatz* geht davon aus, dass Interaktion zwischen dem Altsystem und dem Zielsystem während der Migration unnötig ist. Daher besteht keine Notwendigkeit für die Komplexität erhöhende Gateways (vgl. [BLWG99]).

Als grundlegend für eine erfolgreiche Migration wird die Migration der Daten angesehen. Daher teilt der Butterfly-Ansatz die Migration in zwei Teile.

- Das Kopieren der Daten aus dem Altsystem
- Das Erstellen des Zielsystems

Als erstes werden die Daten im Altsystem für Schreibzugriffe gesperrt. Daraufhin werden die Daten aus dem Altsystem kopiert. Um das Altsystem weiterhin nutzen zu können, werden sogenannte *TempStores* angelegt, die sämtliche Änderungen an den Daten festhalten und auf Anfragen des Altsystems die korrekten Daten herausgeben (vgl. [BLWG99]).

Sind die Daten aus der Datenbank des Altsystems komplett gespeichert, werden die Daten aus dem ersten TempStore kopiert. Hier wird – analog zum Speichern der Datenbank – ein zweiter TempStore benutzt (vgl. [BLWG99]). Dies wird so lange wiederholt, bis die voraussichtliche Zeit, die für das Sichern eines TempStores benötigt wird, ausreichend klein ist. Dann kann das Altsystem abgeschaltet und der letzte TempStore gesichert werden. Daraufhin kann auf das Zielsystem umgeschaltet werden.

Dieser Ansatz hat den Vorteil gegenüber dem Cold-Turkey-Ansatz, dass das Altsystem nur sehr kurz abgeschaltet werden muss, um die TempStores einzurichten.

Gegenüber dem Chicken-Little-Ansatz hat er den Vorteil, dass die Migration jederzeit gestoppt werden kann, bevor das Altsystem abgeschaltet wird (vgl. [BLWG99]). Dies ist beim Chicken-Little-Ansatz nicht möglich, da die Daten teilweise nur im Neusystem und teilweise nur im Altsystem vorhanden sind. Es müsste die bisherige Migration rückgängig gemacht werden.

2.5 Kapitelzusammenfassung

In diesem Kapitel wurde die in dieser Arbeit verwandte Definition des Begriffs *Migration* dargestellt. Es wurden Gründe für Migrationen genannt und Softwarealterung dargestellt. Daraufhin wurden drei Migrationsstrategien vorgestellt und drei Migrationsprozesse erläutert.

3 Kontext: J2EE

Die J2EE soll die Entwicklung von mehrschichtigen Unternehmensanwendungen vereinfachen (vgl. [Sha06]). Sie baut auf der *Java 2 Platform, Standard Edition* (J2SE) auf, die eine virtuelle Maschine, diverse Bibliotheken und Werkzeuge wie Compiler und Debugger für die Sprache Java enthält (vgl. [Sun08a]). Beide Plattformen wurden von der Firma *Sun Microsystems, Inc.* (im Folgenden kurz Sun genannt) entwickelt.

3.1 Geschichte von J2EE

Sun hat im April 1997 die *Java Platform for the Enterprise* (JPE) für Ende 1997 angekündigt (vgl. [Sun97]). Dabei handelt es sich um eine Sammlung von Spezifikationen (vgl. [Sun97]), u. a.:

- Enterprise JavaBeans (EJB)
- Java Database Connectivity (JDBC)
- Java Naming and Directory Interface (JNDI)
- Java Interface Definition Language (Java IDL)

JPE sollte dazu führen, dass Hersteller von Middleware diesen Standard in bereits vorhandener Middleware implementieren oder neue Produkte herstellen. Der Entwickler könnte dann von einem plattformneutralen (*Write Once, Run Anywhere* gilt auch für JPE/J2EE) und herstellerübergreifenden Standard für serverseitige Programme profitieren (vgl. [Mar01]).

Laut [Mar01] war dieses Vorgehen erfolgreich; allerdings gab es auch Kritik an JPE. So gab es keine Möglichkeit zu überprüfen, ob ein Middlewareprodukt tatsächlich den Spezifikationen der JPE entsprach. Außerdem war die Version der verschiedenen enthaltenen Spezifikationen nicht festgelegt, so dass sich der Entwickler um die Integration der verschiedenen Versionen der APIs kümmern musste.

J2EE 1.2 ist die Weiterentwicklung der JPE in Hinsicht auf die oben genannten Probleme. Sie besteht aus mehreren Bestandteilen (vgl. [Sha06]):

J2EE Platform ist eine Plattform zum Bereitstellen von J2EE-Anwendungen, die als Menge von benötigten APIs spezifiziert ist.

J2EE Referenzimplementation ist eine Implementation der J2EE-Spezifikation. Wenn die Spezifikation nicht eindeutig ist, gibt das Verhalten der Referenzimplementation den Standard vor. Die aktuelle Referenzimplementation ist der Glassfish Application Server[1].

J2EE Compatibility Test Suite ist eine Reihe von Kompatibilitätstests, die ein Applikationsserver erfolgreich durchlaufen muss, um sich als J2EE-zertifiziert bezeichnen zu dürfen.

J2EE Blueprints ist eine Sammlung von empfohlenen Vorgehensweisen zur Entwicklung von J2EE-Anwendungen.

Seit J2EE 1.3 wird die Spezifikation im Rahmen des *Java Community Process* (JPC) [2] in der Form von *Java Specification Requests* (JSR) erarbeitet. An der Erarbeitung der Spezifikation ist nicht nur Sun beteiligt, sondern auch andere Softwarehersteller wie IBM, Bea Systems, Hewlett-Packard und Oracle (vgl. [SM00]).

Die Spezifikation kann von verschiedenen Herstellern in Produkte umgesetzt werden. Diese Produkte sollen durch die Standardisierung kompatibel zueinander sein, so dass sie jederzeit gegeneinander ausgetauscht werden können. Dies soll zu einem Wettbewerb zwischen den Anbietern um das beste Produkt führen (vgl. [Rb99]). Allerdings steht es jedem Anbieter frei, Funktionen in seine Produkte einzubauen, die über den Standard hinausgehen.

Mit der 2006 erschienenen 5. Version der Plattform hat Sun den Namen in *Java Platform, Enterprise Edition* (Java EE) geändert. Analog dazu wurde J2SE in *Java Platform, Standard Edition* (Java SE) umbenannt (vgl. [SM08]).

Java-EE-Anwendungen werden für gewöhnlich in *Containern* ausgeführt, die von Applikationsservern bereitgestellt werden. Diese Container stellen transparent für die Anwendung grundlegende Dienste wie deklarative Transaktionen, Authentifizierung, Resource Pooling und Persistenz bereit. Diese Infrastruktur steht den verschiedenen Komponenten automatisch zur Verfügung, so dass sich der Entwickler stärker auf die Umsetzung der Geschäftslogik konzentrieren kann (vgl. [Sha06]).

Im Rahmen dieser Arbeit bezeichnet J2EE alle Version von J2EE 1.0–1.4, während Java EE alle Versionen von J2EE und Java EE bezeichnet. Angaben mit einer Versionsnummer beziehen sich nur auf die genannte Version. Einen Überblick darüber, welche Versionen von J2SE für welche Version von Java EE benötigt wird und welche Version von Enterprise JavaBeans (siehe unten) unterstützt wird, gibt Tabelle 1.

3.2 Schichten einer Java-EE-Anwendung

Java EE teilt Anwendungen in die folgenden vier Schichten ein. Eine grafische Darstellung findet sich in Abbildung 4. In der Abbildung wurden die Web-Schicht und die Geschäftsschicht zum Middle Tier zusammengefasst.

[1] `https://glassfish.dev.java.net`
[2] `http://www.jcp.org`

JAVA EE	JAVA SE	EJB	VERÖFFENTLICHUNG
1.2	1.2	1.1	1999
1.3	1.3	2.0	2001
1.4	1.4	2.1	2003
5	5.0	3.0	2006
6	6	3.1	2009

Tabelle 1: Versionen von Java EE, Java SE und EJB

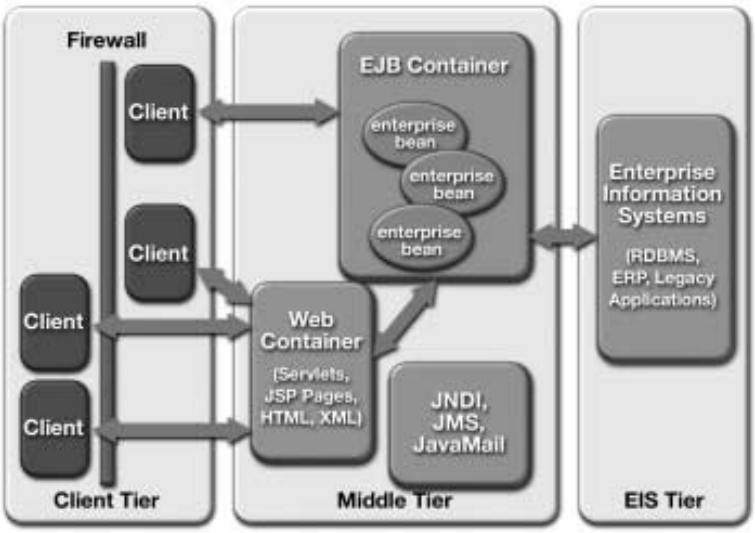

Abbildung 4: Schichten einer J2EE-Anwendung. Quelle: [Sin02]

3.2.1 Client-Schicht

Die *Client-Schicht* (Client Tier) enthält die Benutzungsoberfläche für den Anwender. Dabei sind drei verschiedene Ansätze möglich:

Web-Clients greifen über das *Hypertext Transfer Protocol* (HTTP) auf die Web-Schicht der Anwendung zu. Dabei kann der Client ein gewöhnlicher Web-Browser sein. Es kann sich aber auch um ein auf Java SE oder *Java Platform, Micro Edition* (Java ME) basierendes Programm handeln, dass HTTP-Anfragen erzeugt und die vom Server erhaltenen Antworten interpretiert (vgl. [Sin02]).

EJB-Clients greifen direkt auf die Geschäftsschicht der Anwendung zu. Dazu benötigen sie

einen *Client Side Container (CSC)*. Dieser stellt die Verbindung zum Applikationsserver her und stellt Proxies zum Zugriff auf die Komponenten der Geschäftsschicht zur Verfügung. Der Zugriff auf die serverseitigen Komponenten geschieht hier per *Remote Method Invocation* (RMI) über das *Internet Inter-ORB Protocol* (IIOP). Clients auf Basis von Java ME sind von dieser Art des Zugriffs ausgeschlossen, da die benötigten Klassen für diese Plattform nicht verfügbar sind (vgl. [Sin02]).

EIS-Clients greifen auf die Integrationsschicht der Anwendung zu. Dies soll nur in Ausnahmefällen geschehen, da ein fehlerhafter oder bösartiger Client die Konsistenz und Korrektheit der Daten gefährden kann. Außerdem muss ein Client, um nützlich zu sein, die schon auf dem Server vorhandene Geschäftslogik duplizieren (vgl. [Sin02]).

3.2.2 Web-Schicht

Die *Web-Schicht* (Web Tier) ist die Schicht, mit der ein Web-Client direkt interagiert (vgl. [JBC+06, Sin02]). Die Komponenten der Web-Schicht werden in einem JSP-Container (auch Web-Container) ausgeführt. Java EE spezifiziert zwei Arten von Web-Komponenten:

Servlets sind spezielle Java-Objekte, die vom Applikationsserver ein Request-Objekt erhalten, das die Daten des HTML-Requests enthält. Daraus erzeugt das Servlet eine HTML-Seite, die vom Server an den Client zurückgeschickt wird. Es handelt sich also um Java-Code, der HTML-Seiten erzeugt. Es ist auch möglich, die Darstellung der Antwort an eine JSP zu delegieren (vgl. [Mor07, Sin02]).

JavaServer Pages (JSPs) sind eine Technologie um Java-Code, der vom Applikationsserver vor dem Ausliefern der Seite ausgeführt wird, direkt in statische Seiten (z. B. HTML, XHTML, XML) einzubetten, und so dynamischen Inhalt in die eigentlich statischen Seiten bringt (vgl. [DLR06, Sin02]). JSPs werden vor dem ersten Aufruf auf syntaktische Korrektheit überprüft und dann für gewöhnlich zu einem Servlet kompiliert (vgl. [DLR06, Sin02]).

JSPs und Servlets unterscheiden sich vor allem durch das ihnen zu Grunde liegende Programmiermodell. JSPs sind Präsentationskomponenten; sie sollen also die Sicht des Clients auf das Modell (im Sinne des *Model-View-Controller-Musters* (vgl. [GHJV96])) realisieren. Größere Mengen Java-Code lassen darauf schließen, dass sie Aufgaben des Controllers übernehmen, da die Darstellung der View typischerweise in HTML oder einer anderen Beschreibungssprache geschieht. Servlets wiederum sollten als Controller fungieren und vor allem nicht die Präsentation übernehmen (zu erkennen an viel HTML im Java-Code) (vgl. [Sin02]).

Zusätzlich können zur Organisation der JSPs *JavaServer Faces* (JSF) eingesetzt werden. JSF ist ein Framework zur Entwicklung von Benutzungsoberflächen für Java-Webanwendungen (vgl. [Sun06c]).

3.2.3 Geschäftsschicht

In der *Geschäftsschicht* (Business Tier) wird die Geschäftslogik der Anwendung ausgeführt. Java EE spezifiziert drei Typen von Komponenten in der Geschäftsschicht – die *Enterprise JavaBeans* (EJB). Alle drei Arten von EJB werden in einem EJB-Container ausgeführt.

Entity Beans sind Komponenten, deren Zustand dauerhaft gespeichert wird. Dies geschieht für gewöhnlich in einer relationalen Datenbank, es sind aber auch andere Speicherungsarten denkbar. Sie stellen also eine objektorientierte Sicht auf persistierte Daten dar.

Jede Entity Bean hat einen Primärschlüssel, über den sie eindeutig identifizierbar ist (vgl. [Sun03]). Zusätzlich kann sich ein Client ein sog. Handle geben lassen. Dabei handelt es sich um ein serialisierbares Objekt, das alle Informationen enthält, um eine Referenz auf ein bestimmtes Exemplar einer Entity Bean erhalten zu können. Ein Handle ist nur über das Remote Interface der Entity Bean erhältlich. Es muss das Wiederauffinden der Entity Bean auch über einen Server-Neustart hinweg ermöglichen (vgl. [Sun03]).

Entity Beans bestehen aus einem *Home Interface*, einem *Componenent Interface* sowie einer *Implementationsklasse*.

Das *Home Interface* dient Clients dazu, Entity Beans *zu erzeugen*, *zu finden* und *zu löschen*. Außerdem enthält es Methoden, die sich nicht auf ein bestimmtes Exemplar einer Entity Bean beziehen, sondern auf alle Entity Beans eines Typs. Diese Methoden heißen *Home Business Methods*.

Zum Erzeugen neuer Entity Beans müssen im Home Interface Methoden erstellt werden, deren Name mit `Create` beginnt. Diese werden vom Client aufgerufen. Der Rückgabewert ist die neu erzeugte Entity Bean und ist vom Typ des Component Interface.

Zum Auffinden von Beans gibt es *Finder Methods*. Ihre Namen beginnen mit `Find` und sie liefern eine `Collection` von Entity Beans zurück.

Die Namen der Methoden zum Löschen von Beans heißen `Remove` und werden durch das zu implementierende Interface `EJBHome` vorgegeben. Sie akzeptieren das Handle oder den Primärschlüssel der Bean als Argument.

Es existieren zwei Möglichkeiten, Entity Beans zu persistieren: *Bean-managed persistence* (BMP) und *Container-managed persistence* (CMP).

Beiden Ansätzen ist gemein, dass die Entity Bean *Callback-Methoden* implementieren muss, die vom Container im Falle einer Persistierung bzw. der Wiederherstellung eines Objekts aufgerufen werden.

Im Falle von *CMP* ruft der Container Callback-Methoden der Entity Beans auf, bevor bzw. nachdem er einen Persistierungsvorgang durchführt. In diesen Methoden kann auf die durch den Container durchgeführten Persistierungsvorgänge reagiert werden. Z. B. könnte eine Exemplarvariable, deren Wert nicht persistiert wird, die aber von Werten von

anderen, persistierten Exemplarvariablen abhängt, nach der Wiederherstellung neu berechnet werden, um sicherzustellen, dass die Invariante (im Sinne des Vertragsmodells nach Meyer (vgl. [Mey97])) der Bean erfüllt wird. In dieser Arbeit werden auf diese Art persistierte Exemplarvariablen *CMP-Variablen* genannt.

Im Falle von *BMP* muss der Entwickler die eigentliche Persistierung und Wiederherstellung der einzelnen Exemplarvariablen selber vornehmen. Wenn die Daten in einer relationalen Datenbank gespeichert werden sollen, kann dazu JDBC genutzt werden.

Das *Component Interface* (auch Business Interface) ist die Schnittstelle für die fachliche Logik der Komponente. Die Implementationsklasse der Entity Bean muss das Component Interface nicht implementieren. Der Applikationsserver nutzt das Interface und die Implementationsklasse um eine neue Klasse zu erzeugen, in der die nötigen Aufrufe an den EJB-Container enthalten sind (vgl. [Sha03, Sin02]).

Das Home Interface und das Component Interface können seit EJB 2.0 entweder *Local Interfaces* oder *Remote Interfaces* sein. In früheren Versionen waren nur Remote Interfaces möglich.

Die Remote Interfaces erlauben einen Zugriff auf Objekte aus einer anderen Java Virtual Machine (JVM) und über Netzwerke. Dies geschieht über *RMI over IIOP* (RMI-IIOP). Die Remote Interfaces müssen von `java.rmi.Remote` erben. Daher müssen die im Remote Interface definierten Methoden den Vorgaben für RMI-IIOP genügen. Diese beinhalten, dass jede Methode mit der Klausel `throws RemoteException` versehen werden muss. Die Argumente und Rückgabewerte der Methoden müssen RMI-IIOP-kompatibel sein und evtl. vorhandene Superinterfaces müssen den Vorgaben von RMI-IIOP entsprechen (vgl. [Sun03]).

Werden der Aufrufer und der Empfänger in der gleichen JVM ausgeführt, wird bei Benutzung eines Remote Interfaces dennoch das *Marshalling/Unmarshalling* durchgeführt. Als Marshalling wird das Transformieren der Objektstruktur im Speicher in eine für eine Übertragung über ein Netzwerk geeignete Form bezeichnet. Unmarshalling bezeichnet die umgekehrte Transformation (vgl. [RSL99]).

Dies führt zu einem vermeidbaren Overhead. Allerdings sind Remote Interfaces notwendig, um das von einigen Applikationsservern angebotene Clustering, also die Verteilung der EJB über mehrere Server, zu ermöglichen.

Local Interfaces können nur genutzt werden, wenn der Aufrufer und die Entity Bean in der gleichen JVM ausgeführt werden. Außerdem ändert sich die Semantik des Aufrufs: Wird RMI benutzt, werden die Argumente beim Methodenaufruf als Kopie übergeben. Werden also Veränderungen an den Argumenten vorgenommen, ändert sich das Exemplar des Aufrufers nicht. Bei einem lokalen Aufruf wird eine Referenz auf das Objekt übergeben, so dass es durch die aufgerufene Methode veränderbar ist (vgl. [Sun03]).

Session Beans werden dazu genutzt, den Workflow eines Systems darzustellen. Sie wer-

den nicht persistiert; nach einem Neustart des Servers gibt es also keine Garantie, dass die Session Beans im gleichen Zustand sind wie vorher.

Session Beans können in *Stateful Session Beans* und *Stateless Session Beans* unterteilt werden. Erstere behalten ihren Zustand zwischen verschiedenen Aufrufen des (gleichen) Clients. Letztere haben (aus Sicht des Clients) immer den gleichen Zustand.

Eine Stateful Session Bean kann immer nur durch einen bestimmten Client für die Dauer einer Session benutzt werden (vgl. [Sun03]).

Stateless Session Beans können durch mehrere Clients benutzt werden, da sich ihr (durch den Client beobachtbarer) Zustand durch Aufrufe nicht verändert. Dies ermöglicht die Anwendung von Pooling; also des Vorhaltens mehrerer Exemplare der Stateless Session Bean, um Anfragen schneller beantworten zu können (vgl. [Sun03]).

Eine Session Bean besteht analog zu einer Entity Bean aus einem Component Interface, einem Home Interface und einer Implementationsklasse. Das Home Interface dient zum Erhalten einer Referenz auf eine Session Bean. Auch sie können durch ein Handle referenziert werden, das aber nur solange gültig ist, wie die Session des Clients andauert (vgl. [Sun03]).

Message-Driven Beans sind Ziele für Nachrichten. Diese können vom *Java Message Service* (JMS) oder – seit EJB Version 2.1 – auch von anderen Messaging-Middlewares stammen. Sie bestehen lediglich aus einer Implementationsklasse und beantworten – im Gegensatz zu Entity und Session Beans – Anfragen asynchron (vgl. [Sun01, Sun03]).

EJBs finden sich gegenseitig durch das *Java Naming and Directory Interface*. JNDI bietet eine Schnittstelle zu Verzeichnisdiensten wie z. B. dem *Lightweight Directory Access Protocol* (LDAP). Dabei bringt ein Applikationsserver typischerweise eine eigene Implementation eines Verzeichnisdienstes mit. Dieser wird z. B. genutzt, um Datenbankressourcen und EJBs zu finden. Es existiert ein *globaler JNDI-Namensraum*, über den Einträge auch in anderen Anwendungen und auf anderen Servern auffindbar sind und ein *lokaler JNDI-Namensraum*, in dem Objekte nur innerhalb einer Anwendung auffindbar sind (vgl. [SM07]).

Das Suchen eines Eintrags im JNDI wird als *JNDI-Lookup* bezeichnet.

3.2.4 Integrationsschicht

Die *Integrationsschicht* (auch Enterprise Information System (EIS)-Tier) enthält andere Systeme, mit denen die J2EE-Anwendung interagieren soll. Dies können z. B. Datenbanksysteme, Mainframes oder andere Altsysteme sein (vgl. [Sin02]). Dabei stellt J2EE verschiedene APIs zur Verfügung, um auf andere Systeme zuzugreifen. Diese sind:

J2EE Connector Architecture Die *J2EE Connector Architecture* (JCA) definiert Schnittstellen, um J2EE-Anwendungen an bereits bestehende EIS anzubinden. Dies geschieht

durch *JCA Resource Adapter*, die als Plug-In für den Applikationsserver installiert werden. Ein JCA Resource Adapter kann in jedem Java-EE-kompatiblen Applikationsserver installiert werden. Der Applikationsserver sorgt in Zusammenarbeit mit dem Resource Adapter für Transaktionssicherheit und Connection Pooling und übernimmt auch die Authentifizierung am Fremdsystem.(vgl. [Sha03]).

Java Database Connectivity (JDBC) definiert eine Schnittstelle zum Zugriff auf relationale Datenbanken. Sie dient zur Herstellung von Verbindungen zu Datenbanken und dem Ausführen von SQL-Abfragen. Um auf die Datenbank zugreifen zu können, ist ein zu der jeweiligen Datenbanksoftware passender Resource Adapter notwendig (vgl. [Sha03]).

3.3 Packaging und Deployment

Java-EE-Anwendungen können aus folgenden fünf Komponententypen bestehen (vgl. [Sin02]):

- EJBs
- Servlets und JSPs
- Applets
- Application Clients
- Connectors

Diese Komponenten werden zu *Modulen* zusammengefasst, die .jar-Dateien ähnlich sind, allerdings eine eigene Dateiendung haben und einer bestimmten Struktur entsprechen müssen. Es existieren vier Modultypen (vgl. [Sin02]):

EJB Module enthalten Enterprise JavaBeans und zugehörige Klassen.

Web Module enthalten die Komponenten der Web-Schicht und die zugehörigen Ressourcen, wie Webseiten.

Application Client Modules enthalten Klassen für Application Clients.

Adapter Modules enthalten Resource Adapter und unterstützende Bibliotheken und Ressourcen.

Mehrere dieser Module können zu einer *Java-EE-Anwendung* zusammengefasst werden. Diese hat auch eine vorgeschriebene Struktur und Dateiendung. Dieser Vorgang des Zusammenstellens von Komponenten zu Modulen und von Modulen zu Enterprise Applications wird als *Packaging* bezeichnet. Module sind die kleinsten Einheiten, die auf einem Server deployed werden können (vgl. [Sin02]).

Als *Deployment* wird das Installieren und Konfigurieren der Module und Enterprise Applications bezeichnet. Jede Deployment-Einheit enthält einen Deployment Deskriptor. Dabei handelt es sich um eine XML-Datei, die zwei Arten von Informationen enthält:

Strukturelle Information beschreibt die Komponenten in dem jeweiligen Modul. So werden z. B. die Home und Component Interfaces von Entity Beans dort angegeben. Eine Änderung führt dazu, dass die Anwendung nicht mehr richtig funktionieren kann.

Zusammenstellungsinformation beschreibt das Verhältnis der Komponenten untereinander. Dort kann definiert werden, auf welche Bean eine andere Bean zugreift oder welcher Benutzer welche Methoden aufrufen darf. Eine Änderung dieser Informationen ändert das Verhalten der Anwendung, ohne sie notwendigerweise funktionsunfähig zu machen.

3.4 Kritik an J2EE

Teile von J2EE und besonders die Entity Beans wurden als zu komplex kritisiert. Kritik an Entity Beans in der Version 1.1 umfasst u. a:

Container-managed relationships *CMR* waren im Standard nicht spezifiziert. CMR soll automatisch Relationen zwischen Entity Beans verwalten. So war es nicht möglich, bestimmte Exemplare von Entity Beans zu löschen, wenn alle sie referenzierenden Beans ebenfalls gelöscht wurden. CMR wurde mit EJB 2.0 standardisiert (vgl. [Sun01, BK05, Joh05]). Exemplarvariablen, die CMR nutzen, werden in dieser Arbeit *CMR-Variablen* genannt.

Es existierte keine Abfragesprache, um Entity Beans zu finden. Es musste Applikationsserver-spezifischer Code geschrieben werden, um Entity Beans finden zu können. Dies wurde durch die Einführung der *Enterprise JavaBeans Query Language* (EJB QL), einer SQL-ähnlichen Abfragesprache, in EJB 2.0 geändert. In EJB 2.1 wurde diese Abfragesprache erweitert (vgl. [Sun01, Sun03, BK05, Joh05]).

Schlechte Testbarkeit Zusätzlich ist die Testbarkeit von EJBs schlecht, weil sie nur innerhalb eines Containers lauffähig sind. Dies erschwert auch häufiges Testen bzw. testgetriebene Entwicklung, da die Anwendung nach jeder Änderung wieder deployed werden muss, um die Tests direkt im Container auszuführen. Als Alternative kann ein Mock-Up eines EJB-Containers programmiert werden, was aber möglicherweise zur Reimplementation eines EJB-Containers führt (vgl. [Joh05, BK05]).

Keine Polymorphie Vererbung zwischen Entity Beans ist zwar möglich, allerdings sind keine polymorphen Aufrufe möglich. Dies erschwert es, objektorientiert zu programmieren (vgl. [Joh05, BK05]).

In den EJB Versionen 2.0 und 2.1 wurden Verbesserungen umgesetzt, allerdings konnten diese nicht alle Kritikpunkte beseitigen und es hatten sich als Folge der Unzulänglichkeiten von EJB 1.1 bereits Alternativen etabliert (vgl. [BK05]).

Hibernate[3] und *Spring*[4] sind zwei populäre Beispiele dafür.

3.4.1 Hibernate

Hibernate ist ein objekt-relationaler Mapper (O/R Mapper) und damit eine Alternative zur Persistierung durch Entity Beans. Vorteile von Hibernate sind, dass die Objekte, die die zu persistierenden Daten enthalten, POJOs sind. Als *Plain Old Java Object* (POJO) bezeichnet Martin Fowler reguläre Java Objekte, im Gegensatz zu z. B. EJB, die ohne Container nicht funktionieren können (vgl. [Fow]). Mit *Plain Old Java Interface* (POJI) werden analog Interfaces bezeichnet, deren Implementation nicht dazu führt, dass die implementierende Klasse nur noch z. B. in einem Container ausführbar ist. Somit können Vererbung und Polymorphie genutzt werden und die Testbarkeit ist höher.

3.4.2 Spring

Spring ist ein Framework, das die Entwicklung von J2EE-Anwendungen erleichtern soll. Dabei setzt es auf Dependency Injection (vgl. [Fow04]) und aspekt-orienterte Programmierung (vgl. [EFB01]). Außerdem bietet es eine Alternative zu EJB, indem es Transaktionen mit POJOs (die nicht den gleichen Einschränkungen wie EJBs unterliegen) unterstützt (vgl. [Joh05]).

3.5 Wesentliche Neuerungen in Java EE 5

Das Ziel bei der Entwicklung von Java EE 5 war die Erleichterung der Entwicklung gegenüber J2EE (vgl. [Sun08c]). Um dies zu erreichen nutzt Java EE 5, inspiriert von Hibernate und Spring, Ansätze wie aspekt-orientierte Programmierung und Dependency Injection sowie die aus Java SE bekannten Annotationen statt der Deployment Deskriptoren.

3.5.1 Annotationen statt Deployment Deskriptoren

In Java EE 5 können erstmals die aus J2SE 5.0 bekannten Annotationen (vgl. [Sun06a, GJSB05]) statt Deployment Deskriptoren benutzt werden. D. h. die Informationen, die vorher getrennt vom Programmcode in XML-Dateien standen, können jetzt direkt in den Code eingebettet werden. Um die Konfiguration der Anwendung weiterhin zu ermöglichen, sind Deployment Deskriptoren allerdings weiterhin zugelassen und gelten vorrangig vor Annotationen (vgl. [Sha06, EJB06]).

[3] `http://www.hibernate.org`
[4] `http://www.springframework.org/`

3.5.2 Java Persistence API Entities

Java Persistence API Entities (JPA Entities, Entities) stellen eine leichtgewichtige Alternative zu Entity Beans dar und sollen sie in Zukunft ganz ersetzen (vgl. [Sun06a]).

EJB 1.1 Entity Beans mit CMP sind bereits in Java EE 5 als veraltet (deprecated) gekennzeichnet und sollen nicht mehr benutzt werden (vgl. [Sha06]).

JPA Entities sind als POJOs realisiert. D. h. sie müssen nicht, wie die Entity Beans, von einer vorgegebenen Klasse erben, um persistiert werden zu können. Somit ist es möglich, eine fachliche Vererbungshierarchie zu modellieren. Polymorphe Aufrufe sind möglich und sie sind außerhalb eines Containers lauffähig. Dies ermöglicht auch die Nutzung eines Test-First-Ansatzes bei der Entwicklung.

Außerdem entfallen das Home Interface und die Finder-Methoden; da es sich um POJOs handelt, können JPA Entities einfach über den `new`-Operator erzeugt werden. Die Persistenz wird weiter durch den Container verwaltet, allerdings ist – im Gegensatz zu Entity Beans – eine manuelle Persistierung durch das Aufrufen einer Methode an einem durch Dependency Injection vom Container zur Verfügung gestellten Entity Manager nötig.

Eine weitere Verbesserung betrifft das Mapping zwischen Java-Typen und den Typen einer relationalen Datenbank. Dies war bei EJB applikationsserverspezifisch, was zu einer Verringerung der Portabilität der Anwendung führte. Für JPA Entities ist das Mapping vorgeschrieben und die Persistierung wird vom Container an einen Pluggable Persistence Provider delegiert. Diese Persistence Provider sind, wenn sie der Spezifikation JPA 1.0 genügen, gegeneinander austauschbar (vgl. [Sun06b]).

Im Deployment Deskriptor für die Entities (`persistence.xml`) wird der Name der *Persistence Unit* (PU) gesetzt. Dabei handelt es sich um eine Organisationseinheit für Entities, innerhalb derer jeder Name eines Entities eindeutig sein muss. Der Deployment Deskriptor wird in einem Archiv einer Anwendung gespeichert. Die Entities sind nur in dieser Anwendung und den dieser Anwendung untergeordneten Anwendungen sichtbar. Eine Anwendung kann mehrere PUs haben (vgl. [Sun06b]). Außerdem werden in diesem Deployment Deskriptor auch der zu benutzende Persistence Provider angegeben. Dabei handelt es sich um eine Klasse, die als Fabrik im Sinne des Entwurfsmusters *abstrakte Fabrik* aus [GHJV96] `EntityManager` erzeugt. Der `EntityManager` wird wiederum in der Anwendung genutzt, um Entities anzulegen, zu finden und zu löschen.

3.5.3 Configuration by Exception

In Java EE 5 wurde, so weit es möglich war, eine Standardeinstellung für verschiedene Konfigurationsoptionen der EJB vorgegeben. Ist ein davon abweichendes Verhalten gewünscht, muss dies durch den Entwickler konfiguriert werden. Dies reduziert die Anzahl der zur Konfiguration nötigen Annotationen.

Dieses von Sun *Configuration by Exception* genannte Konzept ist vor allem im Zusammenhang mit der Sprache Ruby unter dem Namen *Convention over Configuration* bekannt geworden

und in [Che06] als Entwurfsmuster formuliert.

3.6 Ausblick auf Java EE 6

Java EE 6 sollte als JSR 313 entwickelt werden. Allerdings wurde der JSR aus unbekannten Gründen zurückgezogen. Etwas später wurde mit der Entwicklung von Java EE 6 als JSR 316 begonnen (vgl. [Sun08d, Sun08e]).

Am 22. November 2008 ist ein früher Entwurf der Spezifikation für Java EE 6 veröffentlicht worden (vgl. [CS08]). Aufgrund der Vorläufigkeit dieses Entwurfs werden die im folgenden dargestellten Änderungen nicht notwendigerweise in der finalen Version von Java EE 6 enthalten sein.

3.6.1 Profile

Profile stellen eine Möglichkeit dar, die Java-EE-Plattform so zu konfigurieren, dass sie besser für einen bestimmten Anwendungsbereich geeignet ist. Zu diesem Zweck können Teile der Spezifikation, die als optional gekennzeichnet und für den Anwendungsbereich nicht nützlich sind, entfernt werden.

Es können auch Technologien zu einem Profil hinzugefügt werden, die nicht zum Java-EE-Standard gehören. Dies setzt voraus, dass die Integration dieser Technologien so erfolgen kann, dass der Java-EE-Standard weiterhin befolgt wird.

Profile können auch Technologien als optional kennzeichnen. Diese können, müssen aber nicht, implementiert werden.

Ein Applikationsserver darf ein oder mehrere Profile oder die gesamte Spezifikation unterstützen. Die Profile sind nicht Teil der Java-EE-Spezifikation sondern werden als eigene JSRs im Rahmen des JCP definiert (vgl. [CS08]).

3.6.2 Pruning

Java EE 6 spezifiziert einen *Pruning* genannten Prozess, um nicht mehr benötigte Teile der Spezifikation zu entfernen. Bisher war jede Version von Java EE bis auf Kleinigkeiten zu ihren Vorgängerversionen kompatibel. In Zukunft werden Technologien in der Spezifikation als optional gekennzeichnet werden, so dass diese Kompatibilität nicht mehr notwendigerweise gegeben sein wird.

Der Prozess besteht aus zwei Schritten:

- Die Expertengruppe, die eine Spezifikation erarbeitet, schlägt in der Spezifikation für Version N vor, eine Technologie optional zu machen. Dies wird in der Spezifikation dokumentiert.

- Die Expertengruppe, die die Spezifikation für Version $N + 1$ erarbeitet, entscheidet, ob die Technologie optional werden soll, ob es auf der Liste der optional zu machenden Technologien bleibt oder ob es von dieser Liste entfernt werden soll.

Wird die Technologie optional, darf ein standardkonformes Produkt es nach wie vor implementieren, muss es aber nicht.

3.6.3 EJB 3.1

Die Spezifikation von EJB 3.1 wird Teil von Java EE 6 sein (vgl. [CS08]). Die wesentlichen Änderungen werden voraussichtlich sein:

Kandidaten für Entfernung

In der Spezifikation von EJB 3.1 werden einige Technologien der alten Spezifikation entsprechend dem oben dargestellten Prozess zum Entfernen vorgeschlagen. Dazu zählen u. a. die folgenden fünf Technologien (vgl. [EJB08]):

- EJB 1.1 Entity Beans mit CMP
- EJB 2.1 Entity Beans mit BMP
- EJB 2.1 Entity Beans mit CMP
- Client-Sicht auf EJB 2.1 Entity Beans. D. h. die Benutzung von Remote und Local Home Interfaces
- EJB QL, die Abfragesprache für EJB 2.1 Entity Beans mit CMP

Dabei handelt es sich jeweils um Teile der Spezifikation, die sich mit Entity Beans befassen, die in Java EE 5 durch Entities ersetzt wurden.

Singleton Session Beans

Singleton Session Beans sind eine neue Art von Session Beans. Sie existieren, entsprechend dem Entwurfsmuster *Singleton* aus [GHJV96], innerhalb einer Anwendung nur einmal. Dies gilt auch, wenn die Anwendung über mehrere virtuelle Maschinen verteilt ist. Sie behält ihren Zustand nach Aufrufen (vgl. [EJB08]).

Singleton Session Beans erlauben parallelen Zugriff aus mehreren Threads. Dabei ist es möglich, die Synchronisation durch den Container durchführen zu lassen. Dies wird *container-managed concurrency* genannt. Programmiert der Entwickler die Synchronisation selbst, wird von *bean-managed concurrency* gesprochen (vgl. [EJB08]).

Wird nicht angegeben, welche Art von Nebenläufigkeitsverwaltung genutzt werden soll, wird container-managed concurrency verwandt (vgl. [EJB08]).

Als Voreinstellung verursacht jeder Methodenaufruf eine Schreibsperre (oder exklusiven Zugriff), d. h. solange diese Methode ausgeführt wird, kann keine andere Methode der Singleton Session Bean ausgeführt werden (vgl. [EJB08]).

Wird eine Methode mit `@Lock(Read)` annotiert, verursacht der Methodenaufruf eine Lesesperre (geteilter Zugriff). Solange diese Methode ausgeführt wird, können beliebige andere Methoden, die ebenfalls eine Lesesperre benötigen ausgeführt werden. Die Ausführung von Methoden mit Schreibsperre ist während der Ausführung dieser Methode nicht möglich (vgl. [EJB08]).

Wird bean-managed concurrency verwandt, muss der Entwickler mit den aus Java SE bekannten Möglichkeiten, wie den Schlüsselwörtern `synchronized` und `volatile`, für Synchronisation sorgen (vgl. [EJB08]).

Asynchrone Methodenaufrufe

Eine weitere Neuerung ist die Möglichkeit asynchroner Methodenaufrufe an Session Beans (vgl. [EJB08]).

Zustandsbehaftete Webservices

Ab EJB 3.1 wird es möglich sein, Stateful Session Beans als Webservice zu verwenden und somit einfach zustandsbehaftete Webservices realisieren zu können (vgl. [EJB08]).

Optionale Local Business Interfaces

Es kann auf Local Business Interfaces verzichtet werden. Die Implementationsklasse ist ausreichend (vgl. [EJB08]).

Callbacks auf Applikationsebene

Es werden neue Callbacks auf Applikationsebene eingeführt. Dazu zählen auch Callbacks, wenn der Container initialisiert oder heruntergefahren wird (vgl. [EJB08]).

Verbesserungen am EJB Timer Service

Ab Java EE 6 wird ein, dem Unix-Programm *cron* nachempfundes, Scheduling möglich sein. Es können also Callback-Methoden zu bestimmten Zeitpunkten aufgerufen werden. Außerdem können Timer zum Zeitpunkt des Deployments erzeugt werden, statt wie bisher nur bei einem Aufruf einer Methode des Objekts. Diese Möglichkeiten können nicht von Stateful Session Beans genutzt werden (vgl. [EJB08]).

EJB in Java SE

EJB Version 3.1 spezifiziert die *eingebette Nutzung* (*Embedded Usage*) von EJBs. Es ist möglich aus Java SE einen eingebetteten EJB-Container zu instanziieren, in dem die EJBs ausgeführt werden. Dies löst das in Abschnitt 3.4 dargestellte Problem der Testbarkeit und bietet auch die Möglichkeit, EJBs in Java-SE-Anwendungen zu nutzen (vgl. [EJB08]).

EJBs im Servlet Container

Ab Version 3.1 wird es möglich sein, EJB auch im Web Container zu deployen (vgl. [EJB08]). Unter anderem um dies zu ermöglichen, werden die Regeln für das Packaging von EJBs vereinfacht (vgl. [EJB08]).

3.6.4 Web Beans

Java EE 6 beinhaltet die in JSR 299 definierte Technologie der *Web Beans*.

Web Beans dienen dazu, die Persistenzschicht mit der EIS-Schicht zu integrieren. Sie können in beliebige Container-verwaltete Klassen injiziert werden. Zusätzlich können sie direkt durch JSF genutzt werden (vgl. [Web08]).

In älteren Versionen des Standards musste der Zugriff von JSF auf die Anwendung durch *JavaBeans* (nicht EJB) erfolgen (vgl. [Sun08b]). JavaBeans sind gewöhnliche Java-Klassen, die einen parameterlosen Konstruktor und öffentliche Methoden, deren Namen mit `get` bzw. `set` beginnen um auf ihre Exemplarvariablen zuzugreifen. Außerdem sind sie serialisierbar (vgl. [Gra97]).

Session Beans sind automatisch auch Web Beans, so dass sie ohne weitere Konfiguration durch JSF genutzt werden können (vgl. [Sun08b]).

3.6.5 Java Persistence API 2.0

Erweiterung des O/R-Mappings

Das objekt-relationale Mapping wurde erweitert. Es können jetzt auch Collections von *eingebetteten Typen* (embeddable types) und verschachtelte eingebettete Typen genutzt werden. Eingebettete Typen sind Entities, die fest zu anderen Entities gehören und gemeinsam mit ihnen persistiert werden. Sie dienen vor allem dazu, die Performance zu steigern, da sie in der gleichen Tabelle wie die Objekte in die sie eingebettet sind gespeichert werden. Dies vermeidet Joins auf der Datenbank.

Es ist jetzt auch möglich sortierte Listen zu speichern und ihre Ordnung dabei zu erhalten. Vorher war die Erhaltung der Ordnung nicht garantiert (vgl. [Jav08]).

Desweiteren muss der Deployment Deskriptor für die JPA, `persistence.xml`, jetzt vom Container beim Deployment validiert werden (vgl. [Jav08]).

Erweiterung der JPQL

Die *Java Persistence query language* (JPQL) wurde an die Neuerungen bei den eingebetteten Typen angepasst. Außerdem unterstützt sie in der neuen Version Funktionen zur String-Manipulierung, arithmetische Funktionen und Zeit-/Datumsfunktionen in Select-Anweisungen (vgl. [Jav08]).

Zusätzlich sind mittels der Anweisungen `CASE`, `NULLIF` und `COALESCE` bedingte Ausführungen von JPQL-Abfragen möglich (vgl. [Jav08]).

Es ist möglich, statt der statischen, String-basierten Queries aus Java EE 5, dynamisch Queries aus Objekten zu erstellen (vgl. [Jav08]).

Hints

Es wurden *Hints* standardisiert, die für Datenbankabfragen und die Konfiguration des Entity Managers genutzt werden können. Hints sind Hinweise an den Applikationsserver, wie er sich verhalten soll. Die Befolgung von Hints ist nicht obligatorisch. In der Java Persistence API 2.0 werden Hints eingeführt, die bei Datenbankabfragen Timeouts erlauben (vgl. [Jav08]).

3.6.6 Java Authentication Service Provider Interface for Containers

Ein weitere Neuerung ist die Einführung von *Java Authentication Service Provider Interface for Containers* (SPI). SPI definiert einen JCA Resource Adapter, der Authentifizierungsmechanismen eines Containers über eine standardisierte Schnittstelle bereitstellt (vgl. [Mon07]).

Bisher war es nötig, eine Java-EE-Applikation, die Authentifizierungsmechanismen eines Containers nutzen sollte, für jeden Applikationsserver anzupassen, da die Authentifizierungsmechanismen jeweils eigene Schnittstellen hatten.

3.6.7 Servlets

Servlets werden in der Version 3.0 in Java EE 6 enthalten sein (vgl. [CS08]).

Die wesentliche Neuerung bei den Servlets ist, dass sie auch mit Annotationen statt mit Deployment Deskriptoren konfiguriert werden können.

Da Servlets häufig nicht selber programmiert, sondern vorgefertigte Klassen aus Frameworks benutzt werden, ist dies nicht ausreichend, um die Konfiguration zu vereinfachen. Die Servlets müssten noch per Deployment Deskriptor konfiguriert werden, weil die Annotationen Bestandteil der Klasse aus dem Framework sind (vgl. [Chi08]).

Um trotzdem den Konfigurationsaufwand zu verringern, wurden *Web Fragments* eingeführt. Dabei handelt es sich um Deployment Deskriptoren, die nur den Teil der `web.xml` enthalten, der typischerweise in der oben geschilderten Situation geändert werden muss. Außerdem werden asynchrone Aufrufe von Servlets spezifiziert.

3.7 Migrierbarkeit von J2EE-Anwendungen

Migrationen von J2EE-Anwendungen unterscheiden sich von denen in Kapitel 2 dargestellten Migrationen vor allem dadurch, dass die enge Kopplung zwischen Programm und Daten nicht mehr vorhanden ist. J2EE-Anwendungen speichern ihre Daten für gewöhnlich in relationalen Datenbanken, auf die mehrere Programme gleichzeitig zugreifen können.

Eine Konstruktion wie Gateways oder der Butterfly-Ansatz dürften also zur Migration von J2EE-Anwendungen nicht nötig sein. Java EE 6 ist, bis auf geringfügige Ausnahmen, kompatibel zu allen vorangegangenen Versionen der J2EE-Spezifikationen. Es ist also möglich die Anwendung iterativ zu migrieren und dabei den gleichen Applikationsserver zu verwenden.

Bei der Migration von einem Server auf einen anderen handelt es sich um eine Migration der Laufzeitumgebung.

Werden EJBs migriert, ist die Einordnung schwieriger, da sie in der gleichen Laufzeitumgebung bleiben können, da Applikationsserver zu den älteren Versionen kompatibel sein müssen (vgl. [Sha06]). Auch wenn sich die Laufzeitumgebung nicht tatsächlich ändert, könnte argumentiert werden, dass die EJBs vor der Migration eine andere Laufzeitumgebung brauchen als danach. Nämlich z. B. erst eine Laufzeitumgebung für EJB 2.0 und danach eine für EJB 3.0 oder JPA Entities.

3.7.1 Migration von einem Server auf einen anderen

Java EE-Anwendungen sollten sich sehr leicht von einem Server auf einen anderen migrieren lassen.

Hält sich die Anwendung an den Standard, benutzt also keine Server-spezifischen Erweiterungen, müssen die Server-spezifischen Deployment Deskriptoren für den neuen Server erstellt werden. Dies kann als Konversion der Deployment Deskriptoren des alten Servers verstanden werden. Diese Konversion kann automatisiert geschehen. So wird für Glassfish ein Tool angeboten, dass Java-EE-Anwendungen von bestimmten Servern so verändert, das sie auf dem Glassfish lauffähig werden (vgl. [mig08]). Dabei werden auch einige Server-spezifische und nicht standardkonforme Einstellungen so verändert, dass sie standardkonform werden.

Schwieriger stellt sich die Migration von einem Server zu einem anderen dar, wenn die Java-EE-Anwendung Server-spezifische Erweiterungen benutzt, die nicht dem Standard entsprechen.

Wenn es eine standardkonforme Alternative zu der Server-spezifischen Erweiterung gibt, kann die bestehende Anwendung so verändert werden, dass sie dem Standard entspricht. Während dieser Veränderung kann sie weiter auf dem alten Server deployed werden. Ist die Anwendung standardkonform gemacht worden, kann sie, von Hand oder mit Hilfe eines Werkzeugs, auf dem neuen Server eingesetzt werden.

Gibt es keine standardkonforme Alternative, aber auf dem Zielserver eine Server-spezifische Erweiterung mit äquivalenter Funktion, ist eine Big-Bang-Umstellung möglich.

Die Erweiterung des alten Servers wird in der Anwendung nicht mehr genutzt, sondern die ähnliche Erweiterung im neuen Server. Während der Umstellung wäre die Anwendung aller-

dings nicht mehr deploybar und die EJB somit bis zur Vollendung der Migration untestbar. Dies erhöht das Risiko der Migration.

Anhand der Beispielmigration in Kapitel 4.4 kann vermutet werden, dass die meisten Probleme bei einer Migration von einem Server zu einem anderen auftreten, weil die Anwendung nicht standardkonform ist und der Server von dem migriert werden soll dies zulässt.

3.7.2 Migration von EJB 1.1 zu EJB 2.0

Es existiert ein Migrationsleitfaden von Sun, in dem ein Migrationspfad von EJB 1.1 zu EJB 2.0 für Entity Beans beschrieben wird. Im Laufe der in Kapitel 4.4 dargestellten Migration traten einige Probleme auf, die Schwächen im Migrationsleitfaden aufzeigten.

Laut dem Leitfaden kann eine Entity Bean nicht von Version 1.1 auf Version 2.0 migriert werden, wenn einer der folgenden vier Fälle vorliegt (vgl. [Sun07]):

1. Wenn eine Entity Bean eine Klasse als Primärschlüssel benutzt, dürfen die Namen der Schlüsselfelder nicht mit einem Großbuchstaben beginnen.
2. Wenn es eine CMP-Variable vom Typ T gibt, deren Name `name` ist und es bereits eine Methode gibt, deren Signatur `<T> getName()` oder `void setName(<T>)` ist.
3. Wenn eine Finder-Methode die SQL-Konstrukte `Join`, `Outer Join` oder `Order by` benutzt.
4. Wenn die Entity Bean ein persistentes Feld vom Typ `java.util.Enumeration` hat.

Sind alle Voraussetzungen erfüllt, kann laut dem Leitfaden die Migration in folgenden Schritten durchgeführt werden:

1. Die Implementationsklasse der Entity Bean muss abstrakt werden.
2. Die Getter und Setter für CMP-Variablen müssen abstrakt werden.
3. Für jede CMP-Variable überprüfen, ob es für sie jeweils einen Getter und Setter gibt. Wenn nicht, müssen sie als abstrakte Methoden hinzugefügt werden.
4. Alle alten CMP-Variablen aus dem Code auskommentieren.
5. Für jede CMP-Variable eine `protected` Exemplarvariable mit dem entsprechenden Namen deklarieren.
6. In allen Methoden namens `ejbCreate()` den Zugriff auf CMP-Variablen durch einen Zugriff auf den entsprechenden Getter und Setter ersetzen.

   Sollte in einer dieser Methoden der Zugriff auf eine Variable oder Methode per Referenz auf `this` erfolgen, muss dies entfernt werden.

7. In den Methoden names `ejbPostCreate()` die `protected` Exemplarvariablen initialisieren, indem ihnen der vom entsprechenden Getter zurückgegebene Wert zugewiesen wird.

8. In den Methoden namens `ejbLoad()` den `protected` Exemplarvariablen den vom entsprechenden Getter zurückgegebenen Wert zuweisen.

9. In der Methode `ejbStore()` den Wert der Exemplarvariablen mittels des entsprechenden Setters in die Datenbank schreiben.

10. Alle Vorkommnisse einer CMP-Variable durch die entsprechende `protected` Exemplarvariable ersetzen.

Wie die Beispielmigration zeigte, ist dieser Leitfaden fehlerhaft. Die Erstellung von Exemplarvariablen, mit einem Namen, der identisch mit dem einer CMP- oder CMR-Variable ist, verstößt gegen den J2EE-Standard (vgl. [Sha03]).

Es kann aber statt, wie in Schritt 10 vorgeschlagen, den Zugriff auf CMP-Variablen durch den Zugriff auf die entsprechende `protected` Exemplarvariable zu ersetzen, der Zugriff auf die CMP-Variable durch einen Zugriff auf den jeweiligen Getter bzw. Setter ersetzt werden. Dies macht die Schritte 5 und 7–10 überflüssig.

Außerdem schlägt Sun im Migrationsleitfaden vor, den Deployment Deskriptor wie folgt zu ändern (vgl. [Sun07]):

1. Alle Namen von CMP-Variablen müssen kleingeschrieben sein.

2. Das Tag `<abstract-schema-name>` hinter dem Tag `<reentrant>` einfügen. Der Name des abstrakten Schemas ist der Name Entity Bean mit dem Präfix `ias_`.

3. Hinter dem Tag `<primkey-field>` Folgendes einfügen:

```
<security-identity>
        <use-caller-identity/>
</security-identity>
```

4. Der SQL-Code in der Finder-Methode muss nach EJB QL übersetzt werden.

5. Das `<query>`-Tag wird hinter dem `<security-identity>`-Tag eingefügt.

Damit ist die Entity Bean von Version 1.1 auf Version 2.0 migriert. Da das Component Interface und das Home Interface dafür nicht geändert werden mussten, funktionieren die Clients weiterhin.

Migraton von Session Beans 1.x oder 2.x auf Version 3.0

In [PRL07] werden folgende Schritte vorgeschlagen, um diese Migration durchzuführen:

1. Aus der Implementationsklasse ein POJO machen, indem der EJB-spezifische Code entfernt wird.
2. Mindestens ein Business-Interface definieren.
3. Die Einstellungen im Deployment Deskriptor durch Annotationen ersetzen.
4. JNDI-Lookups durch Dependency Injection ersetzen.

Bezüglich des letzten Schritts ist zu beachten, dass die Clients der Session Bean auch an die neuen Gegebenheiten angepasst werden müssen. Exemplare der alten Session Beans wurden durch den Aufruf einer Methode des Home Interface der Session Beans erzeugt. Die Referenz auf das Home Interface wiederum wurde durch einen JNDI-Lookup erhalten.

Im Falle von Stateless Session Beans in Container-verwalteten Klassen kann der JNDI-Lookup durch DI in eine Exemplarvariable ersetzt werden. In Klassen, die nicht Container-verwaltet sind, ist dies vom Standard nicht gefordert; allerdings dürfen Applikationsserver dies unterstützen (vgl. [EJB06]).

Referenzen auf Stateful Session Beans sollten generell nicht per DI sondern per JNDI-Lookup bezogen werden. Dies liegt darin begründet, dass DI direkt nach dem Aufruf des Konstruktors stattfindet. Daher ist DI von Stateful Session Beans in nicht client-spezifische Objekte nicht sinnvoll, da eine Stateful Session Bean immer nur von einem Benutzer genutzt werden sollte (vgl. [PRL07]).

3.7.3 Migration von Entity Beans zu JPA Entities

In Anlehnung an [PRL07] können Entity Beans in den folgenden Schritten zu einem JPA-Entity migriert werden:

1. Home- und Business-Interface können entfernt werden.
2. Den im Deployment Deskriptor angegebenen Primärschlüssel in der Klasse mit `@Id` annotieren.
3. Die Einstellungen für das O/R-Mapping aus dem Applikationsserver-spezifischen Deployment Deskriptor in die Annotationen `@Table` und `@Column` umsetzen.
4. Die Callback-Methoden entsprechend annotieren bzw. entfernen, wenn sie leer sind.
5. Wenn in den Entity Beans JNDI-Lookups genutzt werden, um andere Entity Beans zu finden, sollte versucht werden, den JNDI-Lookup außerhalb des JPA Entities zu verlegen.

Der JNDI-Lookup kann auch durch den Zugriff auf einen Entity Manager ersetzt werden, um ein entsprechendes JPA Entity zu finden, allerdings ist das Entity dann nicht außerhalb eines Containers testbar.

6. Die Finder-Methoden, die im Home-Interface und dem Deployment Deskriptor definiert sind, in Annotationen in der Klasse umsetzen. Dabei muss statt EJB QL JPQL genutzt werden.

7. Die CMR für die Entity Beans aus dem Deployment Deskriptor in Annotationen umsetzen.

8. Transaktions- und Sicherheitseinstellungen von den Entity Beans in die Session Beans verlagern.

9. Die Clients umstellen.

Um die Implementationsklasse einer Entity Bean 2.0 zu einem Entity zu machen, müssen die CMP-Variablen, auf die mit abstrakten Gettern und Settern zugegriffen wurde zu Exemplarvariablen gemacht werden.

Außerdem implementiert ein Entity nicht mehr das Interface `EntityBean`. Es muss daher auch nicht die in diesem Interface deklarierten Methoden implementieren. Benötigte Callback-Methoden werden, analog zu den Session Beans, mit Annotationen markiert.

Die Zuordnung des Entity zu einer Tabelle in der Datenbank und der Exemplarvariablen zu Spalten der Tabelle erfolgt durch Convention over Configuration.

Der Name der Tabelle entspricht dem Namen der Klasse und die Namen der Spalten entsprechen denen der Exemplarvariablen. Sollen die Namen abweichen ist dies durch Annotation der Klasse bzw. der Exemplarvariablen möglich.

Die Migration ist durch diesen Schritt noch nicht abgeschlossen, da auch die Clients der Entity Bean migriert werden müssen. Statt eines Aufrufs am Home Interface der Entity Bean, wird ein Entity, wie ein gewöhnliches Java-Objekt, durch den Aufruf des `new`-Operator erzeugt und durch den Konstruktor initialisiert. Soll es persistiert werden, geschieht dies durch den Aufruf der Methode `persist` am Entity Manager, der für gewöhnlich per DI zugänglich gemacht werden kann.

Statt Entity Beans über das Home Interface zu suchen, können sie über den Entity Manager gefunden werden.

3.8 Kapitelzusammenfassung

In diesem Kapitel wurden die grundlegenden Technologien von Java EE sowie die Schichtenarchitektur vorgestellt. Dabei wurde besonders auf Entity Beans und Java Persistence Entities eingegangen. Außerdem wurden die wesentlichen für Java EE 6 erwarteten Änderungen kurz vorgestellt. Es wurden Leitfäden zur Migration von Entity Beans 1.1 zu Entity Beans 2.0 und

von dort zu Entities sowie zur Migration von Session Beans 1.x bzw. 2.x zu Session Beans 3.0 vorgestellt.

4 Eine Beispielmigration

Als Beispiel soll eine Webanwendung, die am Institut für Logistik und Transport der Universität Hamburg in der Lehre benutzt wird, migriert werden. Diese Webanwendung ist eine verteilte Version des *Beer Distribution Game*, das in den 1960ern als Brettspiel am MIT entstanden ist (vgl. [LSL02]).

Sie dient dazu, den teilnehmenden Spielern den *Bullwhip-Effekt* zu demonstrieren.

4.1 Das Beer Distribution Game

Beim *Beer Distribution Game* (kurz BDG) handelt es sich um ein Spiel für vier Spieler, von denen jeder einen Bestandteil einer Lieferkette spielt. Die Nachfrage des vom Computer gesteuerten Endkunden wird dabei vom Ersteller des Spiels vor Beginn festgelegt.

Die Lieferkette ist in Abbildung 5 dargestellt. Die Pfeile stellen jeweils Bestellung, Lieferungen bzw. Produktion dar. Die Zahlen an den Pfeilen geben die Dauer in Runden an, die die jeweilige Aktion benötigt.

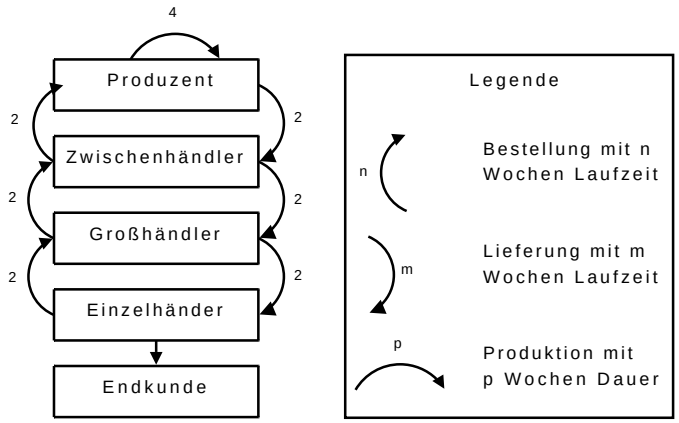

Abbildung 5: Lieferkette

Die Spieler bestellen jede Runde Bier mit dem Ziel, einerseits die Nachfrage durch das nachgelagerte Mitglied der Lieferkette zu befriedigen, andererseits aber auch die Lagerkosten möglichst gering zu halten. Nicht befriedigte Nachfrage und Lagerbestand verursachen jeweils Kosten, deren Summe es zu minimieren gilt.

Haben alle Spieler innerhalb einer Runde ihre Bestellung getätigt, beginnt die nächste Runde. Die einzige Information, die die Mitglieder der Lieferkette erhalten, sind die Bestellungen, die bei ihnen eingehen.

Für gewöhnlich wird der Bedarf des Endkunden so eingestellt, dass sich die Nachfrage zu Beginn des Spiels verdoppelt und dann konstant bleibt. Auch diese sehr einfache Situation führt bereits zum Bullwhip-Effekt.

Dieser äußert sich darin, dass sich Nachfrageschwankungen in Richtung des Produzenten immer stärker aufschaukeln (vgl. [For58]).

Dies ist im Falle des BDG auf die begrenzte Information der Spieler und die zeitliche Verzögerung zwischen Aufgabe der Bestellung und Ankunft der Lieferung (Vorlaufzeit) zurückzuführen.

Der Einzelhändler wird allerdings nicht nur die Nachfrageschwankung dieser Woche ausgleichen, sondern muss, wegen der Vorlaufzeit von vier Wochen, mit der ersten Bestellung auch die zu erwartenden Nachfrageschwankungen der nächsten vier Wochen ausgleichen. Außerdem wird er den Sicherheitsbestand seines Lagers an die neue Nachfragesituation anpassen wollen. Insgesamt wird seine Nachfrage gegenüber dem Großhändler stärker schwanken als die Nachfrage des Endkunden. Die gleiche Überlegung ist auch auf die Nachfrage des Großhändlers gegenüber dem Zwischenhändler sowie dessen Nachfrage gegenüber dem Produzenten anwendbar. Die Nachfrageschwankung schaukelt sich also in Richtung des Produzenten auf.

In realen Lieferketten gibt es noch andere Gründe für das Auftreten des Bullwhip-Effekts. Diese werden z. B. in [LPW97] dargestellt.

Um den Spielern zu zeigen, wie diesem Effekt begegnet werden kann, wird für gewöhnlich noch ein zweites Spiel gespielt, wobei den einzelnen Spielern aber mehr Informationen zur Verfügung stehen.

4.2 Die Beispielanwendung

Die Webanwendung basiert auf J2EE 1.2. Der eingesetzte Server ist der *Orion Application Server*[1] (kurz Orion). Die Entwicklung der Server-Software wurde nach Erscheinen der Version 2.07, die nur J2EE Versionen bis 1.3 unterstützt, eingestellt.

Die Software soll im Rahmen von Lehrveranstaltungen mit der aktuellen Version von Java EE weiterentwickelt werden. Daher soll ein anderer Server, der Java EE unterstützt, eingesetzt werden. Die Wahl fiel dabei auf Glassfish, da er die Referenzimplementation des Java EE Standards darstellt, als Open Source Software erhältlich und kostenfrei ist.

Die Vorgaben für die Migration sind, dass JPA Entities statt Entity Beans verwandt werden sollen.

Außerdem sollen die Session Beans mit Annotationen statt Deployment Deskriptoren und die Präsentationsschicht mit Java Server Faces realisiert werden.

Diese Ziele setzen die Migration der Anwendung von J2EE 1.2 auf Java EE 5, sowie eine Migration von Orion auf einen Java EE 5 fähigen Server, in diesem Fall Glassfish, voraus.

[1] `http://www.orionserver.com`

Die Webanwendung nutzt eine Orion-spezifische Implementation von CMR, um Beziehungen zwischen Entity Beans der Version 1.1 zu realisieren und kann daher nicht einfach auf Glassfish deployed werden. CMR wurden erst mit EJB 2.0 in den Standard aufgenommen und funktionieren (auf Glassfish) nicht mit EJB 1.1.

Die Webanwendung besteht aus einem Web-Frontend, das mit JSP realisiert ist. Diese JSPs greifen bei Benutzereingaben auf Servlets zu und übergeben mehrere Strings, die die auszuführende Aktion enthalten. Dabei enthält ein String den Namen eines Kommandos im Sinne des Entwurfsmusters *Kommando* aus [GHJV96]. Die Servlets führen jeweils die Kommandos aus.

Für verschiedene Bereiche der Anwendung existiert jeweils ein eigenes Servlet mit eigenen Kommandos. Die Kommandos greifen dann auf *Session Beans* zu, die wiederum die eigentliche Spiellogik enthalten. Eine Auflistung der drei genutzten Session Beans findet sich in Tabellle 2.

NAME DER ENTITY BEAN	FACHLICHE BEDEUTUNG
GameManagerEJB	Übernimmt die Verwaltung von Spielen vor deren Start (Anlegen, Löschen, Starten)
GameSessionEJB	Führt alle vom Spieler vorgegebenen Spielhandlungen durch
NameSuggestorEJB	Schlägt einem Benutzer einen Namen vor, falls der gewünschte Name schon registriert ist

Tabelle 2: Alle im BDG genutzten Session Beans

Die für die Ausführung der Logik notwendigen Daten werden aus Entity Beans geholt und die Ergebnisse auch in solchen gespeichert. Die fünf zu diesem Zweck im Spiel enthaltenen Entity Beans sind Tabelle 3 entnehmbar.

NAME DER ENTITY BEAN	FACHLICHE BEDEUTUNG
DeliveryEJB	Stellt eine Lieferung dar
GameEJB	Stellt ein Spiel dar
OrderEJB	Stellt eine Bestellung dar
RoleEJB	Stellt die Rolle dar, die ein Spieler übernimmt
UserEJB	Stellt einen Benutzer des Spiels dar

Tabelle 3: Alle im BDG genutzten Entity Beans

Die Registrierung eines neuen Benutzers funktioniert über eine JSP. Das an das Servlet übergebene Kommando legt eine neue `UserEJB` an, deren Exemplarvariablen in der relationalen Datenbank gespeichert werden.

Der Applikationsserver greift beim Login auf diese Entity Bean zu und authentifiziert anhand der in ihr gespeicherten Daten den Benutzer. Diese Art der Authentifizierung ist Orion-spezifisch und somit nicht auf Glassfish nutzbar.

Aus diesen Gründen müssen, bevor die Anwendung überhaupt auf Glassfish deployed werden kann, die Entity Beans mindestens dem EJB-Standard 2.0 entsprechen.

In Abbildung 6 ist oben links die beschriebene Ausganssituation zu sehen. Unten links ist das Ziel der Migration dargestellt. Die durchgezogenen Linien stellen einen iterativen Migrationsansatz dar, die gestrichelte Linie einen Big-Bang-Ansatz. Beide Ansätze werden im folgenden Abschnitt vorgestellt.

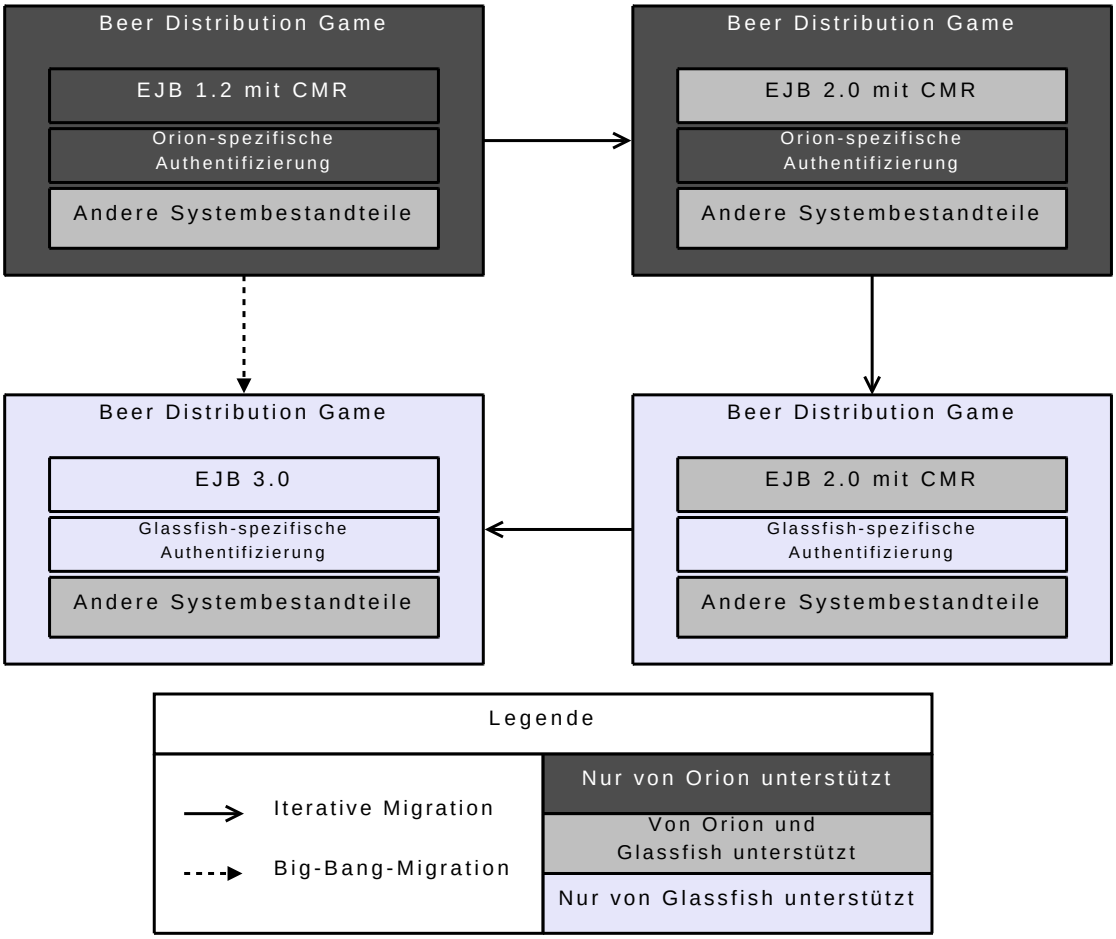

Abbildung 6: Migrationspfad

4.3 Vorgehen bei der Migration

Es existiert für die gesamte Anwendung kein Test. Unabhängig vom Migrationsansatz sollten, um die Korrektheit der Anwendung nach der Migration sicherstellen zu können, als erstes Regressionstests erstellt werden. Für die darauf folgende Migration bieten sich drei Ansätze an:

4.3.1 Ansatz 1 – Inkrementelle Migration

Es kann eine Migration auf EJB 2.0 auf dem Orion Server durchgeführt werden, da diese zu J2EE 1.3 gehören. Dies hat den Vorteil, dass die Software lauffähig bleibt und die Entity Beans eine nach der anderen migriert werden können. Nachdem alle Entity Beans dem EJB 2.0 Standard genügen, sollte die Anwendung auf dem Glassfish deployable sein. Allerdings könnte sich niemand einloggen, da der von der Anwendung genutzte User Manager für Glassfish nicht existiert.

Eine Anpassung der Anwendung an den Authentifizierungsmechanismus von Glassfish sollte mit geringem Aufwand möglich sein. Danach sollte die Anwendung vollständig und korrekt auf Glassfish laufen.

Im nächsten Schritt können dann die Session Beans Annotationen erhalten, die die Deployment Deskriptoren ersetzen, und schließlich die Entity Beans durch JPA Entities ersetzt werden.

Die Strategie für diese Migration wäre also eine Konversion kombiniert mit einem iterativen Migrationsprozess.

Dieser Migrationsansatz beinhaltet folgende Schritte:

1. Migration der Entity Beans von EJB 1.1 auf EJB 2.0
2. Migration von Orion auf Glassfish
3. Austausch des Authentifizierungsmechanismus
4. Migration der Session Beans auf EJB 3.0
5. Migration der Entity Beans 2.0 zu JPA Entities

4.3.2 Ansatz 2 – Big-Bang-artige Migration mit Konversion

Die Migration kann auch direkt von EJB 1.1 auf EJB 3.0 führen und die Anwendung somit auf Glassfish deployable machen. Daraufhin muss der Authentifizierungsmechanismus angepasst werden.

Die migrierte Anwendung ist erst am Ende der Migration wieder lauffähig. Hier ist die Migrationsstrategie ebenfalls eine Konversion, allerdings mit einem Big-Bang-Prozess.

Dieser Migrationsansatz würde aus folgenden Schritten bestehen:

1. Migration der Entity Beans 1.1 zu JPA Entities

2. Migration der Session Beans auf EJB 3.0
3. Migration von Orion auf Glassfish
4. Austausch des Authentifizierungsmechanismus

4.3.3 Ansatz 3 – Big-Bang-artige Migration mit Neuentwicklung

Eine ähnliche Migration wurde parallel zum Entstehen dieser Diplomarbeit innerhalb eines Projekts durchgeführt. Der Projektbericht der Gruppe war vor dem Druck dieser Arbeit noch nicht verfügbar, so dass ich mich im Folgenden auf einen E-Mail-Austausch mit einem Mitglied dieser Gruppe, Lukas Ciostek, beziehe (vgl. [Cio08]).

Diese Gruppe hat einen Big-Bang-Ansatz mit Neuentwicklung als Migrationsstrategie kombiniert.

Die Gruppe um Lukas Ciostek hat die gesamte Anwendung, bis auf das Datenbankschema und die Benutzungsoberfläche, neu erstellt, statt den alten Code wiederzuverwenden. Dabei wurde die Architektur des BDG stark verändert.

Insgesamt wurde mehr Zeit für die Migration benötigt, aber dabei auch mehr geändert. So wurde der Code von J2SE 1.3 auf Java SE 5 migriert, was durch den Einsatz von Generizität vor allem der statischen Typsicherheit zu Gute kam. Außerdem wurde die Benutzungsschnittstelle mit JSF realisiert, was für die in dieser Arbeit dargestellten Beispielmigration auch geplant war, aber wegen Zeitmangels nicht mehr durchgeführt werden konnte. Zusätzlich wurde auch das Design der Benutzungsschnittstelle überarbeitet.

4.3.4 Auswahl des Migrationsansatzes

Der erste Weg scheint der bessere zu sein, da dort nach der Änderung jeder EJB getestet werden kann und so Fehler relativ schnell auffindbar und behebbar sind. Dies entspricht der Idee des Chicken-Little-Ansatz oder auch den kleinen Schritt im Sinne von Kent Beck (vgl. [Bec04]). Auf Gateways kann aus den auf Seite 40 genannten Gründen verzichtet werden.

Bei Nutzung eines Big-Bang-Ansatz wäre die Anwendung erst nach Beendigung der Migration wieder testbar, was dazu führen würde, dass sämtliche Fehler auf einmal geklärt werden müssten.

Dies ist bei der Migration von EJB 1.1 auf EJB 2.0 nicht von großer Bedeutung, da an der Anwendung nur wenig verändert werden muss. Dafür ist es von umso größerer Bedeutung bei der Umstellung von EJB 2.0 auf EJB 3.0, da vor allem für die Umstellung auf Entities viel an der Anwendung geändert werden muss:

Die Business und Home Interfaces fallen weg. Sie werden aber von anderen Teilen der Anwendung benutzt. Außerdem müssen Entities explizit persistiert werden, während dies bei Entity Beans implizit geschieht. Es ist auch zu beachten, dass vom Container verwaltete Beziehungen zwischen Entity Beans automatisch auf beiden Seiten aktualisiert werden, sobald sich eine Seite ändert. Dies muss bei JPA Entities ebenfalls von Hand geschehen.

Dieser Ansatz verursacht allerdings auf den ersten Blick mehr Arbeit als die Alternative, bei der direkt von EJB 1.1 auf EJB 3.0 umgestellt wird, ohne den Umweg über EJB 2.0 zu gehen. Dies gilt aber nur, wenn die Migration problemlos gelingt, wovon erfahrungsgemäß nicht auszugehen ist.

Für die jeweiligen Teilmigrationen sollen die auf Seite 40ff. dargestellten Leitfäden genutzt werden.

Die Umstellung auf JSF lässt sich im Anschluss an die beschriebene Migration durchführen. Allerdings war die Migration auf JSF aus zeitlichen Gründen nicht mehr machbar.

4.4 Durchführung der Beispielmigration

4.4.1 Erstellung der Tests

Vor Beginn der Migration mussten Tests geschrieben werden, da für die gesamte Anwendung kein Test vorhanden. Die Tests dienen dazu, den Erfolg der Migration überprüfen zu können.

Für die *Unit Tests* wurde *JUnitEE*[2] genutzt, da sich EJBs, wie auf S. 32 dargestellt, nicht ohne weiteres außerhalb eines Containers testen lassen. JUnitEE besteht aus einem Servlet, dass die mit *JUnit*[3] definierten Tests auf dem Applikationsserver ausführt. Für diese Art des Testens spricht vor allem die Identität zwischen Test- und Produktivumgebung.

Funktionstests der Software wurden mit *HttpUnit*[4] durchgeführt. Dabei handelt es sich um eine Bibliothek, die einen Webbrowser nachahmt. So kann aus einem Java-Programm heraus mit verhältnismäßig geringem Aufwand eine Webanwendung aus Benutzersicht getestet werden. Auch hier kam wieder JUnit zum Einsatz, um die erhaltenen Ergebnisse mit den erwarteten zu vergleichen.

Die für das Altsystem geschriebenen Tests sind nicht vollständig, allerdings sollten sie ausreichen um die meisten Fehler automatisiert zu finden. Die Erstellung der Tests hat ca. 40 Stunden in Anspruch genommen. Daran hat die Zeit, die zum Erlernen der Benutzung der beiden Frameworks benötigt wurde und sie auf den Applikationsservern lauffähig zu machen, ca. 20 Stunden Anteil.

Die Testabdeckung durch die Unit Tests ist vermutlich relativ gering, da nur die EJBs getestet wurden. Die meisten Methoden der Entity Beans sind jedoch so einfach, dass sie kaum einen Defekt haben können. Diese Methoden zu testen ist nicht sinnvoll.

4.4.2 Migration von EJB 1.1 zu EJB 2.0

In Listing 1 ist der vollständige Code der `DeliveryEJB` aus dem BDG abgedruckt. Die Formatierung wurde für diese Arbeit so geändert, dass die Darstellung der gesamten Klasse möglichst

[2] `http://www.junitee.org/`
[3] `http://www.junit.org/`
[4] `http://www.httpunit.org/`

wenige Zeilen benötigt. Anhand dieser EJB wird in diesem Abschnitt die Migration der Entity Beans erläutert.

Der Plan sieht vor, die Migration anhand des in Kapitel 3.7.2 dargestellten modifizierten Leitfadens vorzunehmen. Die Überprüfung des BDG nach dem Leitfaden ergab, dass keine Exemplarvariable einer Primärschlüsselklasse mit einem Großbuchstaben begann oder eine persistente Exemplarvariable vom Typ `java.util.Enumeration` hatte. Der Fall, dass auf eine CMR- oder CMP-Variable per gleichnamigem Getter oder Setter zugegriffen wurde trat auf. Eine Migration ist allerdings möglich, wenn in der Methode nichts weiter getan wird, als den Wert der Variable zurückzugeben. Dies trifft auf CMR- und CMP-Variablen der `DeliveryEJB` zu (vgl. Listing 1.

Soll die Methode mehr tun, ist es möglich, die Exemplarvariable umzubenennen, die Methoden beizubehalten und sie mit den Gettern und Settern der CMP-Variable arbeiten zu lassen. Dieser Fall trat zum Beispiel beim Setzen des Passworts in der `UserEJB` auf, da es im Klartext übergeben wird und erst in der Methode `setPassword` gehasht wird.

In einer Finder-Methode trat ein `Order by` auf. Dieses wurde entfernt und wäre in der Migration zu EJB 3.0 wieder hinzugefügt worden. Allerdings stellte sich heraus, dass es überflüssig war.

Wie sich die `DeliveryEJB` durch die Migration zu EJB 2.0 verändert hat, ist in Listing 2 zu sehen. Wie im Vergleich zwischen den beiden Versionen der `DeliveryEJB` zu sehen ist, hat sich der Zugriff auf die CMP- und CMR-Variablen geändert. Statt direkt auf Exemplarvariablen zuzugreifen, erfolgt der Zugriff in EJB 2.0 über abstrakte Getter und Setter, die vom Container implementiert werden.

Die Änderungen im Deployment Deskriptor nach dem in Kapitel 3.7.2 dargestellten Leitfaden konnte problemlos durchgeführt werden. Allerdings wurde statt das Präfix `ias_` für den Namen des abstrakten Schemas zu verwenden, der Name der EJB mit dem Suffix `Bean` verwandt.

Im Folgenden werden Probleme, die während der Migration auftraten, und deren Lösung nicht in den Migrationsleitfäden beschrieben ist, beschrieben.

CMR und EJB 2.0

Wie sich während der Migration herausstellte, unterstützt Orion zwar CMR für EJB 1.1, aber nicht CMR über die Remote Schnittstelle von EJB 2.0. Dies machte es nötig, lokale Schnittstellen für die Entity Beans zu schaffen.

Da Entities nur noch lokale Schnittstellen haben, wurden in diesem Schritt gleichzeitig auch die Remote Schnittstellen entfernt. Dies führte auch dazu, dass die Methoden der Entity Beans keine `Remote Exceptions` mehr werfen müssen und die entsprechenden try-catch-Blöcke entfernt werden konnten.

```
1  package beergame.ejb.dbo;
2  import javax.ejb.CreateException;
3  import javax.ejb.EntityBean;
4  import javax.ejb.EntityContext;
5  import beergame.ejb.util.SequenceGenerator;
6  public class DeliveryEJB implements EntityBean {
7          // persistente Attribute
8          public Long deliveryNumber;
9          public String role;
10         public Game game;
11         public int round;
12         public int amount;
13         // EJB-Methoden zum Erzeugen
14         public Long ejbCreate(String role, Game game, int round, int amount)
15                         throws CreateException {
16                 if (game != null && amount > 0) {
17                         this.deliveryNumber = new Long(SequenceGenerator.getKey());
18                         this.role = role;
19                         this.game = game;
20                         this.round = round;
21                         this.amount = amount;
22                 } else {
23                         throw new CreateException();
24                 }
25                 return null;
26         }
27         public Long ejbCreate(Game game, int round, int amount)
28                         throws CreateException {
29                 return ejbCreate(null, game, round, amount);
30         }
31         public void ejbPostCreate(String role, Game game, int round, int amount) {}
32         public void ejbPostCreate(Game game, int round, int amount) {}
33         // Business Methoden
34         // Gibt die Liefermenge zurueck.
35         public int getAmount() {
36                 return amount;
37         }
38         // Gibt das Spiel zurueck.
39         public Game getGame() {
40                 return game;
41         }
42         // Gibt die Runde zurueck, in der diese Lieferung ausgefuehrt
43         public int getRound() {
44                 return round;
45         }
46         // Gibt die Rolle, die diese Lieferung ausgefuehrt hat, zurueck.
47         public String getRole() {
48                 return role;
49         }
50         // Callback-Methoden
51         public void ejbActivate() {}
52         public void ejbPassivate() {}
53         public void ejbLoad() {}
54         public void ejbStore() {}
55         public void ejbRemove() {}
56         public void setEntityContext(EntityContext context) {}
57         public void unsetEntityContext() {}
58 }
```

Listing 1: Die Implementationsklasse der `DeliveryEJB` als EJB 1.1

```
1  package beergame.ejb.dbo;
2  import javax.ejb.*;
3  import javax.rmi.*;
4  import java.util.*;
5  import beergame.ejb.util.*;
6  public abstract class DeliveryEJB implements  EntityBean {
7          // backend Methoden
8          public Long ejbCreate(String role, Game game, int round, int amount)
9                          throws CreateException {
10                 Long deliveryNumber = SequenceGenerator.getKey();
11                 if (game != null && amount > 0) {
12                         setDeliveryNumber(deliveryNumber);
13                         setRole(role);
14                         setRound(round);
15                         setAmount(amount);
16                 } else {
17                         throw new CreateException();
18                 }
19                 return deliveryNumber;
20         }
21         public Long ejbCreate(Game game, int round, int amount)
22                         throws CreateException {
23                 return ejbCreate(null, game, round, amount);
24         }
25         public void ejbPostCreate(String role, Game game, int round, int amount) {
26                 setGame(game); // CMR-Variablen dürfen nicht in EJBCreate gesetzt werden.
27         }
28         public void ejbPostCreate(Game game, int round, int amount) {
29                 setGame(game); // CMR-Variablen dürfen nicht in EJBCreate gesetzt werden.
30         }
31         // Business Methoden
32         public abstract int getAmount();   // Gibt die Liefermenge zurueck.
33         public abstract Game getGame();    // Gibt das Spiel zurueck.
34         public abstract int getRound();    // Gibt die Runde zurueck, in der diese Lieferung ausgefuehrt
35         public abstract String getRole(); // Gibt die Rolle, die diese Lieferung ausgefuehrt hat, zurueck.
36         // Methoden für CMR-Variablen
37         public abstract Long getDeliveryNumber();
38         public abstract void setDeliveryNumber(Long deliveryNumber);
39         public abstract void setRole(String role);
40         public abstract void setGame(Game game);
41         public abstract void setRound(int round);
42         public abstract void setAmount(int amount);
43         // EJB-Callback-Methoden
44         public void ejbActivate() {}
45         public void ejbPassivate() {}
46         public void ejbLoad() {}
47         public void ejbStore() {}
48         public void ejbRemove() {}
49         public void setEntityContext(EntityContext context) {}
50         public void unsetEntityContext() {}
51 }
```

Listing 2: Die Implementationsklasse der DeliveryEJB als EJB 2.0

User Manager

Durch die Umstellung auf lokale Schnittstellen trat allerdings das Problem auf, dass der von Orion zur Verfügung gestellte `EJBUserManager` nicht mehr funktionierte. Dabei handelt es sich um eine Klasse, die zur Laufzeit die Authentifizierung der Benutzer übernimmt. Der Benutzername und das Passwort sowie die Gruppenzugehörigkeit werden in der Entity Bean `UserEJB` gespeichert, auf die der Usermanager zugreift.

Dazu muss ein von Orion zur Verfügung gestelltes Interface implementiert werden, das gleichzeitig auch von `EJBObject` erbt. Eine Klasse, die `EJBObject` implementiert, ist ein Remote Home Interface einer EJB. Aufgrund des CMR sollten aber alle Entity Beans nur noch Local Interfaces haben. Entity Beans können zwar gleichzeitig ein Remote und ein Local Interface haben, allerdings funktioniert dann der `EJBUserManager` nicht mehr.

Als Lösung wurde der `EJBUserManager` durch den `DataSourceUserManager` ersetzt. Dieser nutzt als Informationsquelle nicht die Entity Bean, sondern eine Datenbank. Da der Zustand der Entity Beans in der Datenbank gespeichert wird, waren die Benutzernamen, Hashes der Passwörter und Gruppenzugehörigkeit dort bereits gespeichert. Es ist also insbesondere nicht nötig, ein bestimmtes Interface zu implementieren bzw. von einer bestimmten Klasse zu erben.

Zeitaufwand und Gründe für Probleme

Insgesamt hat dieser Teil der Migration ca. 60 Stunden Arbeitszeit in Anspruch genommen. Die Migration wäre schneller möglich gewesen, wenn Orion und das BDG besser dokumentiert gewesen wären. Das BDG wurde im Vorfeld der Migration bereits erkundet und insbesondere durch das Anfertigen der Tests verstanden. Einige Details, die für die Migration relevant waren, wurden dabei allerdings übersehen.

Vor dem Beginn der Diplomarbeit war keine Erfahrung mit Orion vorhanden, so dass auch das Erlernen des Umgangs mit Orion teilweise während der Migration geschah.

4.4.3 Migration von Orion zu Glassfish

Der Plan zur Migration von Orion zu Glassfish sah vor, dass das Ersetzen der Applikationsserverspezifischen Deployment Deskriptoren und des User Managers ausreicht.

Dies war nicht der Fall, was vor allem daran lag, dass Orion, im Gegensatz zu Glassfish, auch Programme ausführt, die nicht dem J2EE-Standard entsprechen.

CMR-Variablen

Orion lässt es zu, dass der Wert von CMR-Variablen in der Methode `ejbCreate` gesetzt werden. Dies ist laut den Spezifikationen nicht erlaubt; die Werte können aber in der Methode `ejbPostCreate` gesetzt werden (vgl. [Sun03]). In Listing 1 in Zeile 19 wird der Wert der CMR-Variable `game` durch die Anweisung `this.game = game` gesetzt. Wird dies im

Zuge der Migration durch `setGame(game)` ersetzt, führt dies zu Laufzeitfehlern. Richtig ist die in Listing 2, Zeile 27 und 30 zu sehende Lösung `setGame(game)` in der Methode `ejbPostCreate` aufzurufen.

Collections als CMR-Variablen

Entity Beans können CMR zu mehreren Exemplaren einer anderen Klasse von Entity Beans haben. Dies kann über eine CMR-Variable vom Typ `java.util.Collection`, die mit dem durch das Component Interface der Entity Bean definierten Typ parametrisiert wird, geschehen.

Der Zugriff auf eine `Collection` darf nur in der gleichen Transaktion geschehen, in der sie auch mit dem entsprechenden Getter geholt wurde. Eine Zwischenspeicherung in einer Variable, wie Sun sie im Migrationsleitfaden vorschlägt, kann also in diesem Fall nicht funktionieren kann, da eine Transaktion nur für die Dauer eines Methodenaufrufs einer Session Bean erhalten bleibt (vgl. [Sun01]).

Es muss daher jedes Mal die `Collection` über den vom Container implementierten Getter geholt werden. Sie darf danach nicht über den Setter gespeichert werden – dies führt zu Fehlern. Die in die `Collection` eingefügten Elemente werden auch ohne den Aufruf des Setters gespeichert.

JNDI-Lookups

Die JNDI-Lookups unterscheiden sich ebenfalls zwischen Orion und Glassfish. In Orion konnten die JNDI-Lookups im Konstruktor der Servlets stattfinden. Glassfish unterstützt aber, im Gegensatz zu Orion, keine globalen JNDI-Lookups von Entity Beans mit ausschließlich lokalen Schnittstellen. Da dem Servlet erst nach dem Aufruf des Konstruktors der lokale JNDI-Kontext zugewiesen wird, kann es im Konstruktor keine Entity Beans finden. Die JNDI-Lookups konnten vom Konstruktor in die Methode `init` der Servlets verlegt werden. Diese wird vom Container einmalig nach der Erzeugung eines Servlets aufgerufen. In der Methode sollen einmalige Aktivitäten durchgeführt werden, wie z. B. das Lesen von persistenten Daten oder das Initialisieren von teuren Ressourcen wie z. B. Datenbankverbindungen (vgl. [Mor07]).

Der Authentifizierungsmechanismus

Glassfish bietet verschiedene Authentifizierungsmechanismen an, die *Realms* genannt werden. Der Plan sah vor, Orions `DatabaseUserManager` durch Glassfishs `JDBCRealm` zu ersetzen, da beide die gleiche Funktionalität haben.

Der Austausch war so einfach nicht möglich: Es gibt für jeden Benutzer Gruppenzugehörigkeiten, die als `Set` in der Entity Bean `User` gespeichert werden. Orion hat die einzelnen Elemente des `Set` in einer mit den Daten der Entity Bean verknüpften Tabelle gespeichert.

Glassfish serialisiert das `Set` und speichert es in einem *Binary Large Object* (BLOB), das einem Bytearray der Einträge entspricht, in der Datenbank. Gleichzeitig verlangt das `JDBCRealm`

von Glassfish allerdings eine Datenbanktabelle, in der die Verknüpfung zwischen Benutzern und Gruppen als Relation gespeichert wird.

Da das BDG nur eine Benutzergruppe kennt, wurde das `JDBCRealm` so konfiguriert, dass jeder Benutzer zu dieser Gruppe zugehörig angesehen wird.

Hashing von Passwörtern

In Zusammenhang mit dem Austausch des Authentifizierungsmechanismus ergab sich auch ein Problem mit der Berechnung der Hashes der Passwörter.

Beim Anlegen eines neuen Benutzers wird ein Hash des Passworts in der Datenbank gespeichert. Die Berechnung des Hash wird nicht mittels einer Funktion des User Managers bzw. des Realms durchgeführt, sondern in der Entity Bean `User`. Die Speicherung des Hash unterscheidet sich zwischen Orion und Glassfish:

Orion nutzt einen String, dessen einzelne Zeichen jeweils den Wert eines Bytes darstellen, während Glassfish einen String verwendet, der den Hash als hexadezimale Zahl darstellt. Das Byte mit dem Wert `65` könnte von Orion z. B. als `A` gespeichert werden, während Glassfish den gleichen Wert als `41` speichert.

Eine Möglichkeit, den Hash des Passworts beim Anlegen durch das `JDBCRealm` berechnen zu lassen, wurde nicht gefunden. Daher wurde die Methode, die den Hash des Passworts berechnet, so angepasst, dass sie Hashes im vom `JDBCRealm` erwarteten Format erzeugt.

Login

Java EE spezifiziert, wie die HTML-Form-Elemente und deren Eingabefelder heißen müssen, um die Eingabe des Benutzernamens und des Passworts an den Authentifizierungsmechanismus des Applikationsservers weiterzuleiten (vgl. [Mor07]). Der Login über eine JSP funktioniert in Orion, obwohl die Namen des HTML-Form-Elements nicht dem Standard entsprechen. Glassfish besteht auf diesen Namen und kann auch keine Fehlermeldung ausgeben, da die Benutzung eines „falschen“ Namens keinen Fehler darstellt; es wird lediglich der Inhalt des HTML-Form-Elements nicht an den Authentifizierungsmechanismus weitergeleitet.

Lokale Interfaces in der Signatur von Remote Interfaces

Die Session Beans wurden im ersten Migrationsschritt nicht geändert, da dies scheinbar nicht nötig war, um die Migration auf Glassfish möglich zu machen. Sie hätten direkt von Version 1.1 auf Version 3.0 migriert werden sollen. Glassfish lässt aber dem Standard entsprechend (vgl. [Sun01]) keine Local Interfaces von EJBs in den Signaturen der Methoden eines Remote Interface zu. Genau dies nutzen jedoch die Session Beans des BDG.

Um dieses Problem zu lösen, wurden auch die Session Beans auf die Nutzung von Local Interfaces umgestellt.

Primärschlüsselklassen

Laut der EJB-Spezifikation müssen Primärschlüsselfelder einer Primärschlüsselklasse CMP-Variablen sein (vgl. [Sun03]). Orion lässt auch CMR-Variablen zu, während Glassfish sich an die Spezifikation hält.

Es mussten in der Klasse `RolePK`, die die Primärschlüsselklasse der Entity Bean `Role` ist, Anpassungen vorgenommen werden, da sie als Schlüsselfelder CMR-Variablen vom Typ `User` und `Game` definiert hatte. Um die Eindeutigkeit des Schlüssels zu wahren, wurden die Primärschlüsselfelder der CMR-Variablen, also der Benutzername aus `User` und der Name des Spiels aus `Game`, als Schlüsselfelder verwandt.

Delegation an ejbCreate

Nicht dem Standard entsprechend (vgl. [Sun01]), aber von Orion geduldet, ist der Aufruf einer `ejbCreate`-Methode aus einer anderen `ejbCreate`-Methode. Dies ist in Zeile 29 der `DeliveryEJB` in Listing 1 zu sehen. Glassfish dagegen lässt dies nicht zu. Wird dieser Code auf Glassfish ausgeführt, wird eine `TransactionRollbackException` geworfen, deren Ursache schwer zu finden ist, wenn der oben dargestellte Sachverhalt unbekannt ist. Die Lösung ist in Listing 2 zu sehen: Statt die zweite `ejbCreate`-Methode aufzurufen, wird der Code dupliziert.

Parallele Zugriffe

Laut dem Java-EE-Standard darf ein Applikationsserver eine Ausnahme werfen, wenn auf eine Session Bean aus mehr als einem Thread gleichzeitig zugegriffen wird. Da Session Beans nur für die Benutzung durch einen Client gedacht sind, handelt es sich bei dieser Art des Zugriffs um einen Fehler in der Anwendung (vgl. [Sun01]).

Im BDG erfolgt der Zugriff auf die Session Beans über Servlets. Der Container legt von jedem Typ eines Servlets genau ein Objekt an, das alle Anfragen an dieses Servlet beantwortet.

Das BDG hat zwei Eigenarten, die dazu führen, dass die Servlets aus mehreren Threads auf dieselbe Session Bean zugreifen:

- Es gibt nicht eine Session Bean je Spieler, sondern eine Session Bean je Spiel, die von allen Spielern des jeweiligen Spiels gemeinsam genutzt wird.
- Das BDG nutzt Frames und bei der Anzeige der Daten des Spiels werden zwei Frames gleichzeitig geladen, die das selbe Servlet nutzen. Auch hier finden parallele Aufrufe des Servlets und damit der Session Bean statt.

Orion lässt den parallelen Zugriff zu, der in den Tests nicht zu Problemen führte. Glassfish lässt dagegen keinen parallelen Zugriff zu. Die Verarbeitung des jeweiligen Servlets wird abgebrochen und der Benutzer auf eine Fehlerseite umgeleitet.

Als Lösung des Problems wurde an den entsprechenden Stellen `synchronized`-Blöcke in das Servlet eingefügt. Eine bessere Lösung wäre die Beseitigung der beiden genannten Probleme. Dies wäre allerdings mit einem größeren Refactoring des BDG verbunden.

MySQL

Nach der Migration gab Glassfish Fehlermeldungen aus, wenn auf das Home Interface von `User` zugegriffen werden sollte. Dies lag an der Version des MySQL Connectors (dem Adapter zwischen JDBC und MySQL) der benutzt wurde. Version 5.1.0 des MySQL Connectors verursachte den Fehler, mit Version 5.0.8 trat er nicht mehr auf.

Zeitaufwand und Gründe für Probleme

Dieser Teil der Migration hat ca. 70 Arbeitsstunden in Anspruch genommen. Ein Großteil dieser Zeit wurde dafür aufgewandt, das BDG standardkonform zu gestalten. Dies war zum einen wegen der Migration der Entity Beans notwendig, da dabei der Standard nicht vollkommen eingehalten wurde. Aber auch andere Bestandteil des BDG waren nicht standardkonform.

Hier hätten vor allem ein standardkonformes BDG und mehr Wissen über den Java EE Standard geholfen, die benötigte Zeit zu verringern. Auch ein Werkzeug, das Standardverletzung auffindet wäre sehr hilfreich gewesen. Glassfish liegt zwar ein solches Werkzeug, der *Verifier*, bei, allerdings arbeitet es unzuverlässig und findet nicht alle Abweichungen vom Standard.

4.4.4 Migration von EJB 2.0 zu EJB 3.0/Java Persistence API Entities

Als erstes wurden die Session Beans auf Version 3.0 migriert. Dabei wurde, wie geplant, dem Leitfaden aus Abschnitt 3.7 gefolgt.

Die Migration von Session Beans soll anhand des `NameSuggestor` dargestellt werden. Dabei handelt es sich um eine Stateless Session Bean, die dem Benutzer einen neuen Namen vorschlägt, wenn der von ihm gewählte Name bereits vergeben ist. Der Quelltext der Implementationsklasse im Altsystem findet sich in Listing 3. Der Quelltext für das Neusystem ist in Listing 4 abgedruckt. Beide Quelltexte sind gekürzt und so formatiert, dass möglichst wenige Zeilen zur Darstellung benötigt werden.

Ausgehend von Listing 3 wurde als erstes `implements SessionBean` entfernt und durch die Annotation `Stateless` ersetzt. Eine Session Bean 3.0 muss von ihrem Business Interface erben, also wurde `implements NameSuggestor` hinzugefügt. Das Business Interface erbt nicht mehr von `EJBLocalObject` und wird somit zum POJI. Es wird mit `@Local` annotiert, um es als Local Interface einer Session Bean zu kennzeichnen.

Da die Implementationsklassen der Session Beans nicht mehr das Interface `SessionBean` implementieren müssen, konnten die Callback-Methoden `ejbActivate`, `ejbPassivate`, `ejbRemove` und `setSessionContext` entfernt werden.

Im Falle des BDG waren die Callback-Methoden in allen Session Beans leer.

```
1  package beergame.ejb.util;
2  import javax.ejb.*;
3  import javax.naming.*;
4  import javax.sql.DataSource;
5  import java.rmi.*;
6  import java.sql.*;
7  public class NameSuggestorEJB implements SessionBean {
8          protected String userTable;      // Name der User-Tabelle.
9          protected String userColumn;     // Name der Loginname-Spalte.
10         protected String gameTable;      // Name der Spiele-Tabelle.
11         protected String gameColumn;     // Name der Spielname-Spalte.
12         public void ejbCreate() {
13                 try {
14                         Context context = new InitialContext();
15                         userTable = (String) context.lookup("java:comp/env/userTable");
16                         userColumn = (String) context.lookup("java:comp/env/userColumn");
17                         gameTable = (String) context.lookup("java:comp/env/gameTable");
18                         gameColumn = (String) context.lookup("java:comp/env/gameColumn");
19                 } catch (NamingException exception) {
20                         throw new EJBException(exception);
21                 }
22         }
23         // Business Methoden
24         public String suggestLoginName(String name) {
25                 return suggestName(name, userTable, userColumn);
26         }
27         public String suggestGameName(String name) {
28                 return suggestName(name, gameTable, gameColumn);
29         }
30         // Hilfsmethoden
31         protected String suggestName(String name, String table, String column) {
32                 Connection connection = null;
33                 PreparedStatement statement = null;
34                 try {
35                         Context context = new InitialContext();
36                         DataSource ds = (DataSource) context
37                                         .lookup("java:comp/env/jdbc/beerDS");
38                         connection = ds.getConnection();
39                         ...
40                         return name + (max + 1);
41                 } catch (SQLException exception) {
42                         throw new EJBException(exception);
43                 } catch (NamingException exception) {
44                         throw new EJBException(exception);
45                 } finally {
46                         try {
47                                 if (connection != null) connection.close();
48                                 if (statement != null) statement.close();
49                         } catch (SQLException exception) {
50                                 exception.printStackTrace();
51                         }
52                 }
53         }
54         // EJB Callback Methoden
55         public void ejbActivate() {}
56         public void ejbPassivate() {}
57         public void ejbRemove() {}
58         public void setSessionContext(SessionContext ctx) {}
59 }
```

Listing 3: Der NameSuggestor im Altsystem

```
1  package beergame.ejb.util;
2  import java.sql.*;
3  import javax.annotation.Resource;
4  import javax.ejb.*;
5  import javax.sql.DataSource;
6  @Stateless
7  public class NameSuggestorBean implements NameSuggestor {
8          @Resource(name = "userTable")    // Name der User-Tabelle.
9          protected String userTable = "users";
10         @Resource(name = "userColumn")   // Name der Loginname-Spalte.
11         protected String userColumn = "username";
12         @Resource(name = "gameTable")    // Name der Spiele-Tabelle.
13         protected String gameTable = "game";
14         @Resource(name = "gameColumn")   // Name der Spielname-Spalte.
15         protected String gameColumn = "name";
16         @Resource(name = "jdbc/beerDS") // Name der JDBC-DataSource
17         protected DataSource ds;
18         // Business Methoden
19         public String suggestLoginName(String name) {
20                 return suggestName(name, userTable, userColumn);
21         }
22         public String suggestGameName(String name) {
23                 return suggestName(name, gameTable, gameColumn);
24         }
25         // Hilfsmethoden
26         protected String suggestName(String name, String table, String column) {
27                 Connection connection = null;
28                 PreparedStatement statement = null;
29                 try {
30                         connection = ds.getConnection();
31                         ...
32                         return name + (max + 1);
33                 } catch (SQLException exception) {
34                         throw new EJBException(exception);
35                 } finally {
36                         try {
37                                 if (connection != null) connection.close();
38                                 if (statement != null) statement.close();
39                         } catch (SQLException exception) {
40                                 exception.printStackTrace();
41                         }
42                 }
43         }
44         public NameSuggestorBean() {}
45 }
```

Listing 4: Der NameSuggestor im Neusystem

Wäre eine der ersten drei Methoden noch benötigt worden, hätte eine Methode jeweils mit `@PostActivate`, `@PrePassivate` bzw. `@PreDestroy` annotiert werden können. Um den Session-Kontext per DI, statt über die Methode `setSessionContext`, zu erhalten hätte eine Exemplarvariable des Typs `SessionContext` mit `@Resource` annotiert werden können.

Der Code, der vorher in der Methode `ejbCreate` stand, wurde in den Konstruktor der Session Bean verschoben. Jetzt handelt es sich bei der Bean um ein POJO.

In Zeile 18-22 und 40 wurde im Altsystem jeweils ein JNDI-Lookup durchgeführt. Diese Lookups können durch DI ersetzt werden. Dies geschieht durch Annotieren der Exemplarvariablen mit `@Resource`.

Im BDG sind der `NameSuggestor` und der `GameManager` Stateless Session Beans.

Der `GameSessionManager` ist eine Stateful Session Bean. Die JNDI-Lookups für den `GameSessionManager` wurden aus den im Leitfaden auf S. 43 genannten Gründen nicht durch DI ersetzt.

Der Eintrag in `ejb-jar.xml` für diese Session Bean konnte vollständig entfernt werden.

Schließlich konnte das Home Interface gelöscht werden, da es nicht mehr benötigt wurde. Dies führte zu Fehlern beim Kompilieren, da die Session Bean noch darüber aufgefunden bzw. erzeugt wurde. Die vom Home Interface zurückgegeben Referenz auf die Session Bean, wurde in Exemplarvariablen der jeweiligen Klasse gespeichert.

In Klassen, die vom Container verwaltetet werden, (d. h. Servlets und EJB) konnte DI genutzt werden, um eine Referenz auf die Stateless Session Beans zu erhalten. In anderen Klassen ist, wie bereits geschildert, DI nicht möglich. Für den Fall dieser Migration bedeutet das, dass die Klasse, von der alle Servlets des BDG erben, mit

```
@EJBs(value = {
                @EJB(name="ejb/GameManager",beanInterface=GameManager.class),
                @EJB(name="ejb/NameSuggestor", beanInterface=NameSuggestor.class),
                @EJB(name="ejb/GameSession",beanInterface=GameSession.class)
                })
```

annotiert werden muss. Dies liegt daran, dass die Servlets *Aufrufer* im Sinne des Entwurfsmusters *Kommando* (vgl. [GHJV96]) sind. Die entsprechenden *Befehle*, in deren Methoden die JNDI-Lookups ausgeführt werden sind keine vom Container verwalteten Klassen. Daher ist keine DI möglich.

Dies ersetzt einen Eintrag im Deployment Deskriptor. Die Annotation führt dazu, dass in allen Objekten, die vom Servlet aus aufgerufen werden, die angegebenen EJB per JNDI-Lookup zu finden sind. Alternativ kann auch weiterhin ein Eintrag im Deployment Deskriptor genutzt werden. Dieser ist, wie oben dargestellt, für nicht vom Container verwaltete Klassen weiterhin nötig (vgl. [PRL07]).

In Listing 5 ist ein Ausschnitt aus dem Deployment Deskriptor für die EJB aus dem Altsystem zu sehen. Der Ausschnitt enthält die vollständige Definition der `DeliveryEJB` und die Definition einer CMR.

Der vollständige Deployment Deskriptor für die Entities des BDG ist in Listing 6 zu sehen.

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN" "http://java.sun.com
    /dtd/ejb-jar_2_0.dtd">
<ejb-jar>
        <display-name>Beergame (EJB-Module)</display-name>
        <enterprise-beans>
                <entity>
                        <description>
                                Dieses EntityBean repraesentiert eine Lieferung
                        </description>
                        <display-name>Delivery</display-name>
                        <ejb-name>Delivery</ejb-name>
                        <local-home>beergame.ejb.dbo.DeliveryHome</local-home>
                        <local>beergame.ejb.dbo.Delivery</local>
                        <ejb-class>beergame.ejb.dbo.DeliveryEJB</ejb-class>
                        <persistence-type>Container</persistence-type>
                        <prim-key-class>java.lang.Long</prim-key-class>
                        <reentrant>False</reentrant>
                        <cmp-version>2.x</cmp-version>
                        <abstract-schema-name>DeliveryBean</abstract-schema-name>
                        <cmp-field>
                                <field-name>deliveryNumber</field-name>
                        </cmp-field>
                        <cmp-field>
                                <field-name>round</field-name>
                        </cmp-field>
                        <cmp-field>
                                <field-name>amount</field-name>
                        </cmp-field>
                        <cmp-field>
                                <field-name>role</field-name>
                        </cmp-field>
                        <primkey-field>deliveryNumber</primkey-field>
                        <security-identity>
                                <use-caller-identity />
                        </security-identity>
                        <query>
                                <query-method>
                                        <method-name>findByGame</method-name>
                                        <method-params>
                                                <method-param>
                                                        beergame.ejb.dbo.Game
                                                </method-param>
                                        </method-params>
                                </query-method>
                                <ejb-ql>
                                        select Object(d) from DeliveryBean AS d where d.game
                                        = ?1
                                </ejb-ql>
                        </query>
                </entity>
        </enterprise-beans>
        <relationships>
                <ejb-relation>
                        <description>Ein Spiel hat mehrer Lieferungen</description>
                        <ejb-relationship-role>
                                <multiplicity>Many</multiplicity>
                                <cascade-delete />
                                <relationship-role-source>
                                        <ejb-name>Delivery</ejb-name>
                                </relationship-role-source>
                                <cmr-field>
                                        <cmr-field-name>game</cmr-field-name>
                                </cmr-field>
                        </ejb-relationship-role>
                        <ejb-relationship-role>
                                <multiplicity>One</multiplicity>
                                <relationship-role-source>
                                        <ejb-name>Game</ejb-name>
                                </relationship-role-source>
                        </ejb-relationship-role>
                </ejb-relation>
        </relationships>
</ejb-jar>
```

Listing 5: Ausschnitt aus dem Deployment Deskriptor für die EJBs aus dem Altsystem

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4           xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
5           http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
6    <persistence-unit name="BeerGamePU" transaction-type="JTA">
7      <provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</provider>
8      <jta-data-source>jdbc/beerEJBDS</jta-data-source>
9      <properties>
10       <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
11     </properties>
12   </persistence-unit>
13 </persistence>
```

Listing 6: Deployment Deskriptor für Entities

Im Vergleich der beiden Deployment Deskriptoren ist zu sehen, dass fast alle Einstellungen, die im Altsystem in Deployment Deskriptoren vorgenommen wurden, im Neusystem per Annotation oder *Convention over Configuration* in der Klasse, die das Entity definiert, vorgenommen werden.

Um die Entity Beans zu migrieren, wurde ebenfalls dem Leitfaden aus Abschnitt 3.7 gefolgt.

Das Endergebnis der Migration der `DeliveryEJB` ist in Listing 7 zu sehen. Zum einen ist zu sehen, dass die Klasse weniger Zeilen Code benötigt, zum anderen sind alle Callback-Methoden verschwunden.

Da es sich um ein POJO handelt, ist auch der Aufruf eines Konstruktors aus einem anderen heraus möglich (vgl. Zeile 28). Das Pendant zu CMR der Entity Beans wird über Annotationen definiert (vgl. Zeile 12). In Zeile 5 wird eine JP-QL-Abfrage definiert. Diese ersetzt die Finder-Methoden, die im Deployment Deskriptor der Beans und im Home Interface definiert wurden.

Im Folgenden werden wieder die Probleme, die bei diesem Teil Migration aufgetreten sind, kurz dargestellt.

Name der Implementationsklasse

Bei der Migration trat das Problem auf, dass das Projekt zwar auf Glassfish deployed werden konnte, die Session Beans aber nicht gefunden werden konnten. Dies lag darin begründet, dass der Name der Implementationsklasse einer durch Annotationen konfigurierten Session Bean auf `Bean` enden muss. Dies ist den Spezifikationen nicht zu entnehmen (vgl. [EJB06]), könnte aber dennoch standardkonform sein, da Glassfish die Referenzimplementation des Standards ist.

CMR zwischen Entity Beans und Entities

Auch hier musste von dem Ziel der schrittweisen Migration abgewichen werden. Und zwar aus einem ähnlichen Grund wie bei der Migration von EJB 1.1 zu EJB 2.0.

CMR funktioniert nur zwischen Entity Beans, nicht zwischen Entity Beans und Entities. Es mussten also wiederum alle Entity Beans, die durch CMR untereinander verbunden waren, in

```
1  package beergame.entities;
2  import javax.ejb.CreateException;
3  import javax.persistence.*;
4  @Entity
5  @NamedQuery(name = "findDeliveriesByGame", query = "select Object(d) from Delivery AS d where d.game= ?1")
6  public class Delivery {
7          // persistente Attribute
8          @Id
9          @GeneratedValue(strategy = GenerationType.AUTO)
10         private Long deliveryNumber;
11         private String role;
12         @ManyToOne
13         private Game game;
14         private int round;
15         private int amount;
16         public Delivery(String role, Game game, int round, int amount)
17                         throws CreateException {
18                 if (game != null && amount > 0) {
19                         this.role = role;
20                         this.round = round;
21                         this.amount = amount;
22                         this.game = game;
23                 } else {
24                         throw new CreateException();
25                 }
26         }
27         public Delivery(Game game, int round, int amount) throws CreateException {
28                 this(null, game, round, amount);
29         }
30         protected Delivery() {} //Jedes Entity muss einen private oder protected Konstruktor ohne Parameter haben
31         // Business Methoden
32         public int getAmount() {        // Gibt die Liefermenge zurueck.
33                 return amount;
34         }
35         public Game getGame() {         // Gibt das Spiel zurueck.
36                 return game;
37         }
38         public int getRound() {         // Gibt die Runde zurueck, in der diese Lieferung ausgefuehrt
39                 return round;
40         }
41         public String getRole() {       // Gibt die Rolle, die diese Lieferung ausgefuehrt hat, zurueck.
42                 return role;
43         }
44 }
```

Listing 7: Delivery als JPA Entity

einem Schritt migriert werden. Es wurden alle Entity Beans des BDG in einem Schritt migriert.

Dependency Injection in JSPs

DI in JSPs ist nicht möglich. Es kann aber über einen `EntityManager` auf die JPA Entities zugegriffen werden.

Eine Referenz auf einen `EntityManager` kann über eine `EntityManagerFactory` erlangt werden. Die Referenz auf die `EntityManagerFactory` kann wiederum über eine Klassenmethode von `javax.persistence.Persistence` beschafft werden.

Dieser Fall trat nur in der Datei `details.jsp` auf. In Listing 8 und 9 sind Ausschnitte aus der Datei dargestellt, an denen zu sehen ist, wie der JNDI-Lookup durch die Erzeugung eines `EntityManager` und einen Methodeaufruf an selbigem ersetzt wurde.

```
1 <%@ page language="java"
2          import="beergame.ejb.dbo.*,javax.naming.*,javax.rmi.*,java.text.DateFormat"
3 %>
4 <%
5     String gamename = request.getParameter("game");
6     Context context = new InitialContext();
7     GameHome home = (GameHome)context.lookup("java:comp/env/ejb/Game");
8     Game game = home.findByPrimaryKey(gamename);
9 %>
```

Listing 8: Ausschnitt aus details.jsp aus dem Altsystem

```
1 <%@ page language="java"
2          import="beergame.entities.*,java.text.DateFormat"%>
3 <%
4          EntityManagerFactory entityManagerFactory = Persistence.createEntityManagerFactory("BeerGamePU");
5          EntityManager entityManager = entityManagerFactory.createEntityManager();
6          String gamename = request.getParameter("game");
7          Game game = entityManager.find(Game.class, gamename);
8 %>
```

Listing 9: Ausschnitt aus details.jsp aus dem Neusystem

MySQL

im Zusammenhang mit MySQL trat ein Fehler auf, der nur schwer zu identifizieren war. Nach jedem fünften Aufruf der Startseite wurde eine Ausnahme ausgelöst. In Folge dessen kam es zu der Anzeige einer Fehlerseite. Nach dem Umstieg von Version 5.0.8 des MySQL-Connectors auf Version 5.1.6 war der Fehler behoben.

Unerklärliches Verhalten

Hin und wieder kam es vor, dass Glassfish zwar noch das BDG ausführte, gleichzeitig aber Fehler auftraten, die auch nach längerem Debugging nicht zu erklären waren. Es scheint, dass Glassfish in einen inkonsistenten Zustand gerät, wenn eine Anwendung zu häufig neu deployed wird. Die Lösung ist ein Neustart des Servers.

Zeitaufwand und Gründe für Probleme

Für diesen Abschnitt der Migration wurden ca. 80 Arbeitsstunden benötigt.

Dies erscheint viel im Vergleich zu den anderen beiden Schritten, ist aber teilweise darin begründet, dass die Testklassen an die neuen Schnittstellen der EJBs bzw. der JPA Entities angepasst werden mussten.

Außerdem ist die Umstellung der Entity Beans zu Entities zeitaufwändige Handarbeit. Eine Teilautomatisierung dieses Migrationsschritt wäre denkbar. Die Migration der Entity Beans wurde ohne spezielle Migrationswerkzeuge durchgeführt, da solche nicht gefunden werden konnten.

Sollte solche Werkzeuge existieren, hätte der Zeitgewinn durch die Benutzung eines Werkzeugs bei dieser geringen Anzahl von Entity Beans wahrscheinlich den Zeitaufwand zum Erlernen der Benutzung des Werkzeugs nicht übertroffen.

4.4.5 Beurteilung der Migration

Zu der gesamten Migration ist zu sagen, dass der meiste Arbeitsaufwand nicht durch die eigentliche Migration entstanden ist, sondern durch die Fehlersuche. Die Anzahl der Probleme ist zwar überschaubar, Fehlersuche und der Behebung hat aber jeweils mehrere Stunden gedauert. Durch bessere Dokumentation von Orion, des BDG und in geringerem Umfang auch von Glassfish hätte dies verhindert werden können. Sicherlich wäre auch eine genauere Kenntnis des Java-EE-Standards hilfreich gewesen.

Auch die Fehlermeldungen beider Applikationsserver waren oft irreführend und haben somit häufig nicht soviel zur Fehlersuche beigetragen, wie es vermutlich möglich gewesen wäre.

Ein weiteres Hindernis beim Auffinden von Fehlern in Java-EE-Applikationen ist der Applikationsserver. Dieser übernimmt viele Aufgaben für die Anwendung. Einerseits erleichtert dies dem Entwickler die Arbeit, da er sich z. B. nicht um das Beginnen und Beenden von Transaktion, Nebenläufigkeit usw. kümmern muss. Andererseits gibt er aber die Kontrolle über die Anwendung teilweise an den Applikationsserver ab. Dies erschwert die Fehlersuche, da er nicht nur seinen eigenen Code, sondern auch den Applikationsserver und was dieser tut, verstehen muss.

Eine weitere Fehlerquelle ist die Nutzung von Komponenten, die außerhalb des Applikationsservers liegen. Das BDG benutzt nur eine externes System, MySQL. Es ist aber denkbar, dass größere Anwendungen mehrere externe Systeme anbinden. Wie in dieser Migration gesehen werden konnte, können unerklärliche Fehler auftreten, die letztlich auf verschiedene Versionen des jeweiligen MySQL Connectors zurückzuführen waren.

Bezüglich des gewählten Migrationsansatzes ist festzuhalten, dass er sich aus meiner Sicht als die richtige Wahl herausgestellt hat. Eine Big-Bang-artige Migration hätte viele der oben geschilderten Probleme auf einmal auftreten lassen und damit die Fehlersuche weiter erschwert. Außerdem gibt eine iterative Migration die Möglichkeit während der Migration die Vorgehensweise zu verändern und somit auf unerwartete Probleme sofort reagieren zu können. Dadurch wird verhindert, dass sämtliche gleichartigen Schritte die gleichen Fehler enthalten, die dann

wiederum nach der Migration korrigiert werden müssen.

Es muss aber auch gesehen werden, dass die Migration der Entity Beans von Version 1.1 auf Version 2.0 nicht notwendig war, da sie direkt weiter zu JPA Entities migriert wurden. Auch sind einige Fehler nur aufgetreten, weil zwischenzeitlich auf Version 2.0 der Entity Beans migriert wurde. Es muss also im Einzelfall zwischen Risiko und Arbeitsaufwand abgewägt werden.

Die Gruppe, die die Migration parallel zu dieser Arbeit als Big-Bang-Migration mit Neuentwicklungs-Strategie durchgeführt hat, würde allerdings wieder diesen Migrationsansatz wählen (vgl. [Cio08]).

4.5 Kapitelzusammenfassung

In diesem Kapitel wurden das Beer Distribution Game und drei mögliche Ansätze dieses zu migrieren vorgestellt.

Es wurde der iterative Migrationsansatz gewählt, um die Testbarkeit während der Migration zu gewährleisten.

Anschließend wurde die Migration und die dabei auftretenden Schwierigkeiten geschildert. Die Ursachen für die Schwierigkeiten waren vor allem schlechte Dokumentation, mangelnde Standardkonformität Orions und des BDG sowie fehlende Erfahrung im Umgang mit Java EE und den beiden Applikationsservern.

5 Evaluation

In diesem Kapitel soll versucht werden, anhand verschiedener Code-Metriken zu beurteilen, welchen Einfluss die Migration des Beer Distribution Game auf die Komplexität der Software hatte.

Um dies zu ermöglichen wurden jeweils das Altsystem und das Neusystem mit verschiedenen Metriken vermessen. Dazu wurden JDepend[1] und das Metrics-Plugin für Eclipse[2] verwandt. Außerdem wurden die *Lines of Code* (LOC) der einzelnen Klassen und XML-Dateien mittels eines selbstgeschriebenen Java-Programms gezählt.

Die Metriken und die Ergebnisse der Messungen werden in den folgenden Abschnitten dargestellt. Sie sind in diesem Kapitel auf das Wesentliche gekürzt; eine umfangreichere Übersicht findet sich im Anhang A und auf der beigelegten CD-ROM.

5.1 Lines of Code

Diese Metrik misst die Anzahl der Zeilen in den Quellcode-Dateien. Für diese Arbeit wurden alle Zeilen in Java-Klassen und -Interfaces gezählt. Die JSPs und HTML-Dateien mussten für die Migration nur sehr geringfügig geändert werden, so dass die Anzahl der Zeilen in etwa gleich blieb. Es werden alle Zeilen, also insbesondere auch Kommentare und leere Zeilen, gezählt. Außerdem wurden auch alle Zeilen in den Deployment Deskriptoren gezählt.

Insgesamt ist es wünschenswert eine bestimmte Funktionalität mit möglichst wenigen Zeilen Code zu erreichen. Allerdings hängt diese Metrik stark von Formatierungskonventionen ab. Daher ist bei einem Vergleich von Programmen auf Basis dieser Metrik auf identische Formatierungskonventionen des jeweiligen Quelltexts zu achten (vgl. [Ros97]). Dies ist im Falle der Beispielmigration gegeben.

Wie Tabelle 4 zu entnehmen ist, hat die Migration die Anzahl der Zeilen in Java-Klassen und -Interfaces von 8552 auf 8014 verringert. Außerdem hat sich die Anzahl der Zeilen in XML-Dateien von 801 auf 232 verringert.

In Tabelle 5 ist zu sehen, dass JPA Entities mit wesentlich weniger Code als Entity Beans realisiert werden können. Dies liegt zum einen am Wegfall des Business- und des Home-Interface. Zum anderen sind die Callback-Methoden, die Entity Beans auch implementieren müssen, wenn sie keine Anweisungen enthalten, weggefallen.

[1] http://clarkware.com/software/JDepend.html
[2] http://metrics.sourceforge.net/

GEMESSEN	LOC IM ALTSYSTEM	LOC IM NEUSYSTEM
Java Code	8552	8014
XML	801	232
Gesamt	**9353**	**8246**

Tabelle 4: Veränderung in der Anzahl der Zeilen

	LOC IM ALTSYSTEM				LOC IM NEUSYSTEM
ENTITY	BI	HI	IK	GESAMT	IK
Delivery	36	48	105	189	88
Game	196	59	406	661	456
Order	68	59	175	302	140
Role	62	32	88	182	98
User	55	42	253	350	186
BI: Business Interface		**HI: Home Interface**			**IK: Implementationsklasse**

Tabelle 5: Vergleich der LOC von Entity Beans und JPA Entities

Wie in Tabelle 6 hat sich bei den Session Beans der Code in der Implementationsklasse etwas verringert. Der im Vergleich zu den anderen Session Beans stärkere Rückgang beim `NameSuggestor` ist darauf zurückzuführen, dass viele JNDI-Lookups durch Dependency Injection ersetzt werden konnten.

In Tabelle 7 ist im Detail zu sehen, wie sich die Zeilenanzahl der XML-Dateien entwickelt hat. Eine Anzahl von 0 heißt dabei, dass die Datei nicht existiert.

Die Namen der Applikationsserver-spezifischen Deployment Deskriptoren beginnen mit `sun` bzw. `orion`. In Dateien mit dem gleichen Suffix müssen in etwa die gleichen Einstellungen vorgenommen werden.

Es haben sich vor allem die Deployment Deskriptoren für die EJBs verändert. Dabei handelt es

	LOC IM ALTSYSTEM			LOC IM NEUSYSTEM		
SESSION BEAN	BI	IK	GESAMT	BI	IK	GESAMT
GameManager	91	399	490	84	374	458
GameSession	188	1066	1254	194	1011	1205
NameSuggestor	34	144	178	34	102	136
BI: Business Interface			**IK: Implementationsklasse**			

Tabelle 6: Vergleich der LOC von Session Beans vor und nach der Migration

sich um die Dateien `ejb-jar.xml` und `orion-ejb-jar.xml` bzw. `sun-ejb-jar.xml`. Die Entities werden nicht zusammen mit den EJBs, sondern in `persistence.xml` konfiguriert. Die EJBs und Entities konnten im Neusystem mit 32 Zeilen XML konfiguriert werden. Im Altsystem waren dazu noch 489 Zeilen notwendig. Dies ist vor allem auf die Konfiguration per Annotationen und *Configuration by Exception* zurückzuführen.

Auffällig ist, dass sich die Anzahl Zeilen XML in Applikationsserver-spezifischen Deployment Deskriptoren von 274 auf 23 Zeilen XML reduziert hat. Dies lässt vermuten, dass eine Migration der neuen Version auf einen anderen Server weniger aufwändig ist, als die Migration der alten Version.

DATEINAME	LOC IM ALTSYSTEM	LOC IM NEUSYSTEM
application.xml	21	22
ejb-jar.xml	274	7
junitee-web.xml	72	36
orion-application.xml	41	0
persistence.xml	0	10
sun-ejb-jar.xml	0	15
orion-ejb-jar.xml	215	0
sun-web.xml	0	8
orion-web.xml	18	0
web.xml	160	134
Gesamt	**801**	**232**

Tabelle 7: Veränderung in der Zeilenanzahl der XML-Dateien

Zusätzlich zu den LOC können auch Klassen, Interfaces und Pakete gezählt werden. Auch deren Anzahl hat sich durch die Migration verringert, wie Tabelle 8 zu entnehmen ist.

GEMESSEN	ANZAHL IM ALTSYSTEM	ANZAHL IM NEUSYSTEM
Pakete	18	17
Klassen und Interfaces	105	88

Tabelle 8: Veränderung in der Anzahl der Klassen, Interfaces und Pakete

Insgesamt ist die Veränderung durch die Migration anhand dieser Metriken positiv zu beurteilen.

5.2 Lack of Cohesion in Methods

Die Metrik *Lack of Cohesion in Methods* (LCOM) kann genutzt werden, um die Kohäsion einer Klasse zu messen (vgl. [CK94]). Der etwas seltsam erscheinende Name der Metrik ist darin begründet, dass eine Messung des Mangels an Kohäsion dazu führt, dass ein niedriger Wert gut und ein hoher Wert schlecht ist. Dies entspricht den meisten anderen Metriken, bei denen im Allgemeinen ein niedriger Wert angestrebt wird.

Bei LCOM wird davon ausgegangen, dass eine hohe Kohäsion vorliegt, wenn die Methoden der Klasse jeweils auf möglichst viele Klassen- und Exemplarvariablen zugreifen. Im Folgenden werden zwei Varianten dieser Metrik vorgestellt.

5.2.1 LCOM nach Chidamber und Kemerer

Chidamber und Kemerer definieren LCOM erstmals in [CK91]. Diese Definition ist allerdings ungenau und daher missverständlich. Diese Arbeit verwendet die klarere Darstellung in [CK94]:

Es sei $M_1 \dots M_n$ die Menge aller Methoden einer Klasse. I_n sei die Menge aller Exemplarvariablen, auf die die jeweilige Methode zugreift. Weiterhin sei die Ähnlichkeit zweier Methoden definiert als: $\delta(M_1, M_2) = I_1 \cap I_2$. Die Menge aller ähnlichen Methoden sei:
$P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$
und die Menge aller unähnlichen Methoden sei:
$Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$.

Der Wert der Metrik berechnet sich als die Differenz der Mächtigkeit dieser Mengen und beträgt mindestens 0.
$LCOM = max(0, |P| - |Q|)$

Ein Wert von 0 deutet auf eine hohe Kohäsion hin, während ein hoher Wert auf eine geringe Kohäsion hindeutet.

5.2.2 LCOM* nach Henderson-Sellers

In [HS95] kritisiert Henderson-Sellers LCOM und definiert eine eigene Variante, die er LCOM* nennt:

Es sei m die Anzahl der Methoden und a die Anzahl der Exemplar- und Klassenvariablen der zu vermessenden Klasse.

Die Menge aller Methoden einer Klasse sei $\{M_i\} \mid i = 1 \dots m$ und die Menge aller Klassen- und Exemplarvariablen sei $\{A_j\} \mid j = 1 \dots a$.

Die Anzahl der Klassen- und Exemplarvariablen, auf die eine Methode zugreift sei $\alpha(M_i)$. Die Anzahl der Methoden die auf eine Klassen- oder Exemplarvariable zugreifen sei $\mu(A_j)$.

Die Berechnung der Metrik erfolgt dann folgendermaßen:

$$LCOM^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1-m}$$

Der Wertebereich dieser Metrik liegt zwischen 0 und 1, wobei 0 den besten Wert darstellt.

5.2.3 Ergebnisse

Die LCOM-Werte, die sich durch die Migration geändert haben, sind Tabelle 9 zu entnehmen. Dabei steht HS für LCOM* nach Henderson-Sellers und CK für LCOM nach Chidamber und Kimerer. Die LCOM-Werte für die Session Beans nach Chidamber und Kemerer haben sich

		LCOM im Altsystem		LCOM im Neusystem	
PAKET	KLASSE	HS %	CK	HS %	CK
beergame.ejb.managers	GameManagerEJB	77	0	81	0
beergame.ejb.managers	GameSessionEJB	87	57	88	3
beergame.ejb.util	NameSuggestorEJB	50	0	100	3
beergame.timeout	Timer	63	0	33	0
beergame.web.controller.game	AbstractGameCommand	50	0	0	0
beergame.ejb.dbo	GameEJB	99	180	97	456
beergame.ejb.dbo	OrderEJB	0	0	96	13
beergame.ejb.dbo	RoleEJB	100	3	100	6

Tabelle 9: LCOM-Werte vor und nach der Migration

verschlechtert. Wird der Wert nach Henderson-Sellers berechnet ist die Verschlechterung zu vernachlässigen. Bei den Entity Beans bzw. den Entities sind die Werte nach Henderson-Sellers für Game und Role fast konstant geblieben, während sie für Order von 0 auf 96% gestiegen sind. Dies ist wahrscheinlich durch einen Fehler im verwandten Messprogramm begründet. Die `OrderEJB` hat Getter und Setter, die jeweils nur eine Exemplarvariable benutzen und implementiert einige leere Callback-Methoden. Daher ist der gemessene Wert von 0 nicht nachvollziehbar.

Wird die Metrik nach Chidamber und Kemerer angewandt, verschlechtern sich die Messergebnisse für Order und Role leicht, während sich Game von einem bereits hohen Niveau von 180 auf 456 verschlechtert.

Insgesamt ist die Veränderung durch die Migration anhand dieser Metriken negativ zu beurteilen.

5.3 Weighted Methods per Class

Weighted Methods per Class (WMC) summiert die Komplexität der einzelnen Methoden einer Klasse. Sie wird z. B. in [CK94] und [CK91] dargestellt und lässt abschätzen, wie schwierig es

für einen Entwickler ist, eine Klasse zu verstehen. Zur Berechnung des Werts dieser Metrik werden die Werte für die Komplexität der Methoden einer Klasse summiert. Dabei gelten Klassen mit einem niedrigen Wert als leichter verständlich, und damit weniger komplex, als solche mit einem hohen Wert.

Die Metrik, nach der die Komplexität der Methoden bestimmt wird, ist wählbar (vgl. [HS95]). Es kommt jede Metrik in Betracht, die die Komplexität von Methoden misst. In dieser Arbeit wird die *zyklomatische Komplexität* nach McCabe verwandt.

Die zyklomatische Komplexität gibt die Anzahl der möglichen Pfade durch ein prozedurales Programm an, indem sie die Anzahl der Verzweigungen durch Auswahl und Wiederholung bestimmt (vgl. [McC76]). In objektorientierten Programmen werden, statt der möglichen Pfade durch ein Programm, die möglichen Pfade durch einzelne Methoden bestimmt (vgl. [HS95]).

Wie den Tabellen 10 und 11 zu entnehmen ist, ist durch die Migration die WMC sowohl der Entity Beans (bzw. der Entities, zu denen sie migriert wurden) und der Session Beans kleiner geworden. Bei RolePK handelt es sich um die Primärschlüsselklasse der Entity Bean Role.

Die Veränderung in den Session Beans ist teilweise durch den Wegfall von JNDI-Lookups zu erklären. Jeder JNDI-Lookup muss in einem try-catch-Block durchgeführt werden. Ein solcher Block erhöht die Anzahl der möglichen Pfade, und damit die zyklomatische Komplexität, um eins, da er ganz oder nur teilweise ausgeführt werden kann. Die JNDI-Lookups konnten durch Dependency Injection ersetzt werden.

Die Ursache für die Verringerung der WMC bei den Entity Beans ist nicht auszumachen. Insgesamt ist die Veränderung durch die Migration anhand dieser Metriken positiv zu beurteilen.

Entity Bean/ JPA Entity	WMC der Implementations-klasse der Entity Bean	WMC des Entity	Differenz
Delivery	16	8	8
Game	52	46	6
Order	22	14	8
Role	12	5	7
RolePK	7	5	2
User	40	22	18

Tabelle 10: Vergleich der WMC von Entity Beans und JPA Entities

5.4 Die Robert-Martin-Suite

Robert Martin definiert in [Mar03] und [Mar94] einige Metriken, die er als Heuristiken sieht, um den Zustand eines Software-Systems zu beurteilen.

SESSION BEAN	WMC IM ALTSYSTEM	WMC IM NEUSYSTEM	DIFFERENZ
GameSession	156	138	18
GameManager	56	46	10
NameSuggestor	17	11	6

Tabelle 11: Vergleich der WMC der Session Beans vor und nach der Migration

Martin definiert die Unabhängigkeit (Independence) eines Moduls anhand der Abhängigkeiten dieses Moduls: Enthält ein Modul keine Abhängigkeiten auf ein anderes, kann es nicht passieren, dass es geändert werden muss, weil sich eines der Module, von denen es abhängt, sich ändert. Es ist also *unabhängig*.

Er nennt Klassen *verantwortlich* (responsible), wenn es (viele) Klassen gibt, die von dieser Klasse abhängig sind. Diese Klassen sind für gewöhnlich stabil, da jede Änderung an der Klasse Änderungen an vielen anderen Klassen bedingt.

Klassen, die sowohl unabhängig als auch verantwortlich sind, sind seiner Ansicht nach die stabilsten Klassen:

> Such classes have no reason to change, and lots of reasons not to change. [Mar94, S. 5]

Ein wesentlicher Aspekt der objektorientierten Programmierung ist für Martin die Wiederverwendbarkeit von Programmteilen. Er geht davon aus, dass die Wiederverwendung einzelner Klassen nicht funktionieren kann, da für gewöhnlich mehrere Klassen stark voneinander abhängen. Daher geht er davon aus, dass diese stark voneinander abhängenden Klassen gemeinsam wiederzuverwenden sind. In [Mar94] nennt er diese Ansammlung von Klassen Kategorien (Categories), in [Mar03] nennt er sie Pakete (Packages).

In dieser Arbeit wird der Begriff Paket genutzt, da ein Java-Programm untersucht wird und das Konzept der Kategorien in Java durch Pakete realisiert ist.

Die Abhängigkeiten der Klassen innerhalb eines Pakets untereinander sieht er als harmlos an, da sei bei der Wiederverwendung als Paket nicht ins Gewicht fallen (vgl. [Mar94]).

Es ist also nicht nötig, die Abhängigkeiten auf Klassenebene zu diskutieren, da die Paketebene dafür aussagekräftiger ist. Es sind die Pakete am stabilsten, die einen hohen Wert von Unabhängigkeit und Verantwortlichkeit erreichen. Abhängigkeiten von stabilen Paketen sind „gute“ Abhängigkeiten (vgl. [Mar94]).

Um die Abhängigkeit zu bestimmen, definiert Martin folgende drei Metriken (vgl. [Mar94]):

Efferent Coupling (C_e) Efferente Kopplung bezeichnet die Anzahl an Klassen, von denen die im untersuchten Paket befindlichen Klassen abhängen. Das in dieser Arbeit genutzte Werkzeug zum Ermitteln dieser Metrik, JDepend, weicht von der Definition ab, da es die Anzahl der Pakete misst, von denen die Klassen innerhalb des vermessenen Pakets abhängen (vgl. [Cla08]).

Afferent Coupling (C_a) Afferente Kopplung bezeichnet die Anzahl von Klassen, die von den im vermessenen Paket vorhandenen Klassen abhängen. Auch hier weicht JDepend von der Definition ab und misst die Anzahl der Pakete, deren Klassen vom vermessenen Paket abhängen (vgl. [Cla08]).

Instability (I) Die *Instabilität* bezeichnet den Quotienten aus efferenter Kopplung und der Summe efferenter und afferenter Kopplung:
$I = \frac{C_e}{C_e + C_a}$

Wenn alle Pakete völlig stabil wären, wäre das System starr und könnte nicht verändert werden. Daher ist es nötig, auch instabile Pakete zu haben (vgl. [Mar94]).

Stabile Pakete sollten abstrakte Klassen bzw. Interfaces enthalten. Dadurch sind sie einerseits stabil verwendbar gleichzeitig aber auch erweiterbar. Es gilt also das *Offen-Geschlossen-Prinzip* (vgl. [Mar94, Zül98]).

Instabile Pakete enthalten konkrete Klassen und erben von den abstrakten Klassen in den stabilen Paketen. Sie sollten also von stabilen Paketen abhängig sein. Von ihnen sollten wiederum keine Pakete abhängig sein (vgl. [Mar94]).

Um bestimmen zu können, ob ein Paket abstrakt oder konkret ist, definiert er eine weitere Metrik (vgl. [Mar94]):

Abstractness (A) Die *Abstraktheit* eines Pakets ist definiert als das Verhältnis der Anzahl der abstrakten Klassen und Interfaces (C_a) zu der Gesamtzahl der Klassen und Interfaces des untersuchten Pakets. Es sei C_c die Anzahl der konkreten Klassen.
$A = \frac{C_a}{C_a + C_c}$

Distance from Main Sequence (D) Entsprechend den oben dargestellten Überlegungen zu Instabilität und Abstraktheit, hält Martin eine Balance zwischen diesen beiden Aspekten eines Pakets für wünschenswert. Als optimal sieht er Pakete an, die auf der Gerade $A + I = 1$ liegen. Diese Gerade nennt er *Hauptreihe* (Main Sequence).

Der Abstand eines Pakets zu dieser Geraden berechnet sich als:
$D = \frac{|A+I-1|}{\sqrt{2}}$

In dieser Arbeit wird der normalisierte Abstand von der Hauptreihe verwendet, so dass 0 für einen Wert auf der Geraden steht und 1 für einen Wert steht, der den maximalen Abstand von der Geraden hat. Dieser wird wie folgt berechnet:
$D' = |A + I - 1|$

Die Ergebnisse der Messungen sind in den Tabellen 12 und 13 festgehalten. Dabei ist zu beachten, dass das Paket `beergame.ejb.dbo` durch `beergame.entities` ersetzt wurde. Außerdem ist das Paket `test.beergame.ejb` ersatzlos weggefallen. Des Weiteren stellt Abbildung 7 dar, wie sich der Abstand von der *Hauptreihe* durch die Migration geändert hat.

Es ist zu sehen, dass sich die Anzahl der Klassen für die Entities von 16 auf 6 reduziert hat. Diese Reduktion ist auf den Wegfall der Home- und Business-Interfaces zurückzuführen.

PAKET	C_t	C_a	C_c	C_a	C_e	A	I	D
beergame.ejb.dbo	16	10	6	9	2	0,62	0,18	0,19
beergame.ejb.managers	19	5	14	7	3	0,26	0,30	0,44
beergame.ejb.util	6	2	4	4	1	0,33	0,20	0,47
test.beergame	2	0	2	0	8	0,00	1,00	0,00
test.beergame.ejb	1	0	1	2	2	0,00	0,50	0,50
test.beergame.ejb.dbo	4	0	4	1	4	0,00	0,80	0,20
test.beergame.ejb.managers	3	0	3	1	4	0,00	0,80	0,20
test.beergame.ejb.tools	1	0	1	1	1	0,00	0,50	0,50

Tabelle 12: Ergebnisse der Robert-Martin-Suite für das Altsystem

PAKET	C_t	C_a	C_c	C_a	C_e	A	I	D
beergame.entities	6	0	6	8	0	0,00	0,00	1,00
beergame.ejb.managers	17	3	14	6	3	0,18	0,33	0,49
beergame.ejb.util	5	1	4	4	1	0,20	0,20	0,60
test.beergame	2	0	2	0	4	0,00	1,00	0,00
test.beergame.ejb.dbo	3	0	3	0	3	0,00	1,00	0,00
test.beergame.ejb.managers	3	0	3	0	4	0,00	1,00	0,00
test.beergame.ejb.tools	1	0	1	0	2	0,00	1,00	0,00

Tabelle 13: Ergebnisse der Robert-Martin-Suite für das Neusystem

Gleichzeitig reduziert sich dadurch die *Abstraktheit* des Pakets von 0,62 auf 0. Die *Instabilität* reduziert sich auf 0, weil die *Efferenten Kopplungen* auf 0 gesunken sind. Die Entities haben also keine Abhängigkeiten von anderen Paketen. Das führt allerdings auch dazu, dass der normalisierte Abstand von der Hauptreihe auf 1 steigt. Dies ist der schlechteste Wert, den diese Metrik erreichen kann.

Die Änderungen an den Session Beans in den Paketen `beergame.ejb.managers` und `beergame.ejb.util` sind ähnlich zu bewerten: Es wurde ein (überflüssiges) Interface eingespart, was positiv zu sehen ist.

Allerdings sank dadurch die *Abstraktheit* und durch den Wegfall von Paketen bei den Unit Tests sinkt die Anzahl der *Afferenten Kopplungen* um 1, was die *Instabilität* erhöht. Dies führt insgesamt zu einem höheren *Abstand von der Hauptreihe*, was wiederum negativ zu sehen ist.

Die Bewertung anhand dieser Metrik fällt negativ aus, da die geänderten Pakete einen erhöhten oder gleichbleibenden Abstand von der Hauptreihe haben.

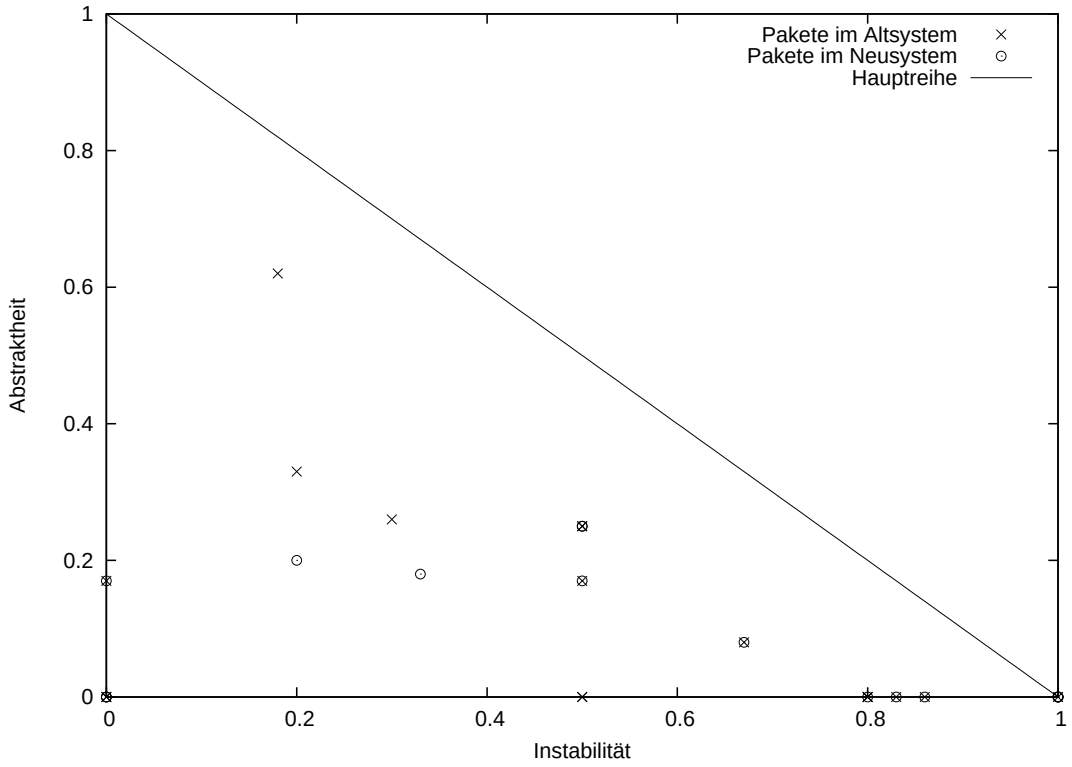

Abbildung 7: Der Abstand der Pakete von der Hauptreihe im Vergleich zwischen dem Alt- und dem Neusystem

5.5 Kapitelzusammenfassung

In diesem Kapitel wurden verschiedene Metriken vorgestellt und die Ergebnisse ihrer Anwendung auf das Alt- und das Neusystem verglichen. Die Metriken nach Martin und beide Varianten von LCOM zeigen eine Verschlechterung des Systemzustands durch die Migration.

Die Anzahl der Code-Zeilen, Klassen und Pakete sowie die WMC zeigen eine Verbesserung des Systemzustands.

Insgesamt ist aus diesen Daten kein endgültiger Schluss bezüglich der Veränderung der Komplexität durch die Migration zu ziehen.

6 Schlussbetrachtung

6.1 Zusammenfassung

In dieser Arbeit wurden zunächst verschiedene Migrationsstrategien und -prozesse dargestellt. Im Folgenden wurden die wesentlichen Punkte von Java EE vorgestellt und ein Ausblick auf Java EE 6 gegeben.

Verschiedene Migrationsarten von J2EE-Anwendungen wurden präsentiert und es wurde versucht sie in die Darstellung von Migrationen in Kapitel 2 einzuordnen. Anschließend wurde anhand einer Beispielanwendung die Migration einer Java-EE-Anwendung gezeigt. Die Frage nach der Möglichkeit einer Migration von J2EE-Anwendungen konnte damit beantwortet werden.

Das durch die Migration entstandene Neusystem wurde anhand verschiedener Metriken mit dem Altsystem verglichen, um herauszufinden, ob die von Sun angestrebte Reduktion der Komplexität durch Java EE 5 tatsächlich erreicht wurde. Die Hälfte der genutzten Metriken zeigten eine Verbesserung durch die Migration, die andere Hälfte zeigte eine Verschlechterung.

Die Reduktion der Komplexität kann also durch diese Arbeit nicht bestätigt werden. Allerdings lässt sich nicht ausschließen, dass andere Metriken eine Verringerung der Komplexität anzeigen würden. Auch könnte dieses Ergebnis an der migrierten Anwendung, dem BDG, liegen.

6.2 Ausblick

Aussagekräftigere Ergebnisse zur Komplexitätsreduktion setzen weitere Migrationen von anderen J2EE-Anwendungen und deren Vermessung voraus. Auch wurden in dieser Arbeit nur wenige Metriken genutzt. Eine größere Auswahl von Metriken könnte auch zu aussagekräftigeren Ergebnissen führen.

Weiterhin könnte die Möglichkeit einer (teil-)automatisierten Migration von J2EE-Anwendungen untersucht werden.

Literaturverzeichnis

[Bec04] BECK, Kent: *Extreme Programming Explained: Embrace Change*. 2. Boston, MA : Addison-Wesley Professional, 2004

[BK05] BAUER, Christian ; KING, Gavin: *Hibernate in Action*. 2. Greenwich, CT : Manning, 2005

[BLWG99] BISBAL, Jesús ; LAWLESS, Deirdre ; WU, Bing ; GRIMSON, Jane: Legacy Information Systems: Issues and Directions. In: *IEEE Software* 16 (1999), Nr. 5, S. 103–111

[BS95] BRODIE, Michael L. ; STONEBRAKER, Michael: *Migrating Legacy Systems: Gateways, Interfaces & the incremental Approach*. 1. San Francisco, CA : Morgan Kaufmann Publishers, Inc., 1995

[Che06] CHEN, Nicholas: *Convention over Configuration*. `http://softwareengineering.vazexqi.com/files/pattern.html`. Version: 2006, Abruf: 7.12.2008

[Chi08] CHINNICI, Robert: *Roberto Chinnici's Blog: Ease of development in the Java EE 6 Platform*. `http://weblogs.java.net/blog/robc/archive/2008/11/ease_of_develop.html`. Version: November 2008

[Cio08] CIOSTEK, Lukas: *Persönliche Mitteilungen*. Oktober 2008

[CK91] CHIDAMBER, Shyam R. ; KEMERER, Chris F.: Towards a metrics suite for object oriented design. In: *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*. New York, NY : ACM, 1991. – ISBN 0–201–55417–8, S. 197–211

[CK94] CHIDAMBER, Shyham R. ; KEMERER, Chris F.: A metrics suite for object oriented design. In: *IEEE Transactions on Software Engineering* 20 (1994), Nr. 6, S. 476–493

[Cla08] CLARK, Mike: *JDepend*. `http://clarkware.com/software/JDepend.html`. Version: 2008, Abruf: 7.12.2008

[CS08] CHINNICI, Roberto ; SHANNON, Bill: *Java^{TM} Platform, Enterprise Edition (Java EE) Specification, v6 – Early Draft*. `http://jcp.org/aboutJava/communityprocess/edr/jsr316/index.html`. Version: 2008, Abruf: 7.12.2008

[DLR06] DELISLE, Pierre ; LUEHE, Jan ; ROTH, Mark: *JavaServer Pages™ Specification Version 2.1*. http://jcp.org/aboutJava/communityprocess/final/jsr245/index.html. Version: 2006, Abruf: 7.12.2008

[EFB01] ELRAD, Tzilla ; FILMAN, Robert E. ; BADER, Atef: Aspect-oriented programming: Introduction. In: *Communications of the ACM* 44 (2001), Nr. 10, S. 29–32

[EJB06] EJB 3.0 EXPERT GROUP: *EJB Core Contracts and Requirements*. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html. Version: 2006, Abruf: 7.12.2008

[EJB08] EJB 3.1 EXPERT GROUP: *JSR 318: Enterprise JavaBeans™, Version 3.1 – EJB Core Contracts and Requirements*. http://jcp.org/aboutJava/communityprocess/edr/jsr318/index.html. Version: 2008, Abruf: 7.12.2008

[FH07] FISCHER, Peter ; HOFER, Peter: *Lexikon der Informatik*. Berlin : Springer, 2007

[For58] FORRESTER, Jay W.: Industrial Dynamics – A major breakthrough for decision makers. In: *Harvard Business Review* 36 (1958), Nr. 4, S. 37–68

[Fow] FOWLER, Martin: *MF Bliki: POJO*. http://www.martinfowler.com/bliki/POJO.html, Abruf: 7.12.2008

[Fow04] FOWLER, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. http://www.martinfowler.com/articles/injection.html. Version: 2004, Abruf: 7.12.2008

[GHJV96] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 1. Bonn u. a., 1996

[GJSB05] GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *The Java™ Language Specification*. Addison-Wesley, 2005

[Gra97] GRAHAM HAMILTON (HRSG.): *JavaBeans™*. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html. Version: 1997, Abruf: 7.12.2008

[GW05] GIMNICH, Rainer ; WINTER, Andreas: Workflows der Software-Migration. In: *Softwaretechnik-Trends* 25 (2005), Nr. 2, S. 22–24

[Hil07] HILDEBRANDT, Knut (Hrsg.): *HMD – Praxis der Wirtschaftsinformatik*. Bd. 257: *IT-Integration und Migration*. Heidelberg : dpunkt.verlag, 2007

[HS95] HENDERSON-SELLERS, Brian: *Object-oriented metrics: measures of complexity*. 1. Upper Saddle River, NJ : Prentice-Hall, Inc., 1995

[Jav08] JAVA PERSISTENCE 2.0 EXPERT GROUP: *JSR 317: Java^{TM} Persistence API, Version 2.0 – Public Review Draft*. http://jcp.org/aboutJava/communityprocess/edr/jsr317/index.html. Version: 2008, Abruf: 7.12.2008

[JBC+06] JENDROCK, Eric ; BALL, Jennifer ; CARSON, Debbie ; EVANS, Ian ; FORDIN, Scott ; HAASE, Kim: *The Java EE 5 Tutorial For Sun Java System Application Server 9.1*. Upper Saddle River, NJ u. a. : Addison Wesley, 2006

[Joh05] JOHNSON, Rod: J2EE development frameworks. In: *Computer* 38 (2005), Nr. 1, S. 107–110

[LPW97] LEE, Hau L. ; PADMANABHAN, V. ; WHANG, Seungjin: The Bullwhip Effect in Supply Chains. In: *Sloan Management Review* 38 (1997), Nr. 3, S. 93–102

[LSL02] LI, Michael ; SIMCHI-LEVI, David: *Beergame Guide*. http://beergame.mit.edu/guide.htm. Version: 2002, Abruf: 7.12.2008

[Mar94] MARTIN, R.: OO Design Quality Metrics-An Analysis of Dependencies. In: *Proc. Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA* Bd. 94, 1994

[Mar01] MARINESCU, Floyd: *The State of The J2EE Application Server Market – History, important trends and predictions*. http://www.theserverside.com/tt/articles/article.tss?l=State-Of-The-Server-Side. Version: 2001, Abruf: 7.12.2008

[Mar03] MARTIN, Robert C.: *Agile Software Development: principles, patterns and practices*. Upper Saddle River, NJ : Pearson Education/Prentice Hall, 2003

[McC76] MCCABE, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* 2 (1976), Dezember, Nr. 4, S. 308–320

[Mey97] MEYER, Bertrand: *Object-oriented software construction*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1997

[mig08] *migrate2glassfish: Migrate To GlassFish*. https://migrate2glassfish.dev.java.net/. Version: 2008, Abruf: 7.12.2008

[Mon07] MONZILLO, Ron: *Java Authentication SPI for Containers*. http://jcp.org/aboutJava/communityprocess/final/jsr196/index.html. Version: 2007, Abruf: 7.12.2008

[Mor07] MORDANI, Rajiv: *Java™ Servlet Specification Version 2.5 MR6*. http://jcp.org/aboutJava/communityprocess/mrel/jsr154/index2.html. Version: 2007, Abruf: 7.12.2008

[Par94] PARNAS, David L.: Software aging. In: *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA : IEEE Computer Society Press, 1994, S. 279–287

[PL08] PELEGRI-LLOPART, Eduardo: *The Aquarium: How do 14 (Millions) Sound? Status of GlassFish Downloads*. http://blogs.sun.com/theaquarium/entry/in_millions_14_status_of. Version: 2008, Abruf: 7.12.2008

[PRL07] PANDA, Debu ; RAHMAN, Reza ; LANE, Derek: *EJB 3 in Action*. 1. Greenwich, CT, USA : Manning Publications Co., 2007

[Rb99] ROMAN, Ed ; ÖBERG, Rickard: *The Technical Benefits of EJB and J2EE Technologies over COM+ and Windows DNA*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.4923&rep=rep1&type=pdf. Version: 1999, Abruf: 7.12.2008

[RGF+] RYMER, John R. ; GILPIN, Mike ; FULTON, Larry ; HEFFNER, Randy ; STONE, Jacqueline: *The Forrester Wave™: Application Server Platforms, Q3 2007*. http://www.forrester.com/Research/Document/Excerpt/0,7211,37138,00.html, Abruf: 7.12.2008

[Ros97] ROSENBERG, J.: Some Misconceptions About Lines of Code. In: *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*. Washington, DC : IEEE Computer Society, 1997, S. 137

[Roy70] ROYCE, W.W.: Managing the Development of Large Software Systems. In: *Proceedings of IEEE WESCON* Bd. 26, 1970

[RSL99] RYAN, V. ; SELIGMAN, S. ; LEE, R.: *Schema for Representing Java(tm) Objects in an LDAP Directory*. Version: 1999. http://www.rfc-editor.org/rfc/rfc2713.txt, Abruf: 7.12.2008

[Sha03] SHANNON, Bill: *Java™2 Platform Enterprise Edition Specification, v1.4*. http://jcp.org/aboutJava/communityprocess/maintenance/jsr151/index.html. Version: 2003, Abruf: 7.12.2008

[Sha06] SHANNON, Bill: *Java™ Platform, Enterprise Edition (Java EE) Specification, v5*. http://jcp.org/aboutJava/communityprocess/final/jsr244/index.html. Version: 2006, Abruf: 7.12.2008

[SHT05] SNEED, Harry M. ; HASITSCHKA, Martin ; TEICHMAN, Therese: *Software-Produktmanagement: Wartung und Weiterentwicklung bestehender Anwendungssysteme*. 1. Heidelberg : dpunkt.verlag, 2005

[Sin02] SINGH, I.: *Designing Enterprise Applications with the J2EE Platform*. Addison-Wesley Professional, 2002

[SM00] SUN MICROSYSTEMS, Inc.: *JSR 58: Java™2 Platform, Enterprise Edition 1.3 Specification*. http://www.jcp.org/en/jsr/detail?id=58. Version: 2000, Abruf: 7.12.2008

[SM07] SUN MICROSYSTEMS, Inc.: *Sun Java SystemApplication Server 9.1 Developer's Guide*. http://dlc.sun.com/pdf/819-3672/819-3672.pdf. Version: 2007, Abruf: 7.12.2008

[SM08] SUN MICROSYSTEMNS, Inc.: *Java EE at a Glance*. http://java.sun.com/javaee/index.jsp. Version: 2008, Abruf: 7.12.2008

[Sne99] SNEED, Harry M.: *Objektorientierte Softwaremigration*. 1. Reading, Mass. u. a. : Addison-Wesley, 1999

[Sun97] SUN MICROSYSTEMS, INC.: *Sun announces Java 2 Platform, Enterprise Edition*. Pressemitteilung. http://www.sun.com/smi/Press/sunflash/1997-04/sunflash.970402.4383.xml. Version: 1997, Abruf: 19.05.2008

[Sun01] SUN MICROSYSTEMS, INC.: *Enterprise JavaBeans™Specification, Version 2.0*. http://java.sun.com/products/ejb/docs.html. Version: 2001, Abruf: 7.12.2008

[Sun03] SUN MICROSYSTEMS, INC.: *Enterprise JavaBeans™Specification, Version 2.1*. http://java.sun.com/products/ejb/docs.html. Version: 2003, Abruf: 7.12.2008

[Sun06a] SUN MICROSYSTEMS, INC. ; MORDANI, Rajiv (Hrsg.): *Common Annotations for the Java™Platform™*. http://www.jcp.org/en/jsr/detail?id=250. Version: 2006, Abruf: 7.12.2008

[Sun06b] SUN MICROSYSTEMS, INC.: *Java Persistence API*. http://jcp.org/aboutJava/communityprocess/edr/jsr220/index.html. Version: 2006, Abruf: 7.12.2008

[Sun06c] SUN MICROSYSTEMS, INC. ; BURNS, Ed (Hrsg.) ; KITAIN, Roger (Hrsg.): *JavaServer™Faces Specification Version 1.2 - Rev A*. http://jcp.org/en/jsr/detail?id=252. Version: 2006, Abruf: 7.12.2008

[Sun07] SUN MICROSYSTEMS, INC.: *Sun Java System Application Server 9.1 Upgrade and Migration Guide*. http://dlc.sun.com/pdf/820-3850/820-3850.pdf. Version: 2007, Abruf: 7.12.2008

[Sun08a] SUN MICROSYSTEMS, INC.: *Java SE Technologies at a Glance*. http://java.sun.com/javase/technologies/index.jsp. Version: 2008, Abruf: 7.12.2008

[Sun08b] SUN MICROSYSTEMS, INC. ; BURNS, Ed (Hrsg.) ; KITAIN, Roger (Hrsg.): *JavaServer™ Faces Specification – Version 2.0 – Early Draft Review 2*. http://jcp.org/aboutJava/communityprocess/pr/jsr314/index.html. Version: 2008, Abruf: 7.12.2008

[Sun08c] SUN MICROSYSTEMS, INC.: *JSR 244: JavaTM Platform, Enterprise Edition 5 (Java EE 5) Specification*. http://jcp.org/en/jsr/detail?id=244. Version: 2008, Abruf: 7.12.2008

[Sun08d] SUN MICROSYSTEMS, INC.: *JSR 313: Java™ Platform, Enterprise Edition 6 (Java EE 6) Specification*. http://www.jcp.org/en/jsr/detail?id=313. Version: 2008, Abruf: 7.12.2008

[Sun08e] SUN MICROSYSTEMS, INC.: *JSR 316: Java™ Platform, Enterprise Edition 6 (Java EE 6) Specification*. http://www.jcp.org/en/jsr/detail?id=316. Version: 2008, Abruf: 7.12.2008

[Web08] WEB BEANS EXPERT GROUP: *JSR 299: Web Beans – Public Review*. http://jcp.org/aboutJava/communityprocess/edr/jsr299/index.html. Version: 2008, Abruf: 7.12.2008

[Wei03] WEISS, Joachim: *Der Brockhaus Naturwissenschaft und Technik*. Mannheim u. a. : F. A. Brockhaus and Spektrum, Akademischer Verlag, 2003

[Zül98] ZÜLLIGHOVEN, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz*. 1. Heidelberg : dpunkt.verlag, 1998

A Die Messergebnisse im Detail

In diesem Anhang sind alle Werte der in Kapitel 5 dargestellten Metriken aufgeführt.

Tabelle 14 enthält die Werte des Altsystems für *Weighted Methods per Class* und für *Lack of Cohesion in Methods* nach Chidamber und Kemerer (CK) sowie Henderson-Sellers (HS). In Tabelle 15 finden sich die entsprechenden Werte für das Neusystem.

Kein Eintrag bedeutet, dass es sich um ein Interface handelt und somit kein Wert ermittelbar ist.

In Tabelle 17 sind sämtliche Werte für das Altsystem, die nach der *Robert-Martin-Suite* ermittelt wurden, aufgeführt. Die entsprechenden Werte des Neusystems finden sich in Tabelle 18.

Die Bedeutung der Abkürzungen ist Tabelle 16 zu entnehmen.

In Tabelle 19 sind die Zeilenanzahlen für das Alt- und das Neusystem aufgeführt. Dabei steht in einer Zeile immer dieselbe Klassen für das Alt- und das Neusystem oder vergleichbare Klassen. Existiert keine vergleichbare Klasse, bleibt der Eintrag in dieser Zeile für das jeweilige System leer.

		LCOM	
Typ	WMC	HS	CK
BeerGameTableModel	20	75	7
BeerGameViewPopUp	33	85	0
BeerGameViewPopUp.CheckBoxListener	26		
BeerGameVisiModel	53	72	0
BeergameApplet	28	86	2
BeergameApplet.PopupListener	4		
GraphPlotter	95	89	1
GraphPlotter$(anonymous)	1		
SyncListener	13	67	0
TimeoutViewer	11	56	0
Tupel	7	0	0
DeliveryEJB	16	100	6
GameEJB	52	99	180
OrderEJB	22	0	0
RoleEJB	12	100	3
RolePK	7	0	0
UserEJB	40	94	24

Typ	WMC	LCOM HS	LCOM CK
Coarse	29	46	0
CoarseElement	14	83	9
DuplicateGameNameException	1		
GameManagerEJB	56	77	0
GameManagerException	5	0	0
GameSessionEJB	152	87	57
GameSessionEJB$(anonymous)	2		
GameSessionEJB$(anonymous)	2		
GameSessionException	5	0	0
RoleOccupiedException	1		
Round	39	81	12
Round$(anonymous)	4		
Round$(anonymous)	4		
SimpleComputerPlayer	1		
GameNameComparator	3		
NameSuggestorEJB	17	50	0
SequenceGenerator	1		
StoppedTimeComparator	7		
Event	7	100	10
FinishRoundAction	5	0	0
StartGameAction	6		
Timer	15	63	0
TimerAction	8	100	1
TimerListener	14	79	2
AbstractCommand	4	0	0
AbstractController	7	50	0
ControllerException	2	0	0
GameControl	2		
Menu	2		
Overview	2		
Public	2		
Register	2		
AbstractGameCommand	7	50	0
AdminCommand	10		
AppletDataCommand	6	0	0
ChangeDemandCommand	7	0	0
ChangeTOCommand	12		
DeleteCommand	4	0	0
EndGameCommand	4		

Typ	WMC	LCOM HS	LCOM CK
OrderCommand	7		
RefreshCommand	8		
RoundCommand	5	0	0
ShowTableCommand	6	0	0
Table	10	0	0
VisualCommand	6	0	0
AbstractMenuItem	4	100	1
FormMenuItem	3	0	0
LinkMenuItem	3	0	0
MenuCommand	4	0	0
ChooseGameCommand	17	0	0
CreateGameCommand	13	0	0
ListCommand	7	0	0
FinishedCommand	7	0	0
StatisticsCommand	9		
WelcomeCommand	7	0	0
SaveCommand	8	0	0
AbstractForm	9	0	0
CHDemandForm	11	50	0
DemandForm	10	50	0
GameCreationForm	48	93	88
GameForm	28	100	3
RegisterForm	26	0	0
LineLimiter	13	67	0
AllTests	1		
Systemtest	15		
EJBFactory	19	86	80
AllDBOTests	1		
GameTest	5	0	0
OrderTest	6	33	0
UserTest	0		
AllManagerTests	1		
GameManagerTest	18		
GameSessionTest	24	63	0
NameSuggestorTest	4		

Tabelle 14: Vollständige Ergebnisse für LCOM und WMC im Altsystem

Typ	WMC	LCOM HS	LCOM CK
BeerGameTableModel	20	75	7
BeerGameViewPopUp	33	85	0
BeerGameViewPopUp.CheckBoxListener	26		
BeerGameVisiModel	53	72	0
BeergameApplet	28	86	2
BeergameApplet.PopupListener	4		
GraphPlotter	95	89	1
GraphPlotter$(anonymous)	1		
SyncListener	13	67	0
TimeoutViewer	11	56	0
Tupel	7	0	0
Coarse	29	46	0
CoarseElement	14	83	9
DuplicateGameNameException	1		
GameManagerBean	46	81	0
GameManagerException	5	0	0
GameSessionBean	136	88	3
GameSessionBean$(anonymous)	1		
GameSessionBean$(anonymous)	1		
GameSessionException	5	0	0
RoleOccupiedException	1		
Round	39	81	12
Round$(anonymous)	3		
Round$(anonymous)	3		
SimpleComputerPlayer	1		
GameNameComparator	2		
NameSuggestorBean	11	100	3
SequenceGenerator	1		
StoppedTimeComparator	6		
Delivery	8	100	6
Game	46	97	456
Order	14	96	13
Role	5	100	6
RolePK	5	0	0
User	22	94	24
Event	7	100	10
FinishRoundAction	4	0	0
StartGameAction	5		
Timer	9	33	0

Typ	WMC	LCOM HS	LCOM CK
TimerAction	8	100	1
TimerListener	14	79	2
AbstractCommand	5	0	0
AbstractController	9	50	0
ControllerException	2	0	0
GameControl	3		
Menu	2		
Overview	2		
Public	2		
Register	2		
AbstractGameCommand	7	0	0
AdminCommand	10		
AppletDataCommand	6	0	0
ChangeDemandCommand	7	0	0
ChangeTOCommand	12		
DeleteCommand	4	0	0
EndGameCommand	4		
OrderCommand	7		
RefreshCommand	8		
RoundCommand	5	0	0
ShowTableCommand	6	0	0
Table	10	0	0
VisualCommand	6	0	0
AbstractMenuItem	4	100	1
FormMenuItem	3	0	0
LinkMenuItem	3	0	0
MenuCommand	4	0	0
ChooseGameCommand	17	0	0
CreateGameCommand	13	0	0
ListCommand	7	0	0
FinishedCommand	7	0	0
StatisticsCommand	9		
WelcomeCommand	7	0	0
SaveCommand	9	0	0
AbstractForm	9	0	0
CHDemandForm	11	50	0
DemandForm	10	50	0
GameCreationForm	48	93	88
GameForm	28	100	3

Typ	WMC	LCOM HS	LCOM CK
RegisterForm	26	0	0
LineLimiter	13	67	0
TestServlet	0		
AllTests	0		
Systemtest	15		
AllDBOTests	0		
GameTest	4		
OrderTest	5	0	0
AllManagerTests	0		
GameManagerTest	16	65	0
GameSessionTest	29	68	0
NameSuggestorTest	4	0	0

Tabelle 15: Vollständige Werte von LCOM und WMC für das Neusystem

C_t	Anzahl der Klassen
C_a	Anzahl der abstrakten Klassen und Interfaces
C_c	Anzahl der konkreten Klassen
C_a	Afferent Coupling
C_e	Efferent Coupling
A	Abstractness
I	Instability
D	Normalisierte Distance from the Main Sequence

Tabelle 16: Abkürzungen für die Robert-Martin-Suite

PAKET	C_t	C_a	C_c	C_a	C_e	A	I	D
beergame.client	11	0	11	0	0	0,00	0,00	1,00
beergame.ejb.dbo	16	10	6	9	2	0,62	0,18	0,19
beergame.ejb.managers	19	5	14	7	3	0,26	0,30	0,44
beergame.ejb.util	6	2	4	4	1	0,33	0,20	0,47
beergame.timeout	6	1	5	1	1	0,17	0,50	0,33
beergame.web.controller	8	2	6	5	5	0,25	0,50	0,25
beergame.web.controller.game	13	1	12	2	4	0,08	0,67	0,26
beergame.web.controller.menu	4	1	3	1	1	0,25	0,50	0,25
beergame.web.controller.overview	3	0	3	1	6	0,00	0,86	0,14
beergame.web.controller.publicpart	3	0	3	1	5	0,00	0,83	0,17
beergame.web.controller.register	1	0	1	1	4	0,00	0,80	0,20
beergame.web.form	6	1	5	3	0	0,17	0,00	0,83
beergame.web.util	1	0	1	2	0	0,00	0,00	1,00
test.beergame	2	0	2	0	8	0,00	1,00	0,00
test.beergame.ejb	1	0	1	2	2	0,00	0,50	0,50
test.beergame.ejb.dbo	4	0	4	1	4	0,00	0,80	0,20
test.beergame.ejb.managers	3	0	3	1	4	0,00	0,80	0,20
test.beergame.ejb.tools	1	0	1	1	1	0,00	0,50	0,50

Tabelle 17: Vollständige Ergebnisse der Robert-Martin-Suite für das Altsystem

PAKET	C_t	C_a	C_c	C_a	C_e	A	I	D
beergame.client	11	0	11	0	0	0,00	0,00	1,00
beergame.entities	6	0	6	8	0	0,00	0,00	1,00
beergame.ejb.managers	17	3	14	6	3	0,18	0,33	0,49
beergame.ejb.util	5	1	4	4	1	0,20	0,20	0,60
beergame.timeout	6	1	5	1	1	0,17	0,50	0,33
beergame.web.controller	8	2	6	5	5	0,25	0,50	0,25
beergame.web.controller.game	13	1	12	2	4	0,08	0,67	0,26
beergame.web.controller.menu	4	1	3	1	1	0,25	0,50	0,25
beergame.web.controller.overview	3	0	3	1	6	0,00	0,86	0,14
beergame.web.controller.publicpart	3	0	3	1	5	0,00	0,83	0,17
beergame.web.controller.register	1	0	1	1	4	0,00	0,80	0,20
beergame.web.form	6	1	5	3	0	0,17	0,00	0,83
beergame.web.util	1	0	1	2	0	0,00	0,00	1,00
test	1	0	1	0	1	0,00	1,00	0,00
test.beergame	2	0	2	0	4	0,00	1,00	0,00
test.beergame.ejb.dbo	3	0	3	0	3	0,00	1,00	0,00
test.beergame.ejb.managers	3	0	3	0	4	0,00	1,00	0,00
test.beergame.ejb.tools	1	0	1	0	2	0,00	1,00	0,00

Tabelle 18: Vollständige Ergebnisse der Robert-Martin-Suite für das Neusystem

Klasse/Interface im Altsystem	Lines of Code	Klasse/Interface im Neusystem	Lines of Code
BeergameApplet	283	BeergameApplet	283
BeerGameTableModel	142	BeerGameTableModel	142
BeerGameViewPopUp	265	BeerGameViewPopUp	265
BeerGameVisiModel	291	BeerGameVisiModel	291
GraphPlotter	619	GraphPlotter	619
SyncListener	162	SyncListener	162
TimeoutViewer	73	TimeoutViewer	73
Tupel	53	Tupel	53
Delivery	36	Delivery	88
DeliveryEJB	105		
DeliveryHome	48		
Game	196		
GameEJB	406	Game	456
GameHome	59		
Order	68		
OrderEJB	175	Order	140
OrderHome	59		
Role	62		
RoleEJB	88	Role	98
RoleHome	32		
RolePK	65	RolePK	42
User	55		
UserEJB	254	User	186
UserHome	42		
Coarse	279	Coarse	279
CoarseElement	178	CoarseElement	178
ComputerPlayer	21	ComputerPlayer	21
DuplicateGameNameException	17	DuplicateGameNameException	17
GameManager	91	GameManager	84
GameManagerEJB	399	GameManagerBean	374
GameManagerException	63	GameManagerException	63
GameManagerHome	15		
GameSession	188	GameSession	194
GameSessionEJB	1066	GameSessionBean	1011
GameSessionException	63	GameSessionException	63
GameSessionHome	21		
RoleOccupiedException	13	RoleOccupiedException	13
Round	320	Round	313
SimpleComputerPlayer	25	SimpleComputerPlayer	25
GameNameComparator	36	GameNameComparator	31

Klasse/Interface im Altsystem	Lines of Code	Klasse/Interface im Neusystem	Lines of Code
NameSuggestor	34	NameSuggestor	34
NameSuggestorEJB	100	NameSuggestorBean	102
NameSuggestorHome	15		
SequenceGenerator	23	SequenceGenerator	23
StoppedTimeComparator	64	StoppedTimeComparator	45
Event	121	Event	121
FinishRoundAction	58	FinishRoundAction	54
StartGameAction	61	StartGameAction	57
Timer	180	Timer	97
TimerAction	126	TimerAction	126
TimerListener	180	TimerListener	182
AbstractCommand	68	AbstractCommand	78
AbstractController	141	AbstractController	166
ControllerException	38	ControllerException	37
GameControl	107	GameControl	119
Menu	38	Menu	38
Overview	56	Overview	56
Public	83	Public	81
Register	41	Register	41
AbstractGameCommand	111	AbstractGameCommand	100
AdminCommand	177	AdminCommand	181
AppletDataCommand	90	AppletDataCommand	92
ChangeDemandCommand	157	ChangeDemandCommand	140
ChangeTOCommand	194	ChangeTOCommand	197
DeleteCommand	93	DeleteCommand	77
EndGameCommand	88	EndGameCommand	90
OrderCommand	136	OrderCommand	135
RefreshCommand	208	RefreshCommand	178
RoundCommand	100	RoundCommand	99
ShowTableCommand	96	ShowTableCommand	98
Table	106	Table	106
VisualCommand	118	VisualCommand	120
AbstractMenuItem	63	AbstractMenuItem	63
FormMenuItem	60	FormMenuItem	60
LinkMenuItem	51	LinkMenuItem	51
MenuCommand	108	MenuCommand	108
ChooseGameCommand	246	ChooseGameCommand	204
CreateGameCommand	330	CreateGameCommand	282
ListCommand	170	ListCommand	134
FinishedCommand	143	FinishedCommand	122

Klasse/Interface im Altsystem	Lines of Code	Klasse/Interface im Neusystem	Lines of Code
StatisticsCommand	150	StatisticsCommand	128
WelcomeCommand	122	WelcomeCommand	107
SaveCommand	200	SaveCommand	183
AbstractForm	110	AbstractForm	110
CHDemandForm	112	CHDemandForm	112
DemandForm	105	DemandForm	105
GameCreationForm	522	GameCreationForm	522
GameForm	222	GameForm	222
RegisterForm	203	RegisterForm	203
LineLimiter	156	LineLimiter	156
AllTests	23	AllTests	15
Systemtest	346	Systemtest	346
EJBFactory	145		
AllDBOTests	23	AllDBOTests	9
GameTest	45	GameTest	34
OrderTest	84	OrderTest	62
UserTest	12		
AllManagerTests	18	AllManagerTests	12
GameManagerTest	217	GameManagerTest	226
GameSessionTest	463	GameSessionTest	520
NameSuggestorTest	23	NameSuggestorTest	28
		TestServlet	19

Tabelle 19: Vollständige Ergebnisse für LOC im Alt- und Neusystem

B Inhalt der beigelegten CD-ROM

Dieser Arbeit ist eine CD-ROM beigelegt, die folgenden Inhalt hat:

- Auflistung aller Dateien

  Die CD-ROM enthält im Wurzelverzeichnis eine Datei namens *index.html*. Diese Datei verlinkt alle anderen Dateien auf der CD.

- Diese Diplomarbeit

  Die Datei *Migration von J2EE-Anwendungen.pdf* im Wurzelverzeichnis der CD-ROM.

- Onlinequellen

  Das Exemplar für die beiden Gutachter enthält alle Onlinequellen im Ordner *Quellen*. Die darin enthaltenen Ordner haben das Zitierkürzel der jeweiligen Quelle als Namen. Die eigentliche Quelle liegt in diesen Unterordnern als HTML- oder PDF-Datei.

  Im Bibliotheksexemplar sind diese Quellen aus urheberrechtlichen Gründen nicht enthalten.

- Altsystem Altsystem.zip: Das Altsystem als Eclipse-Projekt im Ordner Altsystem.
  - Mit JDepend gemessen Metriken für das Altsystem
  - Metrics Plugin gemessen Metriken für das Altsystem

- Neusystem Neusystem.zip: Das Neusystem als Eclipse-Projekt im Ordner Neusystem.
  - Mit JDepend gemessen Metriken für das Neusystem
  - Metrics Plugin gemessen Metriken für das Neusystem

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen übernommen wurden, sind als solche kenntlich gemacht. Alle Quellen, die dem World Wide Web entnommen oder in einer sonstigen digitalen Form verwendet wurden, sind der Arbeit beigefügt.