Eberhardt Karls Universität Tübingen – Wilhelm-Schickard-Institut für Informatik

Transaktionsverarbeitung in J2EE Systemen

Diplomarbeit

betreut von

Professor Dr. Wilhelm Spruth

vorgelegt von

Oliver Raible

Tübingen 2004

Zusammenfassung

Der Einsatz von Java bzw. J2EE für die Gewährleistung der Transaktionssicherheit wird in der Literatur als problematisch angesehen, da durch den Einsatz des Multi-Threading-Modells die Isolation von Transaktionen untereinander nicht gewährleistet werden kann. Ergebnisse von Transaktionen können somit verfälscht werden. Ebenso ist die Fehlerfortpflanzung bei kritischen Ausnahmesituationen in einzelnen Transaktionen, auf alle momentan ausgeführten Transaktionen, ein Kritikpunkt. Darüber hinaus scheint die Entwicklung von stabilen und zuverlässigen Anwendungen auf Basis der J2EE-Plattform durch ihre Komplexität und den verteilten Entwicklungsprozess sehr fehleranfällig.

Für die Problematik der Isolation existieren verschiedene Lösungsansätze. Der Lösungsansatz, die Isolation mittels Betriebssystemprozessen sicherzustellen, wird in der *Persistent Reusable JVM (PRJVM)* von IBM angewendet [Borm01]. Dieser Ansatz wurde im Rahmen einer Diplomarbeit schon näher untersucht [Bey04]. Der Ansatz, die Isolation mittels *Classloader* sicherzustellen, wird in fast allen J2EE-Implementierungen, zusammen mit einem Multi-Threading-Ansatz, eingesetzt. Die Isolation durch Modifikation der *Java Virtual Machine (JVM)* sicherzustellen [Cza00], birgt zusätzlich zum *Classloader*-Ansatz den Vorteil, dass die Isolation automatisch von der JVM bereitgestellt wird.

Allen Ansätzen gemein ist, dass die Problematik, stabile und zuverlässige Anwendungskomponenten zu entwickeln, völlig vernachlässigt wird.

In dieser Arbeit wird ein alternativer Ansatz für die Lösung der geschilderten Problematik untersucht, welcher sowohl Stabilität und Zuverlässigkeit der J2EE-Infrastruktur als auch der Anwendungskomponenten hinsichtlich Transaktionsverarbeitung sicherstellen soll. Aus Ermangelung einer existierenden Testinfrastruktur für die gestellten Anforderungen, wurde hierzu im Rahmen dieser Arbeit das *Transaction Testsystem (TTS)* entworfen und implementiert. Das TTS simuliert Unternehmensprozesse und komplexe Fehlersituationen bei Nebenläufigkeit. Es ist in der Lage verschiedene Transaktionskombinationen mit frei zu definierender Basislast unter Fehlereinwirkung, bis hin zu fatalen Systemfehlern, reproduzierbar zu machen und kontrolliert durchzuführen und auszuwerten. Die Testszenarien und Fehlerfälle können dem System einfach hinzugefügt werden um damit einem realen Anwendungsszenario beliebig Nahe zu kommen. Die zu testende J2EE-Anwendung ist austauschbar, somit kann das TTS auch für Tests von Anwendungen in realen Softwareprojekten eingesetzt werden. Es wird damit möglich, reale Problematiken (aus Transaktionsverarbeitung oder genereller Nebenläufigkeit bedingt) kontrolliert und reproduzierbar innerhalb eines Testsystems zu untersuchen. Darüber hinaus kann unter der Prämisse von stabilen und zuverlässigen Anwendungskomponenten die Zuverlässigkeit der verwendeten J2EE-Infrastruktur überprüft werden.

Um Aussagen über die Eignung des TTS für die Lösung der geschilderten Problematik zu erhalten, wurden Untersuchungen mit dem JBoss Applicationserver (mit MaxDB als Datenbank) und IBM Websphere (mit DB2 als Datenbank) durchgeführt. MaxDB (früher SapDB) ist

eine Opensource Datenbank, welche von der Firma MySql AB, anlässlich einer strategischen Partnerschaft mit SAP, übernommen wurde. Die detaillierten Ergebnisse sind in Kapitel 7.2 dargestellt und zeigen, dass das TTS für das Aufspüren von Anwendungs-, Integrations- und Konfigurationsfehlern geeignet zu sein scheint. Ebenso geht hervor, dass die J2EE-Infrastrukturen mit derartig getesteten Anwendungskomponenten die Transaktionssicherheit, selbst bei fatalen Fehlern (z.B. Systemabsturz), gewährleisten können. Der Lösungsansatz, das TTS im Entwicklungsprozess einzusetzen, um die Transaktionssicherheit des J2EE-Systems in der Produktion zu optimieren, erscheint demnach tragfähig zu sein.

Natürlich konnte im Rahmen dieser Arbeit nur ein kleiner Teil an Test- und Fehlerszenarien betrachtet werden, welche in einem Produktionssystem vorkommen können. Eine Verallgemeinerung der positiven Ergebnisse über die betrachtete Teststellung hinaus, muss noch durch weitere Untersuchungen sichergestellt werden. Die erhaltenen Ergebnisse sind jedoch sehr viel versprechend und durch Hinzunahme weiterer Testfälle kann in aufbauenden Untersuchungen den realen Produktionsszenarien beliebig nahe gekommen werden. Mit diesem Ansatz ist es denkbar, Anwendungen auf J2EE-Systemen mit einem ähnlich hohen Grad an Sicherheit ausführen zu können, wie es unter bewährten Transaktionsmonitoren, beispielsweise CICS und Tuxedo, möglich ist.

Danksagung

Mein Dank gilt allen Personen, die mich während dieser Diplomarbeit und meines Studiums unterstützt und gefördert haben. Für das entgegengebrachte Verständnis und die Geduld während des Studiums und dieser abschließenden Diplomarbeit, bedanke ich mich besonders bei meiner Freundin, Rosemarie Zaharanski.

Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. Wilhelm G. Spruth für die Betreuung der Diplomarbeit, Herrn Prof. Dr. W. Rosenstiel für die Möglichkeit der Durchführung der Diplomarbeit am Lehrstuhl für Technische Informatik und Frau Sabrowski, vom Prüfungsamt Informatik, für Ihre Geduld.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Die Einführung des Konzeptes von Transaktionen sowie die Entwicklung von Transaktionsmonitoren zur Durchführung von hochperformanter, effizienter und zuverlässiger Transaktionsverarbeitung stellen Meilensteine auf dem Weg zur Entwicklung einer stabilen und verlässlichen Plattform, für unternehmenskritische Anwendungen, dar. Die Transaktionsmonitore haben eine lange Entwicklungsgeschichte hinter sich. Der Transaktionsmonitor CICS wurde beispielsweise 1968 [Herr03] entworfen und bis heute kontinuierlich weiterentwickelt, um Fehler zu beheben, potentielle Schwachstellen zu optimieren und die Fehlerbehandlungen auszubauen. Transaktionsmonitore, wie CICS, Tuxedo oder auch SAP R/3, gelten heute als stabil und zuverlässig, aber auch als proprietäre Insellösungen, ohne das Potential als eine unternehmensweite, offene Integrationsplattform, für Frontend- und Backend-Systeme jeglicher Art, zu dienen. Die J2EE-Plattform als Dachspezifikation für das Programmiermodell der *Enterprise JavaBeans (EJB)*, die Transaktionsverarbeitung über die *Java Transaction API (JTA)* und die Anbindung von externen Ressourcen über die *Java Connector Architecture (JCA)*, definiert ein solches System. Es wurde 1998 von Sun Microsystems Inc. in der Version 1.0 veröffentlicht und liegt mittlerweile in der Version 1.4 als offen gelegter etablierter Standard vor.

> *We can ask customers to set aside their freedom of choice and preferences...and all of us -- everyone, everywhere -- move to one architecture provided by, priced by and controlled by one company. Or: We can embrace open industry standards.*
>
> ***IBM CEO Lou Gerstner (Comdex 1995)***

Die Akzeptanz und Verbreitung der J2EE-Plattform wurde maßgeblich von den Firmen Sun, IBM und Bea beeinflusst, welche schnell Implementierungen der Spezifikation als Produkte zugänglich machten. Heutzutage wird der Markt im Bereich J2EE Applicationservern von IBM Websphere und Bea Weblogic dominiert, im Opensource-Bereich hat sich der JBoss Applicationserver etabliert. Angesichts des zunehmenden Einsatzes von J2EE-Applicationservern für unternehmenskritische Anwendungen stellt sich die Frage, wie es um den Entwicklungsstand der zentralen Komponenten, beispielsweise der Transaktionsverarbeitung, bestellt ist. Anders als zu Transaktionsmonitoren existiert sehr wenig Literatur, welche die Transaktionsverarbeitung in J2EE oder in J2EE-Produkten untersucht und Aussagen über die Qualität und Einsetzbarkeit von J2EE-Produkten gibt. Obwohl die J2EE-Plattform eine offen gelegte standardisierte Spezifikation ist, definiert sie nur ein minimales Rahmenwerk und Interak-

tionsschnittstellen. Es ist weitgehend den Herstellern der J2EE-Produkte überlassen, wie sie Spezifikationslücken handhaben, Funktionalitäten intern umsetzen und Stabilität sicherstellen. Das J2EE-Produkt stellt sich damit weitgehend als *Blackbox* dar.

Der Einsatz von Java bzw. J2EE für die Transaktionsverarbeitung wird in der Literatur als problematisch angesehen, da durch den Einsatz des Multi-Threading-Modells die Isolation der Transaktion nicht gewährleistet werden kann und damit Überreste von vorherigen Transaktionen die Ergebnisse weiterer Transaktionen verfälschen. Ebenso ist die Fehlerfortpflanzung, bei kritischen Fehlern in einzelnen Transaktionen, auf alle momentan ausgeführten Transaktionen ein Kritikpunkt. Darüber hinaus deuten viele Erfahrungen aus Praxis-Projekten darauf hin, dass die Entwicklung von stabilen und zuverlässigen Anwendungen problematisch ist, da die Komplexität und der verteilte Entwicklungsprozess der J2EE-Plattform schwer beherrschbar ist.

Zusammengefasst herrscht eine Mischung aus negativen Indikatoren auf der einen Seite und fehlendem Wissen und Transparenz auf der anderen Seite vor. Mit dieser Ausgangslage stellen sich folgende Fragen, welche in dieser Arbeit untersucht werden sollen:

1. Wie funktioniert die *Blackbox* Transaktionsverarbeitung von J2EE-Systemen? Welche Anforderungen verlangt die J2EE-Spezifikation und wie erfolgt die Umsetzung in konkreten J2EE-Produkten?
2. Was gefährdet die Transaktionssicherheit bei J2EE-Systemen? Existieren Lösungsansätze für diese Problematik und wie funktionieren sie?
3. Wie steht es um das Thema Transaktionssicherheit bei J2EE-Produkten? Wie lässt sich eine Aussage zu der Transaktionssicherheit eines J2EE-Produktes machen?
4. Mit Sicherheit hat jedes Unternehmen und jede Behörde unterschiedliche Anforderungen an ein J2EE-Produkt sowohl aus fachlicher, technischer als auch wirtschaftlicher Sicht. Ist es möglich eine Form von Vergleichsmetrik zu schaffen, welche eine Evaluierung von J2EE-Produkten erleichtern kann, um Aussagen über Eignung eines J2EE-Produktes für bestimmte Aufgaben zu treffen und somit eine Produktentscheidung mit mehr Qualität und Substanz zu erhalten?
5. Die Ausführung von fehlerhaft geschriebenen J2EE-Anwendungen kann das gesamte J2EE-System beeinflussen. Wie lässt sich ein unerwünschtes Verhalten von J2EE-Anwendungen hinsichtlich Transaktionsverarbeitung vorbeugen?
6. Der verteilte Entwicklungsprozess von J2EE birgt neue Komplexität und neue Fehlerquellen bei der Softwareentwicklung. Wie lässt sich das Risiko bei der Entwicklung von J2EE-Anwendungen reduzieren?

1.2 Themenabgrenzung

Die Transaktionsverarbeitung in der J2EE-Spezifikation und in konkreten J2EE-Produkten ist ein sehr vielschichtiges und komplexes Thema. Um den Rahmen nicht zu sprengen, wurde sich auf die Interaktion der Komponenten der Transaktionsverarbeitung in J2EE-Systemen und die Gewährleistung von Transaktionssicherheit konzentriert. Obwohl Performanz ein wichtiges Thema bei der Transaktionsverarbeitung ist, spielt es in diesen Untersuchungen eine untergeordnete Rolle. Die Performanzdaten, welche in den Tests gesammelt wurden, sollen nur qualitative Tendenzen andeuten. Für die Erzielung von quantitativen Performanzaussagen müssen zusätzliche Tests durchgeführt werden. Des weiteren wurden Tests ausschließlich mit lokalen Transaktionen auf einer Datenquelle durchgeführt.

Der Bereich der *Business Transactions*, mit seinem Einfluss auf die Anforderungen von Transaktionsverarbeitung und die Definition von Transaktionssicherheit, wird nur kurz angesprochen. Der Fokus liegt auf dem herkömmlichen Verständnis für Transaktionen und dem Transaktionsmodell der *Flat Transaction*.

1.3 Aufbau der Arbeit

Die Arbeit orientiert sich an den in Kapitel 1.1 aufgeworfenen Fragestellungen. Als Grundlage für alle weiteren Untersuchungen werden in Kapitel 2 zunächst die Konzepte moderner Transaktionsverarbeitung vorgestellt sowie ihre Motivation, ihr Nutzen und ihre Wirkungsweise erläutert. Es werden die ACID-Kriterien detailliert beschrieben (Kapitel 2.1). Aus den Problemen, welche beim konkurrierenden Zugriff auf Daten entstehen (Kapitel 2.2), werden die Synchronisierungsverfahren abgeleitet und detailliert vorgestellt (Kapitel 2.3). Danach wird das Konzept der verteilten Transaktionen erläutert und die Auswirkungen auf die Festschreibung (*Commit*) der Daten einer Transaktion beschrieben (Kapitel 2.4). Als letztes werden die Transaktionen in Typen kategorisiert und die Anforderungen gegenüber den ACID-Kriterien miteinander verglichen (Kapitel 2.5).

In den Kapiteln 3 und 4 wird die Funktionsweise von Transaktionsverarbeitung in der J2EE-Spezifikation und in den konkreten J2EE-Produkten (Frage 1 aus Kapitel 1.1) untersucht. Zunächst wird auf die J2EE-Spezifikation an sich eingegangen (Kapitel 3). Nach einer kurzen Vorstellung, was hinter dem Begriff J2EE steht (Kapitel 3.1), werden die beteiligten Komponenten und Spezifikationen vorgestellt (Kapitel 3.2 bis 3.4). Danach wird das Zusammenspiel der vorgestellten Komponenten bei einem Anwendungsszenario detailliert untersucht (Kapitel 3.5).

Die gewonnene Kenntnis, um die vorgeschriebene Funktionsweise für die Transaktionsverarbeitung in J2EE, liefert im folgenden die Basis, um die *Blackbox* der konkreten Implementierung der J2EE-Produkte zu entschlüsseln (Kapitel 4). Es wurden in dieser Arbeit zwei J2EE-Produkte betrachtet. Zunächst wird der Opensource-Applicationserver JBoss vorgestellt sowie das konkrete Zusammenspiel seiner Komponenten, anhand des schon in Kapitel 3 verwendeten Anwendungsszenarios erläutert (Kapitel 4.1). Danach wird dasselbe Verfahren für den kommerziellen Applicationserver Websphere von IBM angewendet (Kapitel 4.2).

Als nächstes werden die Problematiken und Lösungsansätze für die Gewährleistung von Transaktionssicherheit in J2EE-Systemen untersucht (Frage 2 aus Kapitel 1.1). Es wird zunächst definiert, was der Begriff Transaktionssicherheit bedeutet (Kapitel 5.1) und wie es zu einer Gefährdung der Transaktionssicherheit kommt (Kapitel 5.2). Danach werden die unterschiedlichen Lösungsansätze vorgestellt (Kapitel 5.3 bis 5.6) und in Zusammenhang gebracht (Kapitel 5.7).

Mit dem Wissen um die Problematik von Transaktionssicherheit wird der Frage nachgegangen, wie es möglich ist, Aussagen über die Transaktionssicherheit eines J2EE-Produktes zu machen (Frage 3 aus Kapitel 1.1), dies sogar auf die J2EE-Anwendungsentwicklung zu erweitern (Frage 5 aus Kapitel 1.1) sowie eine Vergleichsmetrik für J2EE-Produkte zu schaffen (Frage 4 aus Kapitel 1.1). Eine Problemanalyse (Kapitel 6.1) führt zu dem Entwurf eines Testsystems für Transaktionsverarbeitung auf Basis von J2EE (Kapitel 6.2). Es wird das Datenmodell (Kapitel 6.3) und die Konsistenzbedingungen (Kapitel 6.4) vorgestellt. Danach wird eine Detailmetrik für die Auswertung der Testfälle definiert sowie daraus eine Vergleichsmetrik für die Transaktionseigenschaften von J2EE-Produkten abgeleitet (Kapitel 6.5).

Die Testdurchführung und die Ergebnisse werden in Kapitel 7 dargestellt. Die Testdurchführung sowie Testaufbau und verwendete Testszenarien werden in Kapitel 7.1 erläutert. Die Ergebnisse der Tests werden nach Relevanz für die J2EE-Infrastruktur (Kapitel 7.2.1) und nach Relevanz für die Anwendungstests (Kapitel 7.2.2) gegliedert und anschließend diskutiert. Die Erfahrungen aus der Entwicklung des Testsystems und der Durchführung der Anwendungstests gibt eine Antwort auf die Möglichkeit der Risikominimierung bei der Entwicklung von J2EE-Anwendungen (Frage 6 aus Kapitel 1.1). Abschließend wird eine Zusammenfassung der Thematik erstellt (Kapitel 8.1) und Schlussfolgerungen daraus abgeleitet (Kapitel 8.2). Es folgt ein Ausblick auf weiterführende Themen, welche noch untersucht werden müssen (Kapitel 8.3).

Im Anhang befinden sich Auszüge aus Konfigurationsdateien welche für die Konfiguration der betrachteten J2EE-Produkte benutzt wurden (Anhang A). Die detaillierten Ergebnisse der Anforderungsanalyse an das Testsystem sind in Anhang B dargelegt. Die Beschreibung des Entwurfs des TTS sowie des Zusammenspiels der Komponenten, ist in Anlage C zu finden. In Anhang D werden der Deploymentprozess und die Konfiguration der J2EE-Infrastruktur für das TTS vorgestellt.

1.4 Konventionen

Diese Arbeit orientiert sich an den folgenden typographischen Konventionen:

`Nichtproportionalschrift`
: Für die Darstellung des Inhalts von Konfigurationsdateien, Elementen aus Konfigurationsdateien (z.B. Schlüssel), Darstellung von Quellcode sowie Klassennamen und Dateinamen.

`Nichtproportionalschrift Kursiv`
: Für die Darstellung von Methoden-Aufrufen auf Klassen, sowie Klassenattribute.

Kursiv
: Für englischsprachige Eigennamen, Namen von Entitäten, oder Fachbegriffe (außer in Überschriften und Gliederungen), welche spezifisch für dieses Thema sind und als nicht allgemein gebräuchlich erachtet werden.

Es werden im Text Großbuchstaben für konkrete Instanzen verwendet (z.B. ...die Transaktion T1 wird...). Kleinbuchstaben werden bei Variablen verwendet (z.B. ...für beliebige Transaktionen t existieren...). Die Zugehörigkeit von Feldern zu Entitäten wird mit „::“ gekennzeichnet (z.B. *District::newOrderNumber*).

1.5 Terrific Housewares AG

Um die vielfältigen Szenarien der Transaktionsverarbeitung zu untersuchen und die Thematik greifbarer zu machen, soll eine fiktive, idealisierte Firma zur Herstellung und zum Vertrieb von Haushaltswaren dienen. Die *Terrific Housewares AG* ist eine selbständige, deutsche Tochter des amerikanischen Mutterkonzern *Terrific Things Inc*. Die *Terrific Housewares AG* wurde 1957 in Stuttgart gegründet, um der ständig wachsenden Nachfrage nach qualitativ hochwertigen Haushaltswaren im vom Wirtschaftswunder aufgerüttelten Deutschland nachzukommen. Die Firma wurde in den folgenden Jahren zu einem Synonym für Qualität, Zuverlässigkeit, Innovation und einem hervorragenden Preis-/Leistungsverhältnis. Dies war der Motor für die enorme Expansion des Stuttgarter Unternehmens in den folgenden Jahren, welche Filialgründungen in vielen weiteren Bundesländern zur Folge hatte. Schon früh wurden Computersysteme zur Handhabung des Bestellwesens und der Lagerhaltung eingesetzt. Heute präsentiert sich die *Terrific Housewares AG* als modernes und erfolgreiches Unternehmen für

Haushaltswaren jeglicher Art, mit einer Palette von 100.000 Einzelprodukten. Das Unternehmen beliefert ausschließlich Großabnehmer und verfügt über eine Anzahl von geographisch verteilten Verkaufsbereichen und Warenlagern. Jedes Warenlager verwaltet separate Bestände für die vertriebenen Produkte und deckt den Grundbedarf von zehn Verkaufsbereichen ab. Jeder Vertriebsbereich bedient 3.000 Kunden (Abbildung 1).

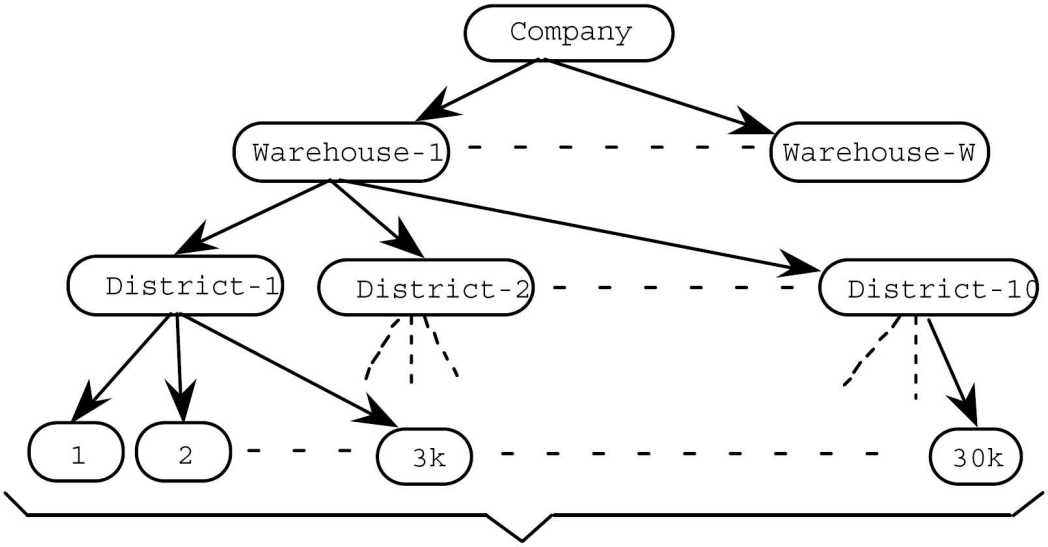

Abbildung 1 Struktur der Terrific Housewares AG (Quelle: [TPCC03])

Kunden tätigen ihre Bestellungen per Telefonanruf bei ihrem Kundenberater oder per Telefax. Ein Kundenberater benutzt das hauseigene Warenwirtschaftssystem *Terrific ERP for Housewares (TERPH)* um Aufträge, Lieferungen und Bezahlungen zu verwalten. Die Hauptfunktionalitäten von TERPH sind die Neuanlage von Aufträgen, die Abfrage des Auftragsstatus zu einem Kunden und die Verwaltung von Preisen zu den Produkten. Darüber hinaus wird TERPH auch dazu benutzt, um Kundenzahlungen zu verwalten und die Auslieferung von Waren zu disponieren. Im speziellen müssen die Warenbestände für die regionalen Warenlager kontrolliert werden, um frühestmöglich Lieferschwierigkeiten zu identifizieren und Ergänzungslieferungen aus benachbarten Warenlagern zu organisieren. Durch statistische Auswertungen konnte ermittelt werden, dass ein Auftrag im Durchschnitt zehn Auftragspositionen (sog. Auftragszeilen) besitzt. Von den für eine Auslieferung benötigten Produkten kann 99 % aus dem eigenen regionalen Warenlager befriedigt werden, nur 1 % muss von einem anderen Warenlager zu geliefert werden.

TERPH ist eine betriebswirtschaftliche Anwendung, die für seine Transaktionsverarbeitung einen eingekauften Transaktionsmonitor verwendet. Das Datenmodell von TERPH ist etwas vereinfacht in Abbildung 2 dargestellt.

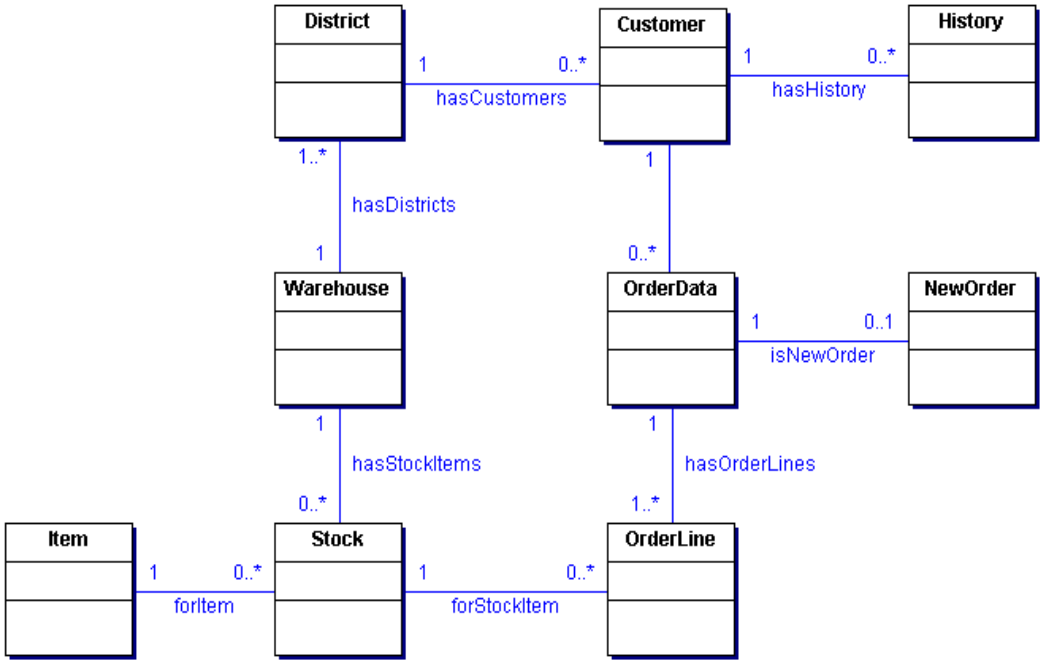

Abbildung 2 Datenmodell von TERPH

In jüngster Vergangenheit wurde der *Terrific Housewares AG* eine besondere Ehre zuteil. Das Unternehmen wurde 2003 vom *Transaction Processing Performance Council (TPC)* als Vorlage für die Erarbeitung der TPC-C Spezifikation gewählt. Die TPC ist eine gemeinnützige Unternehmung, welche das Ziel verfolgt, Benchmarks für Transaktionsverarbeitung und Datenbanken zu definieren und der Industrie objektive und überprüfbare Performanzdaten zu liefern. Der Benchmark TPC-C (definiert in [TPCC03]) simuliert die Transaktionsverarbeitung eines Unternehmens, welche im Schwerpunkt mit einer Vielzahl von Aufträgen mittlerer Größe konfrontiert wird.

2 Konzepte der Transaktionsverarbeitung

2.1 Was sind Transaktionen?

Trans|ak|ti|on, die; -, -en [spätlat. transactio]
= Vollendung, Abschluss, Übereinkunft, zu lat. Transactum
Quelle: Duden

Das Konzept der Transaktionen ist ein wichtiges Paradigma der Informatik für die Konstruktion von zuverlässigen und stabilen Anwendungen, welche konkurrierenden Zugriff auf gemeinsame Daten benötigen. Dieses Konzept wurde zuerst in betriebswirtschaftlichen Anwendungen eingesetzt, um Informationen in zentralen Datenbanken zu sichern. Später wurde das Konzept um die verteilten Transaktionen und im Zuge von *Web Services* durch die Einführung der *Business Transactions* erweitert. Ohne das Transaktionskonzept wäre der Anwendungsentwickler für die korrekte Handhabung von gemeinsam genutzten Daten verantwortlich. Eine äußerst komplexe und fehlerträchtige Aufgabe, vor allem wenn Software- und Hardwareressourcen effizient genutzt werden sollen. Heutzutage ist allgemein anerkannt, dass Transaktionen der Schlüssel sind, um zuverlässige verteilte Anwendungen zu bauen.

Eine Transaktion kann als eine Interaktion mit einem System betrachtet werden, welche den Zustand des Systems ändert. Während die Interaktion andauert, können beliebige Ereignisse die Interaktion unterbrechen und damit die Zustandsänderung unkomplett lassen. Das System befindet sich damit in einem inkonsistenten und somit unerwünschtem Zustand. Jede Änderung am Systemzustand innerhalb einer Transaktionsgrenze hat sicherzustellen, dass die Systemzustandsänderung das System in einem konsistenten und stabilen Zustand überführt.

Eine Transaktion ist nach [OTS03] eine logische Einheit (*Unit of Work*) welche die Eigenschaften Atomarität, Konsistenz, Isolation und Dauerhaftigkeit unterstützt. Diese werden im folgenden genauer ausgeführt.

2.1.1 Atomarität (Atomicity)

Die Betrachtung einer Transaktion als *Unit of Work* beschreibt treffend das Kriterium der Atomarität. Eine Transaktion wird als logische Einheit gesehen, die entweder ganz oder gar nicht ausgeführt werden darf ([TPCC03], [OTS03]). Wenn ein Ereignis die Transaktion negativ beeinflusst (unterbricht) muss das Transaktionssystem dafür sorgen, dass bereits erfolg-

reich ausgeführte Teiloperationen wieder auf ihren Zustand vor der Transaktion zurückgesetzt werden. Hierzu folgendes Beispiel:

Einer der Mitarbeiter der Firma *Terrific Housewares AG* ist Herr Muster, er hat sein privates Girokonto K2 bei seiner Hausbank *Terrific Banking AG* bei der auch die Firma *Terrific Housewares AG* das Firmenkonto K1 unterhält. Sein Gehalt beträgt 2.500 Euro, sein momentaner Saldo auf seinem Girokonto beträgt 1000 Euro. Vor der Überweisung des Gehalts für Mitarbeiter Herr Muster betrug das Firmenkonto K1 10.000.000,- Euro. Das erwünschte Resultat nach der Überweisung sollte sein:

Firmenkonto K1 : Saldo 9.997.500 Euro
Girokonto K2 : Saldo 3.500 Euro

Diese logische Einheit der Überweisung von K1 nach K2 muss technisch in mehrere Operationen unterteilt werden:

1. Saldo K1 muss um 2.500 Euro reduziert werden.
2. Saldo K2 muss um 2.500 Euro erhöht werden.

Diese Operationen müssen in jedem Fall komplett ausgeführt oder komplett nicht ausgeführt werden. Wenn durch ein Störereignis Operation (1) nur ausgeführt werden würde, dann hätte das Unternehmen Gehalt bezahlt, welches nie beim Mitarbeiter Herr Muster angekommen wäre. Wäre Operation (2) nur ausgeführt worden, dann hätte der Mitarbeiter Herr Muster Gehalt bekommen, welches nicht vom Unternehmen sondern von der Bank gezahlt worden wäre.

2.1.2 Konsistenz (Consistency)

Betrachtet man das Beispiel aus dem vorangegangenen Kapitel, so ist das Kriterium ob eine Transaktion korrekt abgewickelt wurde äußerst offensichtlich. Die Transaktion wurde dann korrekt ausgeführt wenn sie das folgende Ergebnis hat:

Firmenkonto K1 : Saldo 9.997.500 Euro
Girokonto K2 : Saldo 3.500 Euro

Damit wurde implizit auch schon eine wichtige Regel aufgestellt, welcher die Daten des Bankensystems genügen müssen:

Saldo K1 nach Transaktion = Saldo K1 vor Transaktion – Überweisungsbetrag
Saldo K2 nach Transaktion = Saldo K2 vor Transaktion + Überweisungsbetrag

Etwas generalisierter gesprochen und um die Berücksichtigung des Systemzustand bereinigt erhält man die folgende Konsistenzbedingung:

Der Saldo für ein Konto ist die Summe aller erfolgreich durchgeführten Überweisungen (Einzahlungen und Auszahlungen) auf das Konto. Dabei werden bei Einzahlungen der Betrag positiv addiert, während bei Auszahlungen der Betrag negativ addiert wird. Solange der Datenbestand dieser Konsistenzbedingung entspricht, ist das System konsistent.

Das Konsistenzkriterium einer Transaktion definiert nun folgerichtig, dass eine Transaktion ein System von einem konsistenten Zustand wieder in einen konsistenten Zustand überführen muss [TPCC03], davon ausgegangen dass das System vor der Transaktion in einem konsistenten Zustand war. Während der Durchführung einer Transaktion ist das System nicht notwendigerweise in einem konsistenten Zustand. Dies bleibt parallel ausgeführten Transaktionen jedoch im Normalfall verborgen [Gray93]. Dieses verbergen eines inkonsistenten Zustandes führt direkt zu einem weiteren Kriterium, welches den Grad der Verborgenheit gegenüber nebenläufigen Transaktionen definiert.

2.1.3 Isolation (Isolation)

Die Isolation von Transaktionen wird im folgenden durch den Bezug zu Phänomen definiert, welche bei der Ausführung von nebenläufigen Zugriffen auf gemeinsame Daten entstehen können.

2.1.3.1 Concurrency Confusion

Bei diesem Phänomen handelt es sich um die generelle Problematik bei der Ausführung von parallelen Transaktionen ohne Isolation. Die Transaktionen erhalten Zugriff auf temporär nicht integere Daten. Das in Kapitel 1.5 vorgestellte Datenmodell von TERPH beinhaltet die Entitäten Auftrag (*OrderData*) und Auftragszeile (*OrderLine*). Die Entität *OrderData* enthält ein Feld *OrderLineCount*, welches die Anzahl der Auftragszeilen beinhaltet. Es lässt sich somit folgende Konsistenzbedingung definieren:

[Summe von *OrderData::OrderLineCount*] = [Anzahl der Datensätze in *OrderLine*]

Es werden nun zwei Transaktion T1 und T2 auf dieses Datenmodell ausgeführt:

- Transaktion T1: Fügt neuen Auftrag (*OrderData*) und Auftragszeilen (*OrderLine*) in die Datenbasis ein. Dabei stellen das Einfügen des Auftrags und der einzelnen Zeilen einzelne Operationen dar.
- Transaktion T2: Automatische Routineüberprüfung der Konsistenzbedingungen der Datenbasis oder aber auch eine statistische Auswertung für das Management, welche sowohl auf das Feld *OrderLineCount* zugreift, als auch die Anzahl der einzelnen *OrderLine*-Datensätze betrachtet.

Bei paralleler Abarbeitung und keiner Isolation zwischen den Transaktionen fügt Transaktion T1 den Auftrag ein (Feld *OrderLineCount* hat den Wert 10), danach beginnt es die Auftragszeilen einzufügen. Parallel dazu wertet die Transaktion T2 das Feld *OrderLineCount* aus und bestimmt die Anzahl der eingefügten Auftragszeilen in *OrderLine*. Es ergibt sich eine Verletzung der obigen Konsistenzbedingung, welche nach Abschluss der Transaktion T1 jedoch nicht mehr existiert.

2.1.3.2 Lost Update (Dirty Write)

Beim Phänomen des *Lost Update* (oder auch *Dirty Write*) überschreiben sich parallel ausgeführte Transaktionen gegenseitig Daten. Es werden wieder zwei Transaktionen auf dem Datenmodell von TERPH betrachtet:

- Transaktion T1: Ein bestehender Auftrag in *OrderData* für einen Kunden K1 wird modifiziert. Es wird das Lieferdatum um eine Woche verschoben, da es Lieferschwierigkeiten gibt.
- Transaktion T2: Derselbe Auftrag wird modifiziert da der Kunde K1 mitteilte, dass die Lieferung einen Tag später ankommen soll, da sein eigenes Lager momentan keine weitere Ware aufnehmen kann.

Bei paralleler Abarbeitung der beiden Transaktionen wird beides Mal das Lieferdatum gelesen. Wenn zunächst die Transaktion T1 abgeschlossen wird und danach die Transaktion T2 hat das Lieferdatum den Wert von Transaktion T2. Die Änderung von Transaktion T1 ist verloren.

2.1.3.3 Phantom Problem

Bei einem Phantom-Problem, werden vorhandene Daten nicht erkannt, d.h. Transaktionen arbeiten auf einer nicht mehr gültigen Ergebnismenge. Wiederum werden zwei Transaktionen auf Basis von TERPH betrachtet:

- Transaktion T1: Eine Suche nach Kunde K1 und Lieferdatum in KW 10 soll die betreffenden Aufträge anzeigen, welche in der nächsten Woche beim Kunden eintreffen werden und alle diese Aufträge mit einem bestimmten Lieferstatus versehen.
- Transaktion T2: Ein neuer Auftrag wird für den Kunden K1 angelegt. Lieferdatum soll der Donnerstag der KW 10 sein.

Bei einer Ablaufreihenfolge Transaktion T1 vor Transaktion T2 wird das neue Element nicht gefunden, wohingegen bei Ausführung von Transaktion T2 vor Transaktion T1 es im Suchergebnis zurückgeliefert wird. Bei paralleler Abarbeitung kommt es auf den Zeitpunkt des Lesens der relevanten Daten für die Ergebnismenge von Transaktion T1 an. Wird Transaktion T2 vor dem Lesen der Ergebnismenge abgeschlossen, wird der neue Auftrag angezeigt, ansonsten ist es ein Phantom. Die Transaktion T1 würde bei einem erneuten Lesen eine andere Ergebnismenge zurückbekommen.

2.1.3.4 Non-Repeatable Read

Bei dem Phänomen des *Non-Repeatable Reads* führen Transaktionen, bei denen mehrere Lesevorgänge auf denselben Daten durchgeführt werden, zu unterschiedlichen Ergebnissen. Als Beispiel dienen wieder zwei Transaktionen auf dem Datenmodell von TERPH:

- Transaktion T1: Eine Suche nach Kunde K1 und Lieferdatum soll die betreffenden Aufträge anzeigen, welche nächste Woche beim Kunde eintreffen werden und die Aufträge mit einem bestimmten Lieferstatus versehen.
- Transaktion T2: Modifiziert das Lieferdatum eines Auftrags, da es auf Grund von Lieferschwierigkeiten eine Verzögerung gegeben hat und nicht diese Woche geliefert werden kann, sondern erst nächste Woche.

Bei einer Ablaufreihenfolge von Transaktion T1 vor Transaktion T2 wird das modifizierte Element nicht gefunden, wohingegen bei Ausführung von Transaktion T2 vor Transaktion T1 es im Suchergebnis zurückgeliefert wird. Bei paralleler Abarbeitung kommt es auf den Zeitpunkt des Lesens der relevanten Daten für die Ergebnismenge von T1 an. Wird Transaktion T2 vor dem Lesen der Ergebnismenge abgeschlossen, wird es angezeigt. Wird demnach die

Transaktion T2 wiederholt ausgeführt (Transaktion T1 -> Transaktion T2 -> Transaktion T1), werden unterschiedliche Ergebnismengen zurückgeliefert.

2.1.3.5 Dirty Read

Das Phänomen des *Dirty Reads* bezeichnet eine falsche Rückgängigmachung von Transaktionsergebnissen und ist auch unter der Problematik der *Rollback*-Fortpflanzung bekannt. Als Beispiel werden wieder zwei Transaktion auf dem Datenmodell von TERPH betrachtet:

- Transaktion T1: Modifiziert einen Auftrag (*OrderData* und *OrderLine*).
- Transaktion T2: Liest den Auftrag und verarbeitet diesen in einer Rechenoperation zur Planung der Warenbestände des Warenlagers (*Stock*).

Bei einer Ablaufreihenfolge von Transaktion T1 vor Transaktion T2 wird der modifizierte Auftrag gelesen und korrekt verarbeitet. Wird die Transaktion T2 vor Transaktion T1 ausgeführt, werden die alten Werte des Auftrags gelesen und verarbeitet. Bei paralleler Abarbeitung wird zunächst in Transaktion T1 der Auftrag verändert, vor dem Abschluss von T1 wird der Auftrag von Transaktion T2 gelesen. Danach wird die Transaktion T1 durch ein Störereignis wieder zurückgenommen (*Rollback*). Da jedoch kein Störereignis in Transaktion T2 aufgetreten ist, wird die Transaktion T2 ordnungsgemäß beendet. Die Transaktion T2 beinhaltet jedoch Daten der fehlgeschlagenen Transaktion T1, welche in dieser Weise nicht mehr in der Datenbasis existieren (und für das System nie existiert haben).

2.1.3.6 Isolationsstufen

Die ANSI SQL Spezifikation, sowie die entsprechenden Dokumentation der gängigsten Datenbanksysteme (DB2, Oracle, MS SQLServer, Sybase) unterscheiden vier Isolationsstufen (*Isolationlevel*), die sich in der Anzahl der gesetzten Sperren (*Locks*) und der damit erreichbaren Parallelverarbeitung von Transaktionen unterscheiden (Tabelle 1).

Tabelle 1 Isolationsstufen nach ANSI SQL Spezifikation

Phänomen / Isolation	***Concurrency Confusion***	***Dirty Write***	***Dirty Read***	***Non-Repeatable Read***	***Phantom***
Read Uncommited	Nicht erlaubt	Nicht erlaubt	Erlaubt	Erlaubt	Erlaubt
Read Commited	Nicht erlaubt	Nicht erlaubt	Nicht erlaubt	Erlaubt	Erlaubt
Repeatable Read	Nicht erlaubt	Nicht erlaubt	Nicht erlaubt	Nicht erlaubt	Erlaubt
Serializable	Nicht erlaubt	Nicht erlaubt	Nicht erlaubt	Nicht erlaubt	Nicht erlaubt

Read Uncommited: Sperren werden auf Elemente gehalten, welche modifiziert wurden und bis zum Ende einer Transaktion gehalten. Das Lesen eines Elements erzeugt keine Sperre. Diese Isolation erlaubt, dass auf Daten einer anderen Transaktion zugegriffen wird, welche gerade durchgeführt, aber noch nicht abgeschlossen wurde. Wird die erste Transaktion zurück gerollt, hat die zweite Transaktion Daten gelesen, welche nie gültig existierten. Trotz den Nachteilen im Bezug auf Isolation, oder vielmehr gerade deswegen, erlaubt diese Isolationsstufe den höchsten Grad an Parallelverarbeitung, da er die wenigsten Sperren setzt.

Read Commited: Sperren werden für das Lesen und Schreiben der Elemente gesetzt. Bei lesendem Zugriff auf Elemente, werden diese nach der Leseoperation wieder entfernt, die Sperren auf modifizierte Elemente werden bis zum Transaktionsende gehalten. Diese Isolationsstufe sichert zu, dass nur auf Daten zugegriffen wird, welche durch erfolgreiche Transaktionen in der Datenbasis festgeschrieben wurden.

Repeatable Read: Sperren werden für den lesenden und schreibenden Zugriffe auf Elemente gesetzt. Sperren auf modifizierte Elemente werden bis zum Ende der Transaktion gehalten, ebenso Sperren für gelesene Daten. Sperren auf nicht modifizierte Zugriffsstrukturen werden nach dem Lesen freigegeben. Diese Isolationsstufe garantiert das Daten, welche innerhalb einer Transaktion mehrfach gelesen werden, immer dieselben Werte zurückgeben. In einer parallelen Transaktion neu eingefügte Datensätze, welche demselben Suchkriterium genügen, werden jedoch nicht zurückgegeben (Phantom-Problem).

Serializable: Sperren werden für alle Daten, welche gelesen oder modifiziert werden gesetzt und bis zum Ende der Transaktion gehalten. Alle Zugriffsstrukturen, welche modifiziert wurden, werden bis zum Ende der Transaktion gesperrt, ebenso alle Zugriffsstrukturen von Abfragen. Diese Isolationsstufe garantiert das Transaktionen, welche in irgendeiner Form auf gemeinsamen Daten oder Abfragekriterien operieren so ausgeführt werden, als wären sie nacheinander ausgeführt worden.

Je höher die Isolationsstufe ist, desto weniger Parallelität und damit weniger Transaktionen innerhalb eines bestimmten Zeitraum können verarbeitet werden [Gray93].

2.1.4 Dauerhaftigkeit (Durability)

Die Eigenschaft der Dauerhaftigkeit einer Transaktion bezieht sich auf die Tatsache, dass das Ergebnis einer Transaktion über die Lebensdauer der Transaktion sowie der Anwendung selbst bestand haben muss, und zwar solange, bis eine andere erfolgreiche Transaktion diese ändert. Dies bedeutet, dass Zustandsänderungen welche innerhalb einer Transaktion gemacht werden, auf permanente Speichereinheiten persistiert werden müssen (beispielsweise auf interne Festplatten oder externe Speichereinheiten wie Dateiserver, Datenbankserver, Bandlaufwerke, optische Medien). Wenn die Anwendung oder das System nach der erfolgreichen Ausführung einer Transaktion fehlerbedingt beendet wird, muss das System beim Neustart der Anwendung sicherstellen, dass die festgeschriebenen Transaktionsergebnisse sichtbar sind.

Wenn persistierte Daten zerstört wurden, müssen Wiederherstellungsroutinen aus Zweit-Datenquellen (Backup von *Recovery Logs*) den Zustand der Datenbasis bis zu einem gewissen Zeitpunkt wiederherstellen können.

Natürlich bietet kein System vollständige Sicherheit. Hier muss immer eine Abwägung zwischen der Wichtigkeit der Daten (gemessen am monetären Aufwand für die komplette Wiederherstellung bei Totalverlust) und dem Budget für Systemsicherungsmaßnahmen gemacht werden.

2.2 Konkurrierender Zugriff auf Daten

Im Normalfall arbeitet nicht eine einzelne Person an den zentralen Datenbeständen, sondern diese werden von unterschiedlichsten Anwendungen und Benutzergruppen gleichzeitig bearbeitet. Es kommt somit zu einem konkurrierenden Zugriff auf gemeinsame Daten. Um die ACID-Kriterien von konkurrierenden Transaktionen sicherzustellen wird eine Komponente benötigt, welche die parallelen Benutzertransaktionen entgegen nimmt, auf ihre Serialisierbarkeitsbedingungen hin überprüft und in geeigneter Weise synchronisiert. Für diese Arbeit wird der Transaktionsmanager wie folgt definiert:

Ein Transaktionsmanager ist eine Komponente eines Transaktionsverarbeitungssystem, welche Transaktionsaufrufe entgegen nimmt und auf ihre Serialisierbarkeit überprüft. Durch geeignete Synchronisationsmethoden wird die geeignete Reihenfolge der Transaktions-

operationen festgelegt, so dass die Transaktionen mit den geforderten ACID-Kriterien ausgeführt werden können.

Einen derartigen Transaktionsmanager stellt in der Praxis z.B. ein gängiger Transaktionsmonitor (*TP heavy*) wie CICS oder Tuxedo dar, aber auch jedes Datenbanksystem (*TP light*), welches auf Mehrbenutzerbetrieb ausgelegt ist (z.B. DB2, Oracle, MS SQLServer, Sybase).

Die Fehlersituationen, welche im Parallelbetrieb entstehen können, wurden im Kapitel über Isolation schon erläutert. Auf der Ebene von einzelnen Datenobjekten, welche durch die Transaktion gelesen oder bearbeitet werden, kommen diese Fehlersituationen durch eine „ungeschickte“ Reihenfolge von Lese- und Schreib-Operationen auf die Datenobjekte zustande. Diese würden nicht auftreten, wenn die Transaktionen als Ganzes in irgendeiner Reihenfolge hintereinander, also seriell ablaufen würden. Um diese Serialisierbarkeitsbedingung näher zu betrachten, definieren wir zunächst den Begriff einer Sequenz von Transaktionen (Schedule) wie folgt:

Eine Sequenz von Transaktionen bezeichnet eine bestimmte zeitliche Anordnung bzw. einen bestimmten Ablauf aller Operationen von Transaktionen, die auf einem Computer ausgeführt werden [Gray93]. Hierzu ein Beispiel. Es werden zwei Transaktionen T1 und T2 betrachtet:

Transaktion T1 (Update A, B):	Transaktion T2 (Update B,C):
read A;	read B;
A := A + 10;	B := B + 10;
write A;	write B;
read B;	read C;
B:= B - 10;	C:= C - 10;
write B;	write C;

Tabelle 2 Sequenz für Transaktion T1 vor Transaktion T2 (Sequenz 1)

Transaktion T1 (Update A, B)	***Transaktion T2 (Update B, C)***	***Ergebnis***
	Startwerte: A=10, B=20, C=30	
read A; A := A + 10; write A; read B; B:= B - 10; write B;		
		A = 20, B = 10
	read B; B := B + 10; write B; read C; C:= C - 10; write C;	
		B = 20, C = 20

Tabelle 3 Sequenz für Transaktion T2 vor Transaktion T1 (Sequenz 2)

Transaktion T1 (Update A, B)	***Transaktion T2 (Update B, C)***	***Ergebnis***
	Startwerte: A=10, B=20, C=30	
	read B; B := B + 10; write B; read C; C:= C - 10; write C;	
		B = 30, C = 20
read A; A := A + 10; write A; read B; B:= B - 10; write B;		
		A = 20, B = 20

Tabelle 4 Parallelbetrieb von T1 und T2 – richtig synchronisiert (Sequenz 3)

Transaktion T1 (Update A, B)	***Transaktion T2 (Update B, C)***	***Ergebnis***
	Startwerte: A=10, B=20, C=30	
read A; (A = 10)		
	read B; (B=20)	
A := A + 10;		
	B := B + 10;	
write A;		A = 20
	write B;	B = 30
read B; (B=30)		
	read C; (C=30)	
B:= B - 10;		
	C:= C - 10;	
write B;		B = 20
	write C;	C = 20
		A = B = C = 20

Tabelle 5 Parallelbetrieb von T1 und T2 – falsch synchronisiert (Sequenz 4)

Transaktion T1 (Update A, B)	***Transaktion T2 (Update B, C)***	***Ergebnis***
	Startwerte: A=10, B=20, C=30	
read A; (A = 10)		
	read B; (B=20)	
A := A + 10;		
write A;		A = 20
	B := B + 10;	
read B; (B=20)		
	write B;	B = 30
B:= B - 10;		
	read C; (C=30)	
write B;		B = 10
	C:= C - 10;	
	write C;	C = 20
		A = C = 20, B=10

Zusammenfassung der Ergebnisse:

- [Tabelle 2] Sequenz für Transaktion T1 vor Transaktion T2 (Sequenz 1):
  A = 20, B = 20, C = 20
- [Tabelle 3] Sequenz für Transaktion T2 vor Transaktion T1 (Sequenz 2):
  A = 20, B = 30, C = 20
- [Tabelle 4] Parallelbetrieb von T1 und T2 – richtig synchronisiert (Sequenz 3):
  A = 20, B = 20, C = 20
- [Tabelle 5] Parallelbetrieb von T1 und T2 – falsch synchronisiert (Sequenz 4):
  A = 20, B = 10, C = 20

Sequenz 1 und 2 sind gültige Ergebnisse einer Transaktion, da die beiden Transaktionen zwar auf gemeinsamen Daten arbeiten, die Transaktionen jedoch nacheinander ausgeführt werden. Sobald jedoch Parallelbetrieb zugelassen wird, können die einzelnen Transaktionsoperationen zu unterschiedlichen Sequenzen angeordnet werden (Serialisierung). Jedoch führt nicht jede beliebige Anordnung von Transaktionsoperationen zu einem gültigen Ergebnis. Sequenz 3 liefert beispielsweise ein Ergebnis, welches schon von Schedule 1 geliefert wird, Sequenz 4 wartet hier jedoch mit einem völlig neuen Ergebnis auf und ist eine ungültige Serialisierung. Es wird deshalb definiert:

Eine Sequenz von Transaktionen t1, t2, ..., tn ist korrekt synchronisiert, wenn sie den gleichen Datenbestand und die gleichen Ausgabedaten erzeugt, wie irgendeine serielle Ausführung der Transaktionen [Gray93].

Bei genauerer Betrachtung ist diese Definition jedoch noch nicht befriedigend, da sie bisher nur liefert, wie korrekt synchronisierte Sequenzen auf Basis ihres Ergebnisses erkannt werden können. Viel wertvoller wäre eine Definition, wie Sequenzen so angeordnet werden können, dass sie der Serialisierbarkeitsbedingung entsprechen. Dazu müssen die Zugriffe auf die Datenobjekte genauer untersucht werden. Für die obigen Sequenzen werden die in Tabelle 6 aufgeführten Zugriffe auf die Datenobjekte A,B und C durchgeführt.

Tabelle 6 Zugriffe auf Datenobjekte A,B und C

Sequenz	***Datenobjekt A***	***Datenobjekt B***	***Datenobjekt C***
Sequenz 1	T1 : read T1 : write	T1: read T1: write T2: read T2: write	T2: read T2: write
Sequenz 2	T1 : read T1 : write	T2: read T2: write T1: read T1: write	T2: read T2: write
Sequenz 3	T1 : read T1 : write	T2: read (a1) T2: write (b1) T1: read (c1) T1: write (d1)	T2: read T2: write
Sequenz 4	T1 : read T1 : write	T2: read (a2) T1: read (b2) T2: write (c2) T1: write (d2)	T2: read T2: write

Es fällt auf das bei den Datenobjekten A und C keine Probleme auftreten, da diese jeweils nur von einer Transaktion benutzt werden. Anders jedoch das Datenobjekt B. Hier entscheidet offensichtlich die Reihenfolge der Operationen über eine gültige Serialisierung. Einleuchtend erscheint, dass sobald eine Transaktion schreibend auf ein Datenobjekt zugreift, der letzte Zugriff auf das Datenobjekt eine Leseoperation aus derselben Transaktion sein darf. Es kann auch ein Lesezugriff einer anderen Transaktion sein, solange sie keine Schreiboperation danach auf das Datenobjekt ausführt. Wir definieren einen Konflikt bei der Synchronisierung zweier Transaktionen t1 und t2 somit wie folgt:

Ein Konflikt aufeinander folgender Operationen o1 aus t1 und o2 aus t2 besteht dann, wenn o1 und o2 auf dasselbe Datenobjekt zugreifen und mindestens eine dieser Operationen eine schreibende Operation ist [Gray93].

Für die Sequenz 3 existieren somit folgende Konflikte:
K1 : T2 (a1) -> T1 (d1)
K2 : T2 (b1) -> T1 (c1)
K3 : T2 (b1) -> T1 (d1)

Konsolidiert erhält man für die Sequenz 3 die Konflikte K : T2 -> T1

Für Sequenz 4 existieren folgende Konflikte:
K1 : T2 (a2) -> T1 (d2)
K2 : T1 (b2) -> T2 (c2)
K3 : T2 (c2) -> T1 (d2)

Konsolidiert erhält man für die Sequenz 4 die Konflikte K : T2 <-> t1, und somit eine zirkuläre Abhängigkeit. Es zeigt sich, dass nicht jeder Konflikt automatisch ein Serialisierbarkeitsproblem bedeutet. Es kommt vielmehr auf die Reihenfolge der Einzeloperationen auf die gemeinsamen Datenobjekte an. Mit diesem Wissen kann die vorherige Definition konkretisiert werden:

Eine Sequenz ist korrekt synchronisiert, wenn in Konflikten die Transaktionen t1, t2, ... tn der Sequenz nicht in unterschiedlicher Reihenfolge auftreten, wenn also der Serialisierungsgraph der Schedule keine Zyklen enthält [Gray93].

Anders als die vorangegangene Definition kann mittels einer solchen Serialisierbarkeitsbedingung ein Transaktionsmanager entscheiden, ob eine momentan ablaufende Sequenz von Transaktionen korrekt synchronisiert ist oder aber er benutzt dieses Wissen um eine Sequenz korrekt synchronisiert anzuordnen. Um herauszufinden, wie ein Transaktionsmanager einen Parallelbetrieb ermöglicht, muss dieser die möglichen Fehlersituationen entweder komplett verhindern oder aber Fehlersituationen korrigieren. Die dafür notwendigen Synchronisierungsverfahren werden im nächsten Kapitel vorgestellt.

2.3 Synchronisiationsverfahren (Locking-Strategien)

2.3.1 Überblick

Die Verfahren, um zu einer richtig synchronisierten Sequenz zu kommen, können in drei Klassen unterteilt werden:

- **Optimistische Verfahren:**
  Hierbei analysiert der Transaktionsmanager kontinuierlich die Sequenzen der ablaufenden Transaktionen. Wird bei der Analyse festgestellt, dass die Serialisierbarkeitsbedingung verletzt wird, müssen die beteiligten Transaktionen zurückgesetzt werden. Der Name des Verfahrens kommt daher, dass das Verfahren „optimistischerweise“ davon ausgeht, dass

die Transaktionen keine Synchronisationsfehler verursachen. Erst wenn die Analyse der Sequenz einen solchen Synchronisationsfehler zeigt wird die Sequenz dadurch „repariert“, dass die fehlerverursachenden Transaktionen zurückgesetzt werden. Ein Vertreter aus der Praxis für ein solches optimistisches Synchronisationsverfahren ist das Zeitstempelverfahren, es kommt in vielen verteilten Datenbanksystemen zum Einsatz [Gray93].

- **Verzögernde Verfahren:**
  Hierbei werden die Änderungen an Datenobjekten zunächst auf Kopien durchgeführt. Der Transaktionsmanager prüft zu bestimmten Zeitpunkten (*Sync Points*) ob die Serialisierbarkeit gewährleistet ist. Falls dies zutrifft, wird das Original durch die Kopie überschrieben.
- **Pessimistische (präventive) Verfahren:**
  Hierbei werden die benötigten Datenobjekte nach dem „Windhundprinzip“ für andere Transaktionen gesperrt, so dass die ablaufende Transaktion für eine bestimmte Zeit einen exklusiven Zugriff auf das gesperrte Datenobjekt hat. Damit wird verhindert, dass nichtserialisierbare Sequenzen überhaupt auftreten können. Vertreter dieses Verfahrens sind die in der Praxis gebräuchlichsten und auch historisch ältesten Synchronisationsverfahren. Bei hohen Transaktionsraten mit überwiegend schreibendem Zugriff werden in der Praxis fast ausschließlich pessimistische Verfahren eingesetzt.

2.3.2 Pessimistische Synchronisationsverfahren (Sperrverfahren)

2.3.2.1 Funktionsprinzip

Bei pessimistischen Synchronisationsverfahren verhindert der Transaktionsmanager durch Sperrung der benötigten Datenobjekte, dass die Synchronisationsbedingung verletzt wird. Dies geschieht dadurch, dass eine Transaktion t1 für ein Datenobjekt D eine Sperre beim Transaktionsmanager anfordert. So ist das Datenobjekt D für andere Transaktionen gesperrt. Erst nachdem die Sperre für Transaktion t1 wieder aufgelöst wurde, können andere Transaktionen auf das Datenobjekt zugreifen. Im folgenden klassifizieren wir die Sperrverfahren nach bestimmten Kriterien.

2.3.2.2 Art der Steuerung von Sperren

Bei (Halb-) Automatische Sperren setzt der Transaktionsmanager entweder autonom oder nach bestimmten, vom Benutzer festgelegten Strategien die Sperren selbst. Eine andere Vari-

ante sind die benutzergesteuerten Sperren. Hier bestimmt der Benutzer durch entsprechende Anweisungen innerhalb der Transaktion, wann welches Datenobjekt gesperrt bzw. freigegeben wird.

2.3.2.3 Sperrobjekte

Eine Klassifizierung wird hier auf Basis der Sperrobjekte getroffen. Es existieren folgende Varianten:

Objekttypen: Sperrung von logischen Objekten (Tupel, Tabelle, Tablespace, Datenbank) oder von physischen Objekten (Volume, Datei).

Granularität der Sperren: Sperrung eines Feldwertes, eines Feldes, eines Datensatzes, einer Tabelle oder einer ganzen Datenbank. Hier ist zu beachten, dass je feiner die Granularität wird, desto mehr wird (theoretisch) die Parallelität bei Transaktionen möglich. Diese wird jedoch durch ein Ansteigen des Verwaltungsaufwandes und der Deadlock-Gefahr kompensiert. In der Praxis kommen nahezu alle Möglichkeiten vor. Man spricht hier von Sperrhierarchien (Datenbank->Tabelle->...->Feldwert). Die Hierarchiestufen können dabei der jeweiligen Transaktion angepasst werden. Hierbei muss jedoch beachtet werden, dass der Transaktionsmanager die Entscheidung, ob eine Sperre vergeben werden kann, nicht nur an der jeweiligen Hierarchiestufe der angeforderten Sperre festmachen darf. Beispielsweise könnte eine Transaktion t1 eine Sperre für einen einzelnen Datensatz in der Tabelle *OrderLine* anfordern, eine Transaktion t2 eine Sperre für die ganze Tabelle *OrderLine*. Hier darf die Sperre auf *OrderLine* für t2 nicht vergeben werden, obwohl nur ein Datensatz aus *OrderLine* gesperrt ist). Hierzu benutzt der Transaktionsmanager ein *Intention Lock* auf hierarchisch übergeordnete Objekte (in unserem Beispiel wäre dies die Tabelle *OrderLine*). Für den Transaktionsmanager bedeutet es, dass niemand anders mehr eine Sperre für die Tabelle anfordern kann. Sperren auf einzelne Datensätze sind jedoch weiterhin möglich.

Sperren von vordefinierten Objekten oder über Prädikate: Bei vordefinierten Objekten kann der Transaktionsmanager über Schlüsselwerte oder Speicheradressen das zu sperrende Objekt auf Grund seiner Zugehörigkeit zu einem logischen oder physischen Bereich sofort lokalisieren und prüfen ob die Sperrung möglich ist.

Beim Sperren über Prädikate (z.B. Wohnort=“Tübingen“) weiß der Transaktionsmanager nicht im voraus, welche Datenobjekte diese Bedingung erfüllen. Diese Sperrart verursacht deshalb erheblichen Mehraufwand (zuerst lesen, dann prüfen, dann sperren). Dieser Mehraufwand zusammen mit dem Überlappungsproblem, welches durch eine nicht disjunkte Ergebnismenge der Sperranfragen mit Prädikaten resultiert hat zur Folge, dass dieses Verfahren in der Praxis keine Bedeutung hat.

2.3.2.4 Sperrmodus

Dieses Kriterium unterscheidet Sperrverfahren danach, wie gesperrt wird bzw. was die Transaktionen t2, t3 usw. noch dürfen, wenn sie auf Datenobjekte zugreifen, die von t1 gesperrt wurden. Es steht also die Frage dahinter, wie intensiv gesperrt wird. Folgende Fälle sind zu unterscheiden:

Exclusive Lock (X): Andere Transaktionen dürfen so lange keine weitere Sperren auf dieses Datenobjekt anfordern, bis t1 das Datenobjekt wieder freigegeben hat. Diese Sperrart ist sinnvoll wenn t1 das Datenobjekt schreiben will; man nennt die X-Sperre daher auch Schreibsperre.

Shared Lock (S): Andere Transaktionen dürfen weitere S-Sperren anfordern, jedoch keine X-Sperren. Dies ist sinnvoll wenn t1 selbst nur liest; dann dürfen andere Transaktionen auch (nur) lesen, da keine Konflikte auftreten können. Man bezeichnet diese Art von Sperre daher auch als Lesesperre.

Update Lock (U): Andere Transaktionen dürfen S-Sperren anfordern; beim Schreibvorgang wird U in X konvertiert. Dadurch, dass andere Transaktionen S-Sperren setzen dürfen, kann es vorkommen das t1 mit dem Schreibvorgang solange warten muss, bis die anderen Transaktionen ihre S-Sperren wieder aufgehoben haben.

Ein **Intention Lock** existiert wiederum in mehreren Varianten:

Shared Intention Lock (IS): Andere Transaktionen dürfen auf dem übergeordneten Objekt S-Sperren setzen.

Exclusive Intention Lock (IX): Andere Transaktionen dürfen auf dem übergeordneten Objekt keine Sperren setzen.

Exclusive and Shared Intention Lock (SIX): Andere Transaktionen dürfen auf dem aktuellen Objekt S-Sperren setzen, auf dem übergeordneten Objekt jedoch nicht.

2.3.2.5 Sperrprotokoll

Dieses Kriterium unterscheidet Sperrverfahren danach, wie der Ablauf des Sperrens und Freigebens ist (Sperrprotokoll). Alle marktgängigen Datenbanksysteme haben mittlerweile das Zwei-Phasen-Sperrprotokoll implementiert; es sorgt dafür, dass keine *Concurrency Confusion* (Kapitel 2.1.3.1) entstehen kann.

Zwei-Phasen-Sperrprotokoll: Auch wenn mit Sperren gearbeitet wird, kann es bei "ungeschicktem" Setzen der Sperren dennoch vorkommen, dass eine der im Kapitel über Isolation beschriebenen Fehlersituationen eintritt. Eine solche Situation ist in Tabelle 7 dargestellt.

Tabelle 7 Fehlersituation ohne Anwendung des Zwei-Phasen-Sperrprotokolls

Transaktion T1	***Transaktion T2***	***Ergebnisse***
lock A;		
A := 10;		A = 10
print A;		
unlock A;		
	lock A;	
	A := 0;	A = 0
	unlock A;	
lock A,B, C;		
read A, B;		
C := B/A		Fehler – Division durch 0.
unlock A;		

Die Fehlersituation ergibt sich, weil Datenobjekt A vor der erneuten Bearbeitung freigegeben und wieder gesperrt wurde. Dies hat denselben Effekt als ob A überhaupt nicht gesperrt worden wäre. Es muss also gefordert werden, dass innerhalb einer Transaktion Datenobjekte nicht mehrmals gesperrt und freigegeben werden dürfen. Das Einhalten einer solchen Forderung ist für den Transaktionsmanager jedoch sehr schwierig, da er am Beginn der Transaktion schon über alle Datenobjekt-Operationen innerhalb der Transaktion informiert sein müsste. Nur so könnte er entscheiden, welches Datenobjekt freigegeben werden kann und welches noch benötigt wird. Stattdessen wird ein etwas gröberer, pragmatischerer Ansatz angewendet. Eine Transaktion darf keine Sperre mehr auf ein Datenobjekt anfordern, nachdem es eine Sperre (auf irgendein Datenobjekt) freigegeben hat. Es gibt also zunächst innerhalb einer Transaktion eine Sperrphase, in welcher die benötigten Datenobjekte sukzessive gesperrt werden. Darauf werden Transaktionsoperationen ausgeführt und danach werden die Datenobjekte in der Freigabephase wieder entsperrt. Dieses Vorgehen wird Zwei-Phasen-Sperrprotokoll genannt.

Zwei-Phasen-Sperrprotokoll mit Sperren bis „End of Transaction“ (EOT): Ein Problem, welches trotz des Zwei-Phasen-Sperrprotokolls besteht, ist das Problem der Rollbackfortpflanzung. Dies ist in Tabelle 8 dargestellt.

Tabelle 8 Fehlersituation ohne Zwei-Phasen-Sperrprotokoll mit Sperren bis EOT

Transaktion T1	***Transaktion T2***	***Zeitpunkt***
lock A;		z0
lock C;		z1
lock B;		z2
unlock B;		z3
unlock A;		z4
	lock A;	z5
rollback;		z6

Die Transaktion T1 sperrt das Datenobjekt A vom Zeitpunkt z0 bis zum Zeitpunkt z4. Zum Zeitpunkt z5 sperrt T2 dieses Datenobjekt. T1 sperrt auch B (von z2 bis z3) und C (ab z1). Zum Zeitpunkt z6 wird T1 abgebrochen und muss zurückgesetzt werden; da T2 dort schon Datenobjekt A verwendet, müsste T2 auch rückgesetzt werden, da es mit einem nicht mehr zutreffenden A arbeitet. Das Zwei-Phasen-Sperrprotokoll wird eingehalten, die Sperren werden durch T1 von z0 bis z2 gesetzt und ab z3 freigegeben.

Die Realisierung einer Rollbackfortpflanzung ist für einen Transaktionsmanager mit erheblichem Verwaltungshauptstadt verbunden, da er die Abhängigkeiten zwischen den Transaktionen, welche sich durch die parallele Nutzung von Datenobjekten in verschiedenen Transaktionen ergeben, protokollieren muss. Nur dadurch lassen sich bei einem *Rollback* einer Transaktion die davon betroffenen Datenobjekte ebenfalls zurückrollen. Dies ist sehr aufwändig, weshalb normalerweise ein etwas einfacherer Ansatz eingesetzt wird. Hierbei werden die Datenobjekte bis zum Transaktionsende (EOT) gesperrt und erst danach wieder freigegeben. Dadurch kann die obige Fehlersituation überhaupt nicht auftreten. Zu bemerken ist jedoch, dass dadurch auch Parallelisierungspotential eingebüßt wird, da bei diesem Ansatz die Datenobjekte von einer anderen Transaktion erst wieder benutzt werden können, nachdem die aktuelle Transaktion beendet wurde.

Zwei-Phasen-Sperrprotokoll mit Sperre bis EOT und Preclaiming: Ein weiteres Problem, welches mit dem Zwei-Phasen-Sperrprotokoll nicht beseitigt wird ist die Gefahr eines *Deadlock*. Ein *Deadlock* entsteht, wenn zwei Transaktionen in unterschiedlicher Reihenfolge auf Datenobjekte zugreifen. Dies ist in Tabelle 9 dargestellt.

Tabelle 9 Fehlersituation ohne Zwei-Phasen-Sperrprotokoll mit Sperre bis EOT und Preclaiming

Transaktion T1	***Transaktion T2***	***Ergebnisse***
lock A;		
	lock B;	
lock B;		Transaktion T1 kann erst eine Sperre auf B erhalten, wenn Transaktion T2 beendet ist. Transaktion T1 ist blockiert!
	lock A;	Transaktion T2 kann erst eine Sperre auf A erhalten, wenn Transaktion T1 beendet ist. Transaktion T2 ist blockiert!
		Fehlersituation : *Deadlock*.

Durch *Preclaiming* kann dieses Problem beseitigt werden. *Preclaiming* bedeutet, dass alle benötigten Datenobjekte innerhalb einer Transaktion am Anfang gesperrt werden. Zu dem Problem, welches schon oben angesprochen wurde (Kenntnis aller Datenobjekte am Beginn einer Transaktion), kommt hier noch die Parallelisierungsproblematik hinzu. Die Datenobjekte für eine Transaktion würden von Beginn bis Ende gesperrt, was eine Parallelverarbeitung für diese Datenobjekte unmöglich machen und einen Parallelbetrieb damit empfindlich beeinträchtigen würde. Deshalb werden in der Praxis *Deadlocks* nicht präventiv verhindert, sondern mit einem Analyseverfahren entdeckt. Die betroffenen Transaktionen werden daraufhin zurückgesetzt. Einige Transaktionsmanager (z.B. Oracle) bieten noch die Optimierung, dass die beteiligten Transaktionen, geordnet nach den wenigsten Datenmanipulationen, sukzessive zurückgesetzt werden bis die *Deadlock*-Situation aufgelöst wurde.

2.3.3 Optimistische Synchronisationsverfahren

Bei optimistischen Synchronisationsverfahren analysiert der Transaktionsmanager die Sequenz der ablaufenden Transaktionen kontinuierlich. Verletzt eine oder mehrere Transaktionen die Serialisierbarkeitsbedingung werden diese zurückgesetzt. Die Serialisierbarkeit ist dann nicht gegeben, wenn in Konflikten die Reihenfolge der darin beteiligten Transaktionen unterschiedlich ist. Diese Bedingung ist in der Praxis, insbesondere bei kritischen

Performanzbedingungen, über die gesamte Sequenz hinweg praktisch nicht zu prüfen. Daher benutzen die in der Praxis eingesetzten optimistischen Synchronisationsverfahren ein etwas gröberes Messkriterium. Am Beispiel des Zeitstempelverfahrens wird im folgenden die Funktionsweise eines optimistischen Synchronisationsverfahrens dargestellt.

Das Zeitstempelverfahren benutzt als Entscheidungskriterium den Zeitpunkt der BOT (*begin of transaction*). Es wird nicht erlaubt, dass "ältere" Transaktionen Operationen auf Datenobjekten ausführen, die von "jüngeren" Transaktionen bereits gelesen oder geschrieben wurden. Damit dieses Verfahren funktioniert, muss jedes Datenobjekt wissen, wann es zum letzten Mal Teil einer Transaktion war. Etwas genauer, wird jeder Transaktion t eine Zeitmarke TS(t) zugeordnet; diese Zeitmarke gibt den BOT-Zeitpunkt wieder, also den Zeitpunkt zu dem die erste Lese- bzw. Schreiboperation dieser Transaktion begonnen wird. Jede Einzeloperation o(t) erhält TS(t) als Zeitmarke. Jedes Datenobjekt D besitzt zwei Zeitstempel:

- TSW(D): Die Zeitmarke TS(t) der letzten Operation o(t) die schreibend auf D zugegriffen hat.
- TSR(D): Die Zeitmarke TS(t) der letzten Operation o(t) die lesend auf D zugegriffen hat.

Ein Transaktionsmanager, welcher nach dem Zeitstempelverfahren synchronisiert, geht nach zwei Regeln vor. Dabei wird angenommen, dass o(t) eine Einzeloperation aus einer Transaktion t sei, die auf ein Datenobjekt D zugreifen will; TS(t1) < TS(t2) soll bedeuten, dass t1 vor t2 vom Transaktionsmanager angenommen wurde, also t1 älter als t2 ist:

1. Für eine Operation o(t), welche eine Leseoperation darstellt, wird geprüft ob TS(t) < TSW (D) ist, also ob es noch eine jüngere Schreiboperation gibt, welche auf D zugegriffen hat.
   a) Falls ja, dann wird o(t) abgebrochen
   b) Falls nein, dann wird o(t) ausgeführt und TSR(D) := max {TSR(D), TS(t)} gesetzt, also auf das jüngste Lesedatum einer Operation o(t) bezogen auf D.
2. Für eine Operation o(t), welche eine Schreiboperation darstellt, wird geprüft ob TS(t) < max {TSR(D), TSW(D)} ist, also ob es es noch eine jüngere Lese- oder Schreiboperation gibt.
   a) Falls ja, dann wird o(t) abgebrochen.
   b) Falls nein, dann wird o(t) ausgeführt und TSW(D) := TS(t) gesetzt, also auf das jüngste Schreibdatum einer Operation o(t) bezogen auf D.

Ein Beispiel soll dieses Vorgehen veranschaulichen. Die in Tabelle 10 aufgeführten einfachen Sequenzen sind nicht korrekt synchronisiert.

Tabelle 10 Transaktionen und Zeitmarken für Beispiel zum Zeitstempelverfahren

Transaktion T1	***Transaktion T2***	
read D;		o1(t1) = 2000
	read D;	o1(t2) = 3000
write D;		o2(t1) = 4000
	write D;	o2(t2) = 5000

Die Zeitstempel der einzelnen Operationen o(t) sind jeweils gemessen in Millisekunden, relativ zu einer absoluten Zeitmarke Z. Unter der Annahme, dass das Datenobjekt D die Zeitstempel TSR(D) = 1000 und TSW(D) = 0 hat, ergibt die Untersuchung des Transaktionsmanager die in Tabelle 11 aufgeführten Schritte und Folgerungen.

Tabelle 11 Durchführung des Zeitstempelverfahrens

Operation	***Prüfung***	***Folge***
Ermittlung von TS(T1) TS(T1) = o1(T1)		TS(T1) = 2000
Ermittlung von TS(T2) TS(T2) = o1(T2)		TS(T2) = 3000
o1(T1) : 2000	[TS(T1) < TSW(D)] [TS(o1(T1)) < TSW(D)] [2000 < 0] -> Bedingung nicht erfüllt!	Es liegt kein Synchronisationsfehler vor, deshalb: TSR(D) = max {TSR(D), TS(T1)} TSR(D) = max {1000, 2000} TSR(D) = 2000
o1(T2) : 3000	[TS(T2) < TSW(D)] [3000 < 0] -> Bedingung nicht erfüllt!	Es liegt kein Synchronisationsfehler vor, deshalb: TSR(D) = max {TSR(D), TS(T2)} TSR(D) = max {2000, 3000} TSR(D) = 3000

Operation	*Prüfung*	*Folge*
o2(T1) : 4000	[TS(T1) < max {TSR(D), TSW(D)}] [2000 < max {3000, 0}] [2000 < 3000] -> Bedingung erfüllt!	Es liegt ein Synchronisationsfehler vor. Die Transaktion T1 muss abgebrochen werden.
o2(T2) : 5000	[TS(T2) < max {TSR(D), TSW(D)}] [3000 < max {3000, 0}] [3000 < 3000] -> Bedingung nicht erfüllt!	Es liegt kein Synchronisationsfehler vor, deshalb: TSW(D) := TS(T2) TSW(D) := 3000

Auffallend ist, dass „ältere“ Transaktionen bei diesem Verfahren benachteiligt werden.

2.4 Two Phase Commit

Bisher wurden ausschließlich lokale Transaktionen diskutiert. Ein Transaktionsmanager arbeitet die Transaktionen ab und sorgt für die notwendigen ACID-Kriterien. Die Ressource, welche er dabei bedient, ist eine Datenbank (gegebenenfalls obliegt der Datenbank selbst die Transaktionssteuerung). Problematisch wird es jedoch mit dem einhalten der ACID-Kriterien, wenn eine Transaktion sich über mehrere Ressourcen (Datenbanken) erstreckt. Hierbei übernimmt der Transaktionsmanager die Rolle eines Koordinators, welcher die Transaktion zwischen den Ressourcen koordiniert. Da die Ressourcen unabhängig voneinander sind, darf das Festschreiben der Daten (*Commit*) nicht in einem zentralen Aufruf passieren, da ein Zurückrollen der Transaktion (*Rollback*) von den anderen Ressourcen nicht mehr berücksichtigt werden könnte. Somit käme es zu einer teilweisen Festschreibung der Transaktion und damit zur Verletzung der ACID-Kriterien. Um in einem solchen Szenario ein *Commit* durchführen zu können, müssen Transaktionsmanager sowie die Ressourcen selbst ein mehrphasiges Festschreibungsprotokoll unterstützen. Das Zwei-Phasen-Festschreibungsprotokoll oder auch *Two Phase Commit (2PC)* ist eine Variante eines solchen Protokolls [Bern97]. Es gibt mehrere proprietäre Implementierungen dieses Protokolls, der Industriestandard hierfür ist unter X/Open XA bekannt [Ope92]. Das Commit ist bei einem 2PC in zwei Phasen unterteilt:

Phase 1 (*prepare phase*): Der Koordinator sendet an jede beteiligte Ressource das Kommando, dass die Transaktionsoperationen daraufhin überprüft werden sollen, ob sie festgeschrieben werden können. Die Ressource meldet an den Koordinator zurück, ob er einen *Commit* durchführen kann.

Phase 2 (*commit phase*): Wenn einer oder mehrere Ressourcen gemeldet haben, dass sie ein *Commit* nicht durchführen können, wird die gesamte Transaktion zurückgesetzt. Die ein-

zelnen beteiligten Ressourcen werden darüber informiert und führen daraufhin ein *Rollback* ihrer Transaktionsoperationen durch. Wenn alle Ressourcen gemeldet haben, dass sie zum *Commit* bereit sind, sendet der Koordinator an jede Ressource das Kommando, dass die Transaktion festgeschrieben werden soll. Jede Ressource führt sein *Commit* aus und meldet dies an den Koordinator zurück.

In verteilten Systemen kann es jedoch auch zu Kommunikationsproblemen zwischen den Systemen kommen, d.h. dass ein Koordinator die Verbindung zu einem oder mehreren Ressourcen verliert. Wenn diese Verbindung nicht innerhalb eines definierten Zeitraums wiederhergestellt werden kann, muss sowohl der Transaktionsmanager als auch die Ressource einseitig eine Entscheidung über *Commit* oder *Rollback* treffen. Eine solche Entscheidung wird „heuristische Entscheidung“ (*heuristic decision*) genannt. Mit einem solchen Fehler ist sehr schwierig umzugehen und kann zu Verletzung der ACID-Kriterien führen [Gray93].

2.5 Typen von Transaktionen

Dieses Kapitel stellt die verschieden Typen von Transaktionen vor und erläutert ihre Merkmale.

2.5.1 Lokale Transaktion

Der historisch älteste Transaktionstyp ist auf die Persistierung von Daten unter Verwendung einer einzelnen zentralen Datenbank ausgerichtet. Hierbei werden eine große Zahl von sehr kurzlebigen Transaktionen auf einer Ressource (Datenbank) ausgeführt. Charakteristisch für die lokalen Transaktionen ist die hohe Performanz, da alle Aufrufe lokal sind und nur eine Ressource die Transaktionen durchführt. Die Transaktionen folgen dem Paradigma „Run-to-Completion“, d.h. die Transaktion laufen ohne Benutzer-Interaktion bzw. Unterbrechung bis sie beendet sind. Daraus resultiert die Aussage der relativen Kurzlebigkeit der Transaktionen. Eine Transaktion für das Speichern von Kundendaten wird dann durchgeführt, wenn die Daten vom Terminal an den Transaktionsmanager übergeben wurden. Die Transaktion umfasst nicht den gesamten Eingabeprozess der Daten, sowohl das Lesen der Daten für die Kundendaten-Eingabemaske als auch das abschließende Speichern der Daten stellt eine separate lokale Transaktion dar. Ein weiteres Merkmal ist die starke ACID-Anforderung dieses Transaktionstyps.

2.5.2 Globale oder Verteilte Transaktion

Eine Erweiterung des lokalen Transaktionsszenarios stellt die Benutzung von mehreren Datenbanken innerhalb einer Transaktion dar. Dazu folgendes Beispiel der *Terrific Housewares AG*. Eine Datenbank A ist für die auftragsrelevanten Daten zuständig, eine Datenbank L ist für die Lagerhaltung zuständig. Ein Auftrag, welcher gespeichert wird, überprüft ob in L noch genügend Stück des bestellten Produktes vorhanden sind:

- wenn ja, dann reduziert er den verfügbaren Lagerbestand um die Anzahl der bestellten Positionen. Danach wird der Auftrag mit dem Status „Zur Auslieferung bereit“ in A gespeichert.
- wenn nein, dann wird der Auftrag mit dem Status „Lagerbestand nicht ausreichend“ gespeichert.

Die Transaktion des Speichern eines Auftrags erstreckt sich also über die Datenbanken A und L, auf die jeweils lesend und schreibend zugegriffen werden.

Bis auf die Charakteristika, dass bei verteilten Transaktionen mehrere Datenbanken beteiligt sind, unterscheiden sich die verteilten Transaktionen im Laufzeitverhalten nicht von den lokalen Transaktionen. Auch sie sind kurzlebig und reichen nicht über eine Benutzerinteraktion hinaus und auch sie folgen dem „Run-to-Completion“ Paradigma. Hingegen gibt es bei der Performanz und dem Verwaltungsaufwand einen großen Unterschied. Bei verteilten Transaktionen muss sehr viel mehr Aufwand investiert werden um die ACID-Kriterien sicherzustellen (Kapitel 2.4). Auch der Kontextwechsel zwischen zwei Prozessen oder gar zwei völlig unterschiedlichen Maschinen (Netzwerkkommunikation) macht eine verteilte Transaktion um ein vielfaches teurer als eine lokale Transaktion.

2.5.3 Business Transaction

Nach [McG03] kann eine *Business Transaction* als eine konsistente Zustandsänderung innerhalb einer Geschäftsbeziehung unter zwei oder mehreren Parteien bezeichnet werden. Dabei unterhält jede Partei ein eigenständiges Anwendungssystem, welches den Status der Anwendungen und der Daten verwaltet. Hieraus ergeben sich einige neue Anforderungen:

Lose Koppelung von Diensten: Betrachtet wird als Beispiel die *Terrific Housewares AG*, welche für den Zahlungsverkehr die Vorgehensweise benutzt, dass durchzuführende Zahlungen im TERPH erfasst werden, jedoch von der Hausbank letztendlich durchgeführt werden. Die Transaktion bezeichnet das TERPH als *Payment*-Transaktion. Während der *Payment*-Transaktion soll der Zahlungsverkehr-Datensatz auf die TERPH-Datenbank geschrieben

werden, sowie mit dem Zahlungsdienst der Hausbank kommuniziert werden, um die relevanten Zahlungsdaten zu übergeben. Die Hausbank selbst könnte für einzelne Dienste, wie z.B. die Ausstellung eines Schecks, im Rahmen einer Gutschrift wiederum auf externe Dienstleister zugreifen.

Langlebige Transaktionen (*longrunning transactions*): Viele *Business Transactions* sind langlebige Transaktionen, da sie eine Zustandsänderung in einer Geschäftsbeziehung beinhalten, wie beispielsweise eine Buchung einer Geschäftsreise eines Vertriebsmitarbeiters von *Terrific Housewares AG* bei einer Reiseagentur. *Terrific Housewares* reserviert zunächst Plätze in einem Flugzeug und in einem Hotel über die Reiseagentur, danach hat sie eine bestimmte Zeit um die Reservierung zu bestätigen. Tut sie dies während dieser Zeit nicht, muss eine Alternativregelung definiert sein, die entweder nochmals anfragt oder die Reservierung storniert. Dies alles kann mehrere Stunden oder Tage dauern.

Hierbei erkennt man, dass dieses Szenario schwerlich mit den bisher bekannten Mitteln der Transaktionsisolation behandelt werden kann. Sperren auf Ressourcen (Datenobjekte) zu halten würde den Grad der Parallelität auf ein unakzeptables Maß senken. Außerdem würde ein direktes aneinander ketten von Sperren auf Datenobjekten und die Reaktion einer anderen Geschäftspartei kritische Sicherheitsfragen aufwerfen. Beispielsweise könnte eine dritte Partei durch Verhinderung der Kommunikation zwischen beiden Parteien den Arbeitsablauf innerhalb der beiden Geschäftsparteien dauerhaft stören. Dies ist mit Sicherheit nicht wünschenswert. Das ACID-Kriterium nach Isolation muss demnach bei *Business Transactions* etwas gelockert werden.

Optionale Untertransaktionen (*optional subtransactions*): Bei *Business Transactions* ist es möglich, dass einzelne Teilaspekte der Gesamttransaktion fehlschlagen können, ohne dass die Gesamttransaktion selbst ungültig wird und zurückgerollt werden muss. Hierzu folgendes Beispiel:

Eine Reiseagentur hat von der *Terrific Housewares AG* den Auftrag für eine Geschäftsreise bekommen. Hierzu konsultiert die Reiseagentur mehrere Fluggesellschaften A, B und C um das beste Preis-/Leistungsverhältnis zu ermitteln. Fluggesellschaft A und B haben entsprechende Plätze frei und reservieren diese vorsorglich, dann übermitteln die Fluggesellschaften die Preise an die Reiseagentur. Fluggesellschaft C hat jedoch nicht mehr so viele Plätze frei, die Reservierung der Plätze schlägt fehl. Dieser Fehler macht jedoch die Gesamttransaktion der „Buchung einer Geschäftsreise“ für die *Terrific Housewares AG* bei der Reiseagentur nicht ungültig, solange A oder B ein entsprechendes Potential haben. Als Folge muss das Atomaritäts-Kriterium auf einer granulareren Ebene betrachtet und für die Gesamttransaktion nicht strikt angewendet werden.

Ebenso kann es notwendig sein, nachdem die Fluggesellschaft A den Zuschlag für die Buchung bekommen hat, dass an Fluggesellschaft B eine Kompensationstransaktion abgesetzt wird um die bei der Anfrage reservierten Plätze wieder freizugeben. Auf der Ebene der Ge-

samttransaktion ist somit auch das Dauerhaftigkeits-Kriterium ebenfalls nicht streng zu sehen.

Propagierung des Transaktionskontexts: Da die *Business Transactions*, Anwendungen von unterschiedlichen Parteien kombiniert, welche unterschiedliche Infrastrukturen, Datenbanken und Anwendungsarchitekturen beinhalten, muss jeweils der Transaktionskontext von System zu System übertragen werden.

In Tabelle 12 sind die dargestellten Unterschiede nochmals zur Übersicht aufgelistet.

Tabelle 12 Zusammenfassung der Unterschiede von herkömmlichen Transaktionen und Business Transactions

Eigenschaft	***Herkömmliche Transaktionen***	***Business Transactions***
Atomarität	Notwendig	Unterschiedlich: Hier kommt es auf den Kontext der *Business Transaction* an, manchmal uneingeschränkt gewünscht, manchmal auch nur auf einer Untermenge notwendig.
Konsistenz	Notwendig	Notwendig. Zeitweilige Inkonsistenz ist jedoch erlaubt.
Isolation	Notwendig	Reduziert; jeder Dienst kontrolliert den Grad der Sichtbarkeit selbst.
Dauerhaftigkeit	Notwendig	Notwendig, jedoch im Zusammenhang mit der reduzierten Atomaritätsanforderung können auch hier Teile überschrieben werden.
Propagierung des Kontext	Nicht notwendig	Notwendig

Konkrete Spezifikationen für Systeme, welche *Business Transactions* durchführen können sind:

- Business Transaction Protocol (BTP), welches im Mai 2002 in der Version 1.0 von der OASIS Business Transaction Technical Commitee (BTTC) veröffentlicht wurde. Hieran waren Firmen wie Bea, IBM, Sun, HP und Oracle beteiligt.
- WS Transaction (WS-TX) sowie die WS-Coordination Spezifikation wurden im August 2002 veröffentlicht. Hieran waren Firmen wie Microsoft, IBM und Bea beteiligt.
- Activity Service der OMG (Object Management Group), welches auf dem CORBA Object Transaction Service (OTS), ebenfalls von der OMG basiert. Dieser wurde bereits Juni 2000 veröffentlicht.

3 Transaktionsverarbeitung in J2EE Systemen

In diesem Kapitel wird die Funktionsweise von Transaktionsverarbeitung gemäß der J2EE-Spezifikation untersucht, bevor im Kapitel 4 auf die Realisierung in konkreten J2EE-Produkten eingegangen wird (Frage 1 aus Kapitel 1.1).

3.1 Die Java 2 Enterprise Plattform (J2EE)

J2EE ist eine Dachspezifikation, bestehend aus einer Vielzahl an Einzelspezifikationen, welche zusammen ein Serverframework für verteilte Systeme sowie mit der EJB-Spezifikation ein Programmiermodell für die Erstellung von stabilen, transaktionalen und skalierbaren Anwendungskomponenten darstellt. Die Transaktionsverarbeitung ist Gegenstand der *Java Transaction API (JTA)*. Die Anbindung von externen Ressourcen geschieht über die *Resource Adapter* der *Java Connector Architecture (JCA)*. Ein *Resource Adapter* beinhaltet einen *Resourcemanager*, welcher die Verwaltung der an einer Transaktion beteiligten Ressourcen obliegt. Unterstützt ein *Resourcemanager* das *Recovery* von Transaktionen wird er als *Recoverable Resource* bezeichnet (z.B. ein RDBMS über JDBC wie DB2 oder Oracle) [J2EE03], [Shar01]. Aktuell ist die Version 1.4 der J2EE-Spezifikation. J2EE-Produkte müssen Transaktionen unterstützen, welche sich über mehrere Komponenten und transaktionale Ressourcen erstrecken können. Als transaktionale Ressourcen werden Verbindungen zu relationalen Datenbanksystemen über JDBC, *JMS Sessions* und Verbindungen zu einem *Resource Adapter* verstanden [JTA99].

3.2 Anforderungen an die Transaktionsverarbeitung

Im folgenden werden aus [J2EE03] die notwendigen und optionalen Anforderungen für die einzelnen Komponenten der J2EE-Plattform zusammengestellt.

3.2.1 Web-Komponenten

Notwendige Funktionalitäten, welche die J2EE-Plattform für die Web-Komponenten (JSP / Servlet) sicherstellen muss, sind:

- Der Zugriff auf die Transaktionssteuerung über `javax.transaction.UserTransaction` muss möglich sein.

- Für die Komponente müssen verteilte Transaktionen mit mehreren *Resourcemanager* und EJBs möglich sein.
- Servlet Filter und *Eventlistener* für Web-Komponenten dürfen `javax.transaction.UserTransaction` nicht benutzen.
- Die J2EE Plattform muss für diese Komponente eine Implementierung von `javax.transaction.UserTransaction` im JNDI unter `java:comp/UserTransaction` bereitstellen.
- Beim Aufruf einer EJB von einer Web-Komponente muss der Kontext der Transaktion mitgegeben werden. Ob das EJB mit demselben Transaktionskontext aufgerufen wird, hängt von ihren Transaktionsattributen (Kapitel 3.3.2) ab.
- Die Zugriffe auf transaktionale Ressourcen müssen als Teil einer Transaktion behandelt werden, wenn diese von einer Web-Komponente innerhalb eines Thread aufgerufen werden, der mit einer Transaktion assoziiert ist.
- Ein, von einer Web-Komponente, erzeugter Thread darf nicht mit einer JTA-Transaktion verknüpft werden.
- Transaktionen müssen beendet sein, bevor die *`service`* - Methode eines Servlets zurückkehrt. Der Web Container muss aktive Transaktion aufspüren und abbrechen.

Optionale Funktionalitäten sind:

- Propagation eines Transaktionskontext ist nur innerhalb derselben J2EE-Instanz notwendig. Unterstützung des *EJB interoperability protocol* ist derzeit noch nicht notwendig.
- Import eines Transaktionskontextes von einer Web-Komponente oder einem Client.
- Propagation eines Transaktionskontext über HTTP.

3.2.2 JDBC

Notwendige Funktionalitäten, welche die J2EE-Plattform für diese Komponente sicherstellen muss, sind:

- JDBC muss als transaktionaler *Resourcemanager* unterstützt werden.
- Der Zugriff auf JDBC muss von Web-Komponenten und EJBs unterstützt werden.
- Wenn JDBC-Treiber verteilte Transaktionen (XA) unterstützen, muss für die Durchführung des *Two Phase Commit* die XA-Schnittstelle des Treibers benutzt werden.

3.2.3 Threads

Notwendige Funktionalitäten, welche die J2EE-Plattform für diese Komponente sicherstellen muss, sind:

- JTA Transaktionen sollten im selben Thread gestartet und beendet werden in der die `service` - Methode aufgerufen wird. Zusätzlichen Thread*s*, welche für andere Zwecke genutzt werden, sollten nicht versuchen eine JTA-Transaktion zu starten.
- Transaktionale Ressourcen können auch von einem anderen Thread als dem der `service` Methode angefordert und freigegeben werden. Jedoch sollten die Ressourcen nicht von mehreren Threads gemeinsam benutzt werden.
- Objekte für transaktionale Ressourcen (z.B. JDBC Verbindungen) sollten nicht in statischen Feldern abgelegt werden. Die Ressourcen dürfen nur mit einer Transaktion (also auch einem Thread) gleichzeitig assoziiert werden. Das Ablegen in statische Felder fördert die ungewollte und fälschliche Benutzung von transaktionalen Ressourcen aus mehreren Transaktionen gleichzeitig.
- Web-Komponenten, welche das Single-Thread-Modell implementieren, dürfen *Top-Level* transaktionale Ressourcen in Klasseninstanz-Feldern speichern (z.B. eine JDBC Verbindung, jedoch kein `Statement`-Objekt aus der JDBC Verbindung). Der Web Container garantiert das Zugriffe auf Servlets, welche das Single-Thread-Modell implementieren, serialisiert werden. Dies bedeutet, dass der Web Container dafür Sorge trägt, dass die Web-Komponente nur von einem Thread gleichzeitig durchlaufen wird, und deshalb auch nur Teil einer Transaktion zur gleichen Zeit ist.
- In Web-Komponenten welche nicht das Single Thread-Modell implementieren, sollten transaktionale Ressourcen nicht in Klasseninstanz-Feldern gespeichert werden. Stattdessen sollten sie im Aufruf der `service` - Methode angefordert und wieder freigegeben werden.
- Web-Komponenten, welche von anderen Web-Komponenten (durch `forward` oder `include`) aufgerufen werden, sollten ebenfalls keine transaktionalen Ressourcen in Klasseninstanz-Feldern speichern.
- Enterprise JavaBeans können von jedem Thread, welcher von einer Web-Komponente benutzt wird, aufgerufen werden.

3.2.4 Transaktionsmanager

Die notwendige Funktionalität, welche die J2EE-Plattform für diese Komponente sicherstellen muss, ist die Fähigkeit zur Koordination von *Two Phase Commits* über mehrere XA-fähige JDBC-Ressourcen. Der Transaktionsmanager muss die *Java Transaction API (JTA)* unterstützen. Der *Java Transaction Service (JTS)* muss nicht unterstützt werden.

Der Transaktionsmanager muss das *Flat Transaction*-Modell unterstützen, das *Nested Transaction*-Modell muss nicht unterstützt werden.

3.2.5 JMS

Die notwendige Funktionalität, welche die J2EE-Plattform für diese Komponente sicherstellen muss ist, dass ein *JMS Provider* als transaktionaler *Resourcemanager* in eine Transaktion eingebunden werden kann.

3.2.6 Resource Adapter

Resource Adapter, welche verteilte Transaktionen unterstützten, müssen innerhalb einer Transaktion als transaktionale Ressourcen benutzt werden können. Der *Resource Adapter* muss die Schnittstelle `javax.transaction.xa.XAResource` aus der JCA implementieren.

3.2.7 Interoperabilität

Hier gibt es keine vorgeschriebene Funktionalität, welche notwendigerweise implementiert werden muss. Einige Punkte werden als optional für die Version 1.4 der Spezifikation genannt, zusammen mit dem Ausblick, dass in späteren Versionen diese vielleicht als notwendig deklariert werden:

- Keine Unterstützung für Transaktions-Interoperabilität über mehrere J2EE Produkte hinweg.
- Empfohlen wird das *IIOP Transaction Propagation Protocol* der OMG zu benutzen.

3.2.8 Connection Sharing

Notwendige Funktionalitäten, welche die J2EE-Plattform für diese Komponente sicherstellen muss, sind:

- Verbindungen (*Connections*) werden prinzipiell als gemeinsam benutzbar (*shareable*) betrachtet. Das gemeinsame Benutzen von Verbindungen resultiert in einer effektiveren Nutzung von Rechner-Ressourcen und einer besseren Performanz.
- Sollte eine Komponente eine Verbindung in einer Weise benutzen, dass sie nicht mehr *shareable* ist, muss dies im *Deployment Descriptor* des *Resource Adapter* angegeben werden. (Beispiel: geänderte Sicherheitsattribute oder unterschiedliche Isolationsstufe).

Eine optionale Funktionalität ist das *Caching* von Verbindungen. Es muss allerdings darauf geachtet werden, dass immer mit den richtigen Transaktions- und Isolationsattributen auf eine Verbindung zugegriffen wird.

3.2.9 Security

Der Zugriff auf den *Resource Adapter* muss über Sicherheits-Konfigurationen (*Principal / Credentials*) geschützt werden können. Allerdings muss innerhalb einer Transaktion nur die Sicherheitskonfiguration mit einem *Principal/Credential* unterstützt werden.

3.2.10 Java Anwendungen und Applet Clients

Für einfache Java Anwendungen und Applet Clients ist keine Unterstützung oder Partizipation an Transaktionen vorgesehen.

3.3 Enterprise JavaBeans (EJB)

3.3.1 Überblick und Funktionsweise

Die *Enterprise JavaBeans (EJBs)* stellen ein Programmiermodell für die Entwicklung und Verteilung von transaktionalen, komponentenbasierten und verteilten Geschäftsanwendungen dar. Ein Enterprise JavaBean besteht aus einer *Bean*-Klasse welche die Geschäftslogik (*SessionBean*) oder eine Datenentität (*EntityBean*) beinhaltet. Ein Client greift jedoch nicht direkt auf die *Bean* zu, sondern auf eine `EJBLocalObject`-Schnittstelle (wenn Client und EJB in derselben JVM residieren) oder eine `EJBObject`-Schnittstelle (wenn Client und EJB in unterschiedlichen JVM ausgeführt werden). Genauer gesagt greift der Client auf eine Implementierung der jeweiligen `EJBObject`-Schnittstelle zu, welche vom *EJB Container* bereitgestellt wird. Intern delegiert der *EJB Container* den Aufruf an eine Instanz der *Bean*-Klasse. Für das Erzeugen oder bei *EntityBeans* auch für das Suchen von Entitäten existiert ein zweiter Typ von Schnittstelle, dass `EJBHome` bzw. EJBLocalHome. Der Zusammenhang zwischen `EJBObject`-, `EJBHome` und der *Bean*-Klasse wird durch den XML basierten *Deployment Descriptor* hergestellt [EJB03], [Rom01].

Hinsichtlich der Transaktionsverarbeitung existieren zwei mögliche Varianten. Die *Bean Managed Transaction (BMT)*, welche über die `UserTransaction`-Schnittstelle der JTA (Kapitel 3.4) gesteuert wird, oder die *Container Managed Transaction (CMT)*, welche im *Deployment Descriptor* durch Transaktionsattribute für die kompletten EJBs oder einzelne Methoden deklariert wird. Der Anwendungsentwickler hat sich in der Folge um die Transaktionssteuerung nicht mehr zu kümmern. Jedoch muss er von Ihr Kenntnis haben, denn die einzelnen Transaktionsattribute können bei falscher Benutzung unerwünschte Ergebnisse hervorbringen.

3.3.2 Transaktionsattribute

Bei *Container Managed Transactions (CMT)* obliegt die Transaktionssteuerung dem EJB-Container. Er benutzt dazu die im *Deployment Descriptor* für die jeweilige Klasse vereinbarten Transaktionsattribute. Im folgenden Text werden die Transaktionsattribute etwas genauer darstellen. Tabelle 13 fasst die Aussagen nochmals zusammen.

NotSupported: Der EJB-Container ruft die EJB-Methode mit einem unspezifizierten Transaktionskontext auf. Wenn ein Client diese Methode des EJBs mit einer assoziierten Transaktion aufruft, muss der EJB-Container die Transaktion suspendieren, bevor er den Auf-

ruf an die EJB-Methode weitergibt. Der Transaktionskontext wird nicht an den *Resourcemanager* weitergegeben oder andere EJBs, welche Teil des Aufrufs sind. Nach der Ausführung der EJB-Methode wird die suspendierte Transaktion wieder fortgesetzt.

Required: Der EJB-Container ruft die EJB-Methode mit einem gültigen Transaktionskontext auf. Wenn ein Client diese EJB-Methode mit einer assoziierten Transaktion aufruft, benutzt der Container denselben Transaktionskontext für den Aufruf der EJB-Methode. Wenn ein Client keine assoziierte Transaktion besitzt, startet der EJB-Container automatisch eine neue Transaktion, bevor er den Aufruf an das EJB delegiert.

Der EJB-Container sorgt für die korrekte Registrierung der *Ressource Manager*, welche von der Ausführung der EJB-Methode innerhalb der Transaktion benutzt werden. Wenn auf weitere EJBs innerhalb des Aufrufs zugegriffen wird, sorgt er bei Bedarf für die Übergabe des Transaktionskontextes an diese EJBs. Nach dem Ende des Methoden-Aufrufs auf dem EJB, wird ein Commit auf die Transaktion durchgeführt, falls diese durch den Aufruf zuvor begonnen wurde.

Supports: Dieser Transaktionsmodus ist eine Verknüpfung von *Required* und *NotSupported*. Wenn ein Client eine mit *Supports* versehene EJB-Methode aufruft und über eine assoziierte Transaktion verfügt, dann handhabt der EJB-Container dies wie der Modus *Required*. Falls der Client-Aufruf über keine assoziierte Transaktion verfügt, dann wendet der Container dasselbe Verfahren an wie im Fall von *NotSupported*.

RequiresNew: Der EJB-Container ruft die EJB-Methode mit einem neuen Transaktionskontext auf. Wenn ein Client die EJB-Methode mit keiner assoziierten Transaktion aufruft, wendet der EJB-Container dasselbe Verfahren wie im Fall *Required* an. Wenn der Client über eine assoziierte Transaktion verfügt, wird diese zunächst vom Container suspendiert. Danach wird eine neue Transaktion gestartet und an die aufgerufene EJB übergeben. Nach Beendigung der Ausführung wird auf die neu erzeugte Transaktion ein Commit ausgeführt. Danach wird die vorher suspendierte Transaktion wieder mit dem Aufruf assoziiert und fortgesetzt.

Mandatory: Der EJB-Container ruft die EJB-Methode mit dem Transaktionskontext des Clients auf. Dies hat zur Folge, dass der Client-Aufruf zwingend mit einer Transaktion assoziiert sein muss. Ist dies der Fall wird vom EJB-Container dasselbe Verfahren angewandt wie im Fall *Required*. Wenn der Client-Aufruf über keine assoziierte Transaktion verfügt wird eine `javax.transaction.TransactionRequiredException` oder `javax.ejb.TransactionRequiredLocalException` geworfen.

Never: Der EJB-Container ruft die EJB-Methode mit keinem Transaktionskontext auf. Der Client darf mit keiner Transaktion assoziiert sein. Wenn dies zutrifft, dann wird vom EJB-Container dasselbe Verfahren angewandt wie im Fall *NotSupported*. Wenn der Client über eine assoziierte Transaktion verfügt, wird eine `java.rmi.RemoteException` oder eine `javax.ejb.EJBException` geworfen.

Tabelle 13 Transaktionsattribute für Container Managed Transactions

Transaktions-attribut	***Transaktion assoziiert mit Client***	***Transaktion assoziiert mit EJB-Methode***	***Transaktion assoziiert mit Ressource Manager***
NotSupported	Keine	Keine	Keine
	T1	Keine	Keine
Required	Keine	T2	T2
	T1	T1	T1
Supports	Keine	Keine	Keine
	T1	T1	T1
RequiresNew	keine	T2	T2
	T1	T2	T2
Mandatory	Keine	(Exception)	--
	T1	T1	T1
Never	Keine	Keine	Keine
	T1	(Exception)	--

3.3.3 Lebenszyklus einer EJB

Enterprise JavaBeans (EJB) sind einem Lebenszyklus unterworfen, welcher vom EJB-Container gesteuert wird. Die Art des EJBs entscheidet darüber, welche Zustände ein EJB einnehmen kann und welche Transitionen möglich sind [EJB03], [McL02].

3.3.3.1 Stateless SessionBean

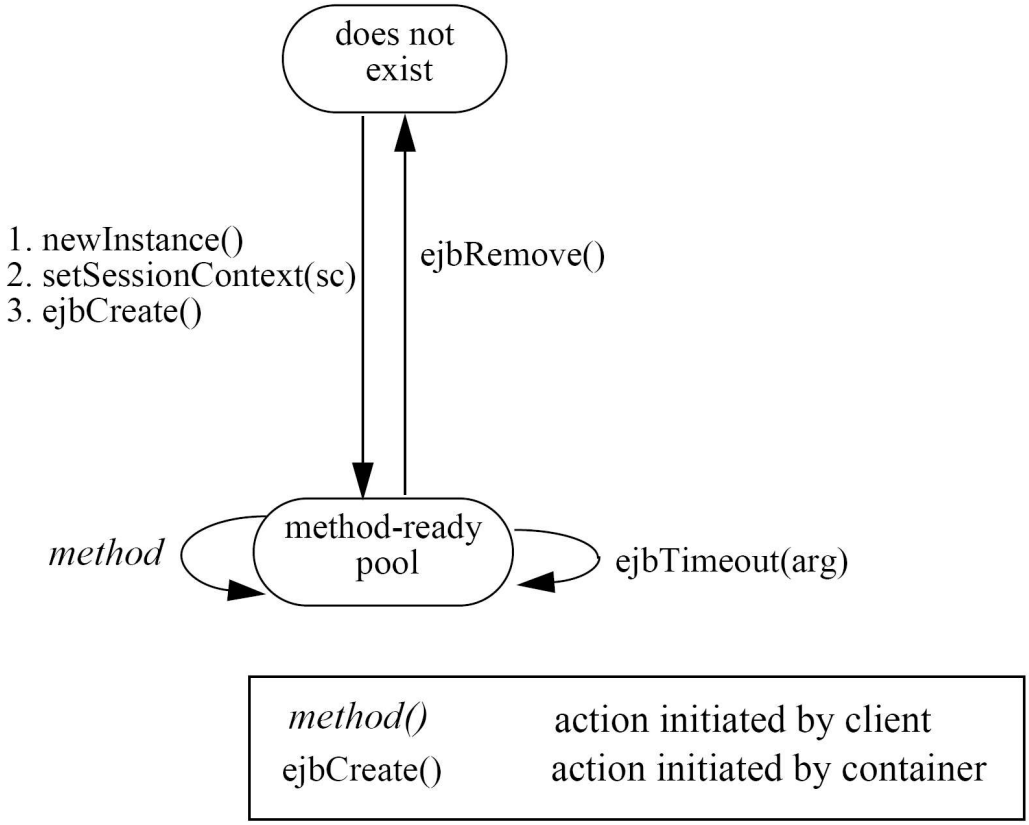

Abbildung 3 Lebenszyklus eines Stateless SessionBeans (Quelle: [EJB03])

Eine *Stateless SessionBean* beginnt ihren Lebenszyklus (Abbildung 3), wenn der EJB-Container die `newInstance` - Methode auf der SessionBean-Klasse aufruft, um eine neue Instanz zu erzeugen. Danach wird das Bean über `setSessionContext` mit dem `SessionContext` initialisiert, bevor die `ejbCreate` -Methode auf der Bean-Instanz aufgerufen wird. Die Bean befindet sich nun im Zustand *Method-Ready Pool* und kann für die Delegation von ankommenden Methoden-Aufrufen von Clients benutzt werden. Wenn der EJB-Container die Instanz nicht mehr länger benötigt (weil er den Pool verkleinern will oder selbst beendet wird) wird die `ejbRemove` -Methode auf der Bean aufgerufen und sie danach für die *Garbage Collection (GC)* freigegeben.

3.3.3.2 Stateful SessionBean

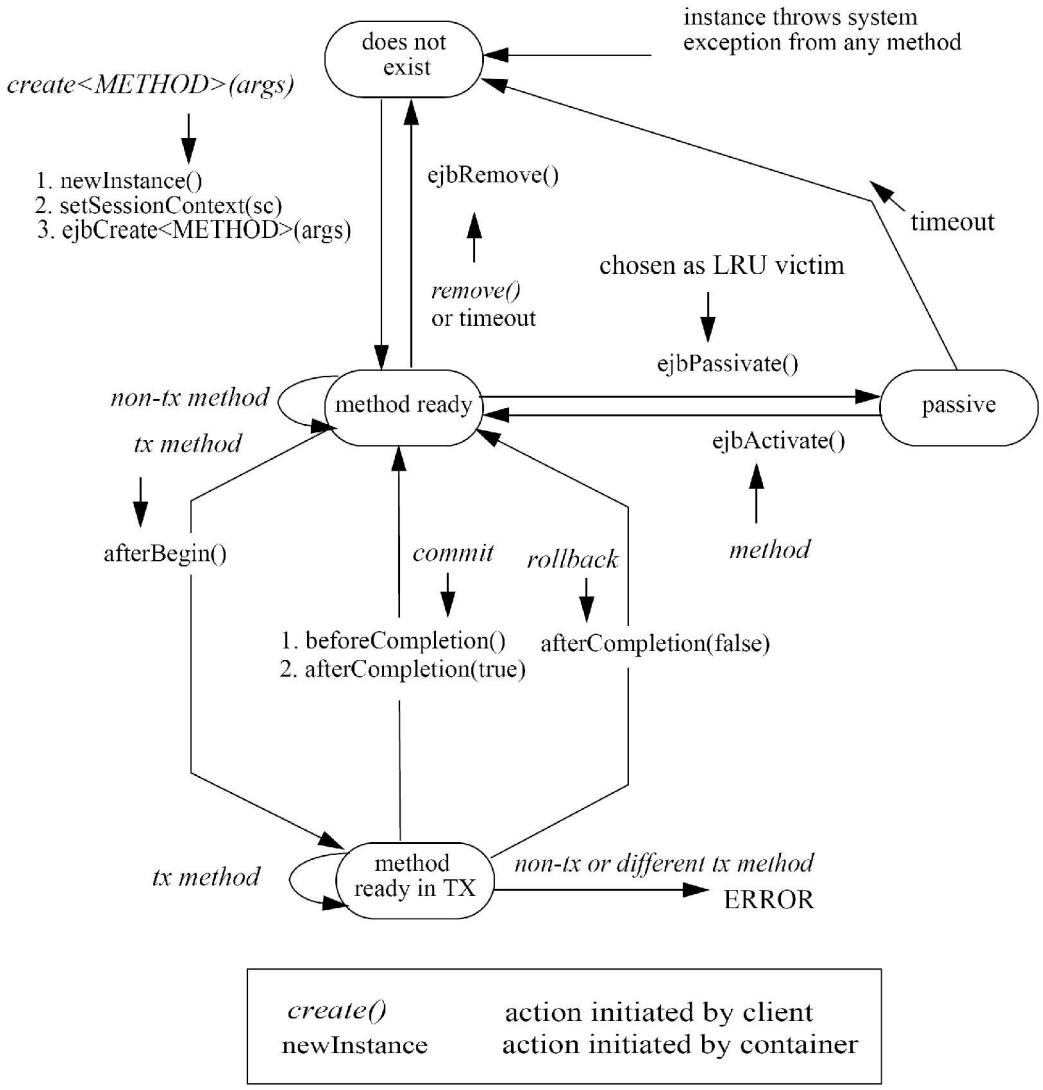

Abbildung 4 Lebenszyklus einer Stateful SessionBean (Quelle: [EJB03])

Sehr viel komplexer ist der Lebenszyklus bei einem *Stateful SessionBean* (Abbildung 4). *Stateful SessionBeans* haben - anders als die *Stateless SessionBeans* - eine Beziehung zu ihrem aufrufenden Client. Ein Client kann eine Folge von Aufrufen auf derselben Bean-Instanz durchführen. Dabei kann eine Transaktion in einer der Aufrufe begonnen und erst in einer späteren wieder beendet werden.

Der Lebenszyklus beginnt hier auch folgerichtig mit dem Aufruf der `create` -Methode auf dem `EJBHome` eines SessionBeans. Dies lässt den EJB-Container eine neue Instanz des Beans mit `newInstance` erzeugen. Danach werden, wie bei der Stateless-Variante, über `setSessionContext` das Bean mit dem `SessionContext` initialisiert, bevor auf der

Instanz die `ejbCreate` -Methode aufgerufen wird. Danach ist sie im Zustand *method-ready* und wird direkt für den verursachenden Client-Aufruf benutzt.

Auf der Bean können nun Methoden aufgerufen werden. Die EJB-Methoden werden gemäß ihren Transaktionsattributen ausgeführt. Aufrufe von EJB-Methoden ohne Transaktionskontext (*non-tx-methods*) werden im Zustand *method-ready* ausgeführt. Aufrufe von EJB-Methoden mit Transaktionskontext überführen die Bean in den Zustand *method-ready in TX*. Falls das SessionBean die `SessionSynchronization` -Schnittstelle implementiert, wird dabei die `afterBegin` -Methode auf dem SessionBean ausgeführt. Diese Methode wird aufgerufen bevor der erste Datenzugriff des Beans innerhalb der Transaktion stattgefunden hat. Nun kann das SessionBean Methoden-Aufrufe mit Transaktionskontext abarbeiten. Fehler können auftreten, wenn die Transaktionsattribute der nacheinander aufgerufenen EJB-Methoden nicht miteinander kompatibel sind (vgl. Kapitel 3.3.2).

Wenn ein Commit auf die Transaktion ausgeführt wird und die SessionBean die `SessionSynchronization` -Schnittstelle implementiert, wird vor dem tatsächlichen Commit die Methode `beforeCompletion` aufgerufen. Die Bean kann diesen Trigger verwenden um beispielsweise ihren Cache mit der Datenbank zu synchronisieren. Bei einem Rollback wird die `beforeCompletion` nicht aufgerufen. Danach führt der Container ein Commit aus und unterrichtet die Bean über `afterCompletion` von dem Ergebnis. Danach befindet sich die Bean wieder im *method ready* Zustand.

Über `ejbPassivate` kann der Container die Bean-Instanz aus dem Hauptspeicher auslagern. Dies kann nur zwischen Transaktionen geschehen. Nach einer im *Deployment Descriptor* definierten Zeitspanne kann der Container das Bean komplett entfernen, Ein Client, welcher auf die Bean-Instanz erneut zugreifen will, erhält eine `java.rmi.NoSuchObjectException` oder `javax.ejb.NoSuchObjectLocalException`. Greift ein Client vor einem Timeout auf die Bean-Instanz zu, wird die Bean wieder in den Speicher geladen und auf ihr die `ejbActivate` aufgerufen. Danach kann die Bean wieder EJB-Methoden-Aufrufe des Clients abarbeiten. Wenn der Client oder der Container ein `remove` auf dem `EJBHome` oder `EJBObject` aufruft, wird auf dem Bean `ejbRemove` aufgerufen und danach für die Garbage Collection freigegeben. Jeder nachfolgende Aufruf des Clients auf die Bean führt zu einer `java.rmi.NoSuchObjectException` oder `javax.ejb.NoSuchObjectLocalException`. Die `ejbRemove` - Methode kann nicht während der Ausführung einer Transaktion (*method-ready in TX*) ausgeführt werden. Dies führt zu einer Ausnahmebedingung (*Exception*).

3.3.3.3 EntityBean

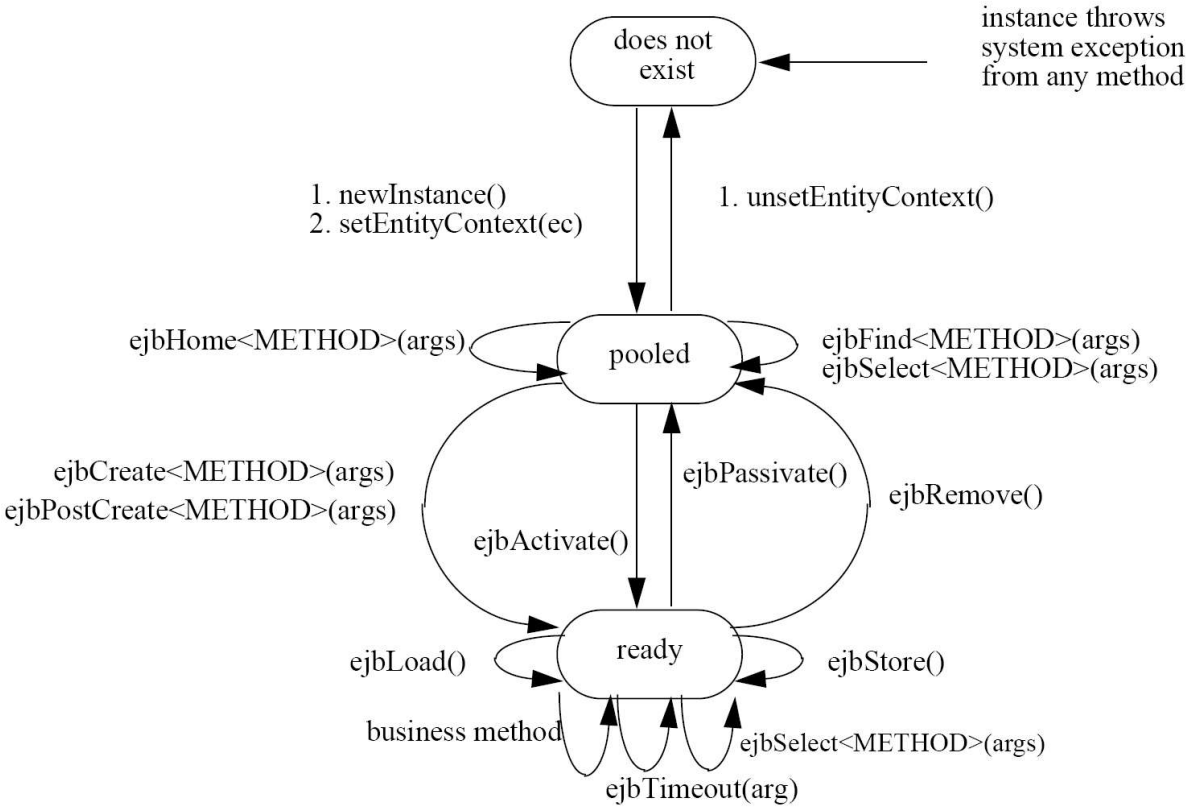

Abbildung 5 Lebenszyklus einer EntityBean (Quelle: [EJB03])

Eine EntityBean-Instanz befindet sich in einem der folgenden drei Zustände (Abbildung 5):

- Nicht existent.
- Zustand pooled. In diesem Zustand ist eine Bean-Instanz nicht mit irgendeiner Identität einer Entität assoziiert.
- Zustand *ready*. In diesem Zustand ist eine Bean-Instanz mit einer Identität einer Entität assoziiert. Bei einem EntityBean gibt es keine Unterscheidung zwischen nicht-transaktionalem Zugriff und dem Zugriff innerhalb einer Transaktion. EntityBeans werden immer innerhalb einer Transaktion ausgeführt.

Der Lebenszyklus (Abbildung 5) einer EntityBean beginnt mit dem Aufruf von `newInstance` auf ihrer Bean-Klasse, um eine neue Instanz des Beans zu erzeugen. Die Bean wird über `setEntityContext` mit dem `EntityContext` befüllt. Über ihn hat das Bean Zugriff auf Dienste des Containers oder auch Sicherheits-Informationen des Aufrufers. Danach ist das Bean im Zustand *pooled*. Während dieses Zustandes ist die Bean-Instanz nicht mit einer spezifischen Identität versehen, alle Bean-Instanzen sind demnach gleichwertig. Deshalb kann der Container auch jede beliebige Bean-Instanz aus dem Pool heranziehen, um

Aufrufe der *ejbFind* oder *ejbSelect* auszuführen. Das Bean wechselt dabei nicht seinen Zustand. Erst wenn ein Client auf eine spezifische Entität zugreifen möchte (z.B. über die *create*-Methoden des EJBHome oder bei der Iteration über die einzelnen Entitäten einer Suchanfrage), wechselt die Bean-Instanz den Zustand.

Dazu gibt es zwei Möglichkeiten. Wenn eine Entität neu erzeugt wird (über *create* von EJBHome) wird *ejbCreate* und *ejbPostCreate* aufgerufen. Wenn eine Entität schon existiert und erneut geladen werden muss (z.B. bei einer Suche über *findByPrimaryKey* des EJBHome), wird *ejbActivate* auf der Bean-Instanz aufgerufen. Danach befindet sich die Bean-Instanz im Zustand *ready* und ist mit einer spezifischen Identität versehen. In diesem Zustand kann der Container beliebig die Daten des Beans mit der Datenbank synchronisieren. Dies geschieht über *ejbLoad* und ejbStore. Ebenso können die in der Schnittstelle definierten Methoden des EJBs aufgerufen werden.

Um eine Bean-Instanz (auch während einer Transaktion) zu passivieren, ruft der Container zunächst *ejbStore* auf und danach *ejbPassivate* auf. Dabei wird die Identität der Bean-Instanz wieder zurückgesetzt und die Bean in den Zustand *pooled* versetzt. Ein ähnlicher Vorgang geschieht beim Benutzen der *remove*-Methode. Hierbei wird jedoch nur *ejbRemove* aufgerufen, bevor die Identität zurückgesetzt wird und das Bean im Zustand *pooled* ist. In diesem Zustand kann der Container das Bean jederzeit entfernen, indem er *unsetEntityContext* aufruft und danach die Bean-Instanz zur Garbage Collection freigibt.

3.3.4 Commit Optionen

Mittels der Festlegung von Commit-Optionen im *Deployment Descriptor* einer EntityBean, kann Einfluss auf die Behandlung einer Instanz zwischen Transaktionen genommen und somit die Nutzung von Ressourcen optimiert werden [EJB03], [JBoss04].

Option A: Der Container hält zwischen Transaktionen das Bean im Zustand *ready* in einem Cache. Der Container geht dabei zusätzlich von der Annahme aus, dass die Bean-Instanz den exklusiven Zugriff auf die Entität in der Datenbank hat, weshalb eine Aktualisierung der Daten am Anfang einer neuen Transaktion nicht erfolgen muss.

Option B: Der Container hält zwischen Transaktionen das Bean im Zustand *ready* in einem Cache. Der Container geht jedoch nicht wie in Option A davon aus, dass die Bean-Instanz alleinigen Zugriff auf die Entität in der Datenbank hat. Deshalb muss bei einer weiteren Transaktion entweder die Bean-Instanz mit den Daten der Entität auf der Datenbank aktualisiert werden (pessimistischer Ansatz), oder mit einem optimistischen Synchronisationsverfahren ein Rollback erzwungen werden, falls die Daten sich geändert haben.

Option C: Der Container hält zwischen Transaktionen keine Bean im Zustand *ready* im Cache. Nachdem eine Transaktion beendet wurde, wird die Bean-Instanz in den Zustand *pooled* überführt. Für eine weitere Transaktion muss das Bean wieder aus dem *Pool* geholt werden und erneut mit einer Entität aus der Datenbank versehen werden.

Die Tabelle 14 stellt die erläuterten Commit-Optionen in einer Übersicht zusammen.

Tabelle 14 Zusammenfassung der Commit Optionen mit Auswirkung auf die Bean

Option	***Schreiben der Daten auf DB***	***Bean-Instanz bleibt in „ready“***	***Bean-Instanz bleibt gültig***
Option A	Ja	Ja	Ja
Option B	Ja	Ja	Nein
Option C	Ja	Nein	Nein

3.4 Java Transaction API (JTA)

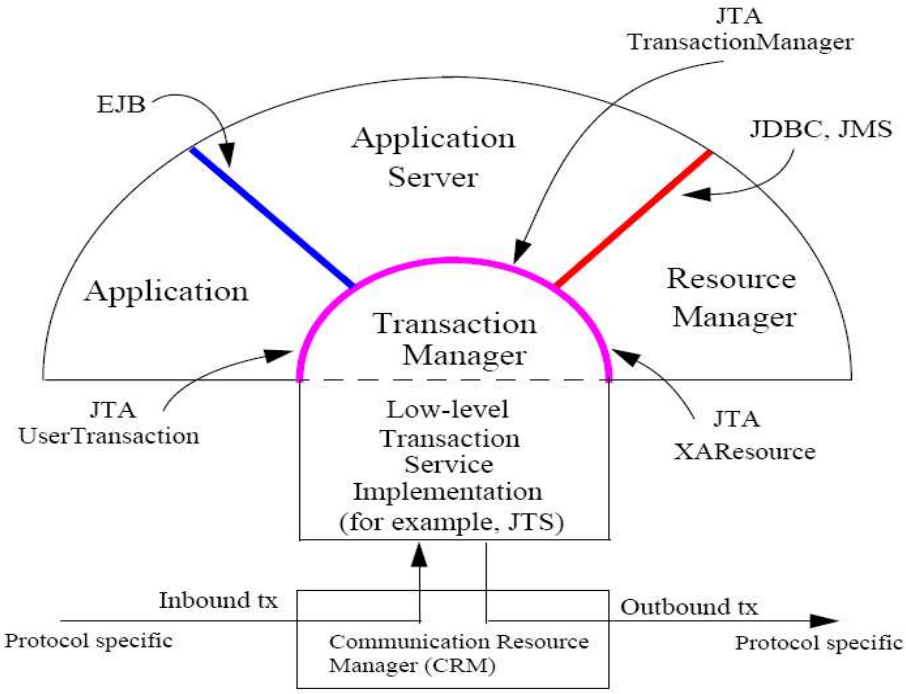

Abbildung 6 Komponenten einer verteilten Transaktion (Quelle: [JTA99])

Die *Java Transaction API (JTA)* spezifiziert eine Schnittstelle zwischen dem Transaktionsmanager und den Parteien einer verteilten Transaktion [JTA99]. Innerhalb der J2EE-Plattform sind fünf Rollen definiert (Abbildung 6):

Transaktionsmanager: Stellt Dienst- und Verwaltungsfunktionalitäten zur Verfügung, welche benötigt werden, um Transaktionsgrenzen zu steuern, transaktionale Ressourcen zu verwalten und die Verarbeitung von `Synchronization`-Objekten sowie Propagation des Transaktionskontextes zu erlauben.

Applicationserver (EJB-Container): Stellt die benötigte Infrastruktur bereit, um eine Laufzeitumgebung für Anwendungskomponenten bereitzustellen und den Transaktionszustand der EJBs zu verwalten.

Resourcemanager: Stellt den Anwendungskomponenten Zugriff auf Ressourcen über einen *Resource Adapter* zur Verfügung. Er wird vom Transaktionsmanager benutzt, um die Assoziierung von Ressourcen mit Transaktionen mitzuteilen, Festschreibung von Transaktionen durchzuführen und Wiederherstellungsarbeiten (Recovery) durchzuführen. Ein Relationales Datenbank-Managementsystem (RDBMS) stellt einen solchen *Resourcemanager* dar.

Anwendungskomponente (EJB): Eine Anwendungskomponente wird in einen Applicationserver betrieben und nutzt Dienste wie das Transaktionsmanagement über genormte Schnittstellen. Im Fall von CMT findet die Festlegung der Transaktionsgrenzen über Deklarationen im *Deployment Descriptor* statt.

Communication Resourcemanager (CRM): Ist für den Import und Export des Transaktionskontextes für Aufrufe von und nach extern zuständig. Diese Rolle wird in der JTA-Spezifikation nicht näher spezifiziert, hierzu wird auf die Spezifikation des *Java Transaction Service (JTS)* in [JTS99] verwiesen.

Die JTA definiert für die Transaktionsverarbeitung mehrere `Interface` und `Exception` - Klassen. Die Klassen sind in Abbildung 7 mit ihren Interaktionen dargestellt und werden im folgenden vorgestellt, um bei der darauf folgenden Erläuterung des Zusammenspiels der einzelnen Komponenten eine Referenzierung zu ermöglichen.

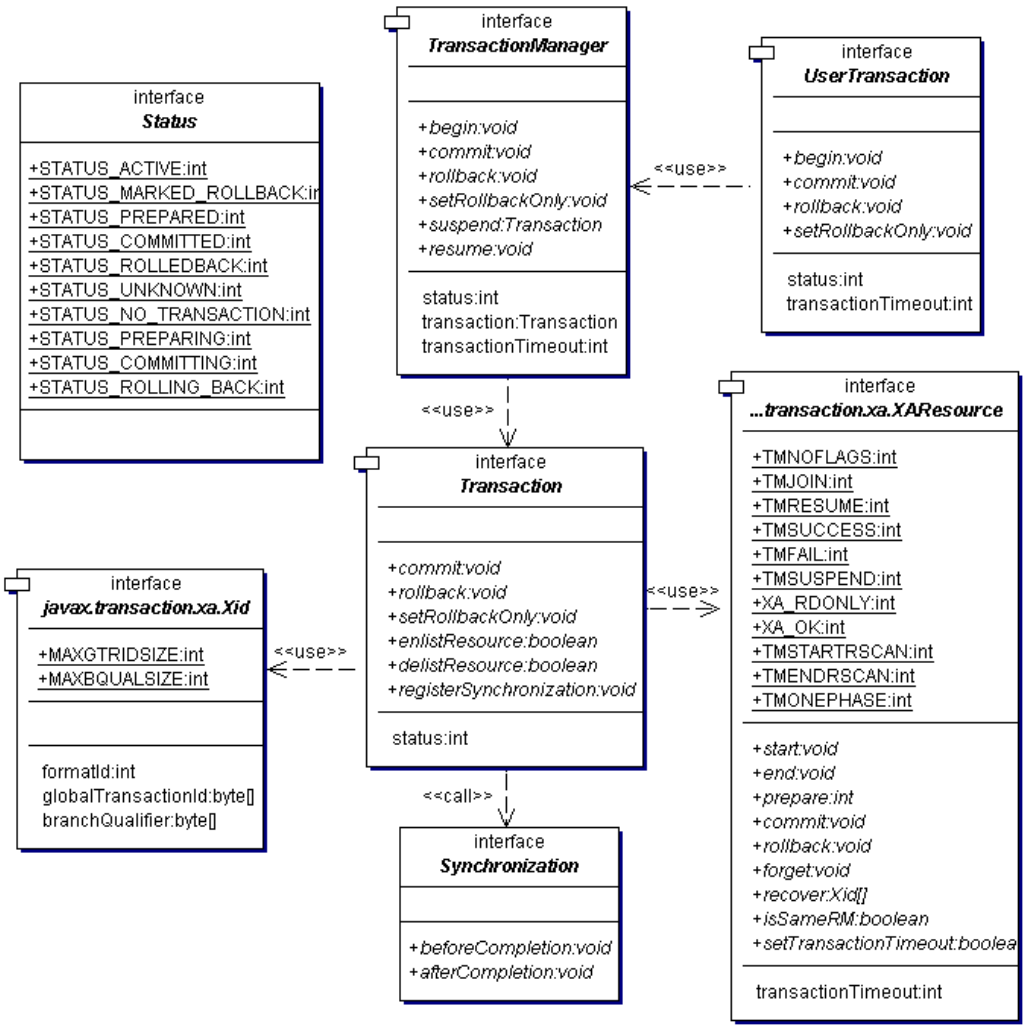

Abbildung 7 Die Java Transaction API (JTA)

Eine `UserTransaction` liefert einem Client (Java-Anwendung, Web-Komponente, SessionBean mit BMT) eine Schnittstelle zur Transaktionssteuerung. Über *`begin`* wird eine neue Transaktion begonnen und mit *`commit`* festgeschrieben, oder mit *`rollback`* zurückgesetzt. Die `UserTransaction` steuert ihrerseits für die Transaktionskontrolle den Transaktionsmanager an. Über *`status`* kann der aktuelle Status einer `UserTransaction` erfragt werden (vgl. `Status`-Schnittstelle für mögliche Werte). Über *`transaction-Timeout`* kann die maximale Dauer einer Transaktion bestimmt werden, bevor sie ungültig und zurückgerollt wird.

Der `TransactionManager` ist die Steuerzentrale für die transaktionale Arbeit. Mittels dem *`transaction`* erhält man Zugriff auf die aktuell mit dem Thread assoziierte Transaktion. Durch Aufruf von *`begin`* wird eine neue Transaktion erzeugt und mit dem aktuellen Thread assoziiert. Mit *`commit`* wird auf die aktuelle Transaktion ein Commit ausgeführt, mit

rollback wird die aktive Transaktion zurückgerollt. Die *setRollback* - Methode legt für die aktive Transaktion fest, dass bei einer Beendigung der Transaktion nur ein Rollback durchgeführt werden darf. Mit *suspend* wird die aktuell mit dem Thread assoziierte Transaktion suspendiert. Es wird die suspendierte Transaktion zurückgeben, damit der Aufrufer (z.B. EJB-Container) nach der Durchführung einer neuen Transaktion die vorherige mit *resume* wieder aktivieren kann. Auch vom `TransactionManager` kann der Status erfragt werden (vgl. `Status`-Schnittstelle für mögliche Werte), sowie der Transaktions-Timeout gesetzt werden.

Die `Transaction` - Klasse stellt den Transaktionskontext dar. Er enthält den Zustand einer Transaktion, den eindeutigen Bezeichner (`Xid`), die registrierten `Synchronization`-Listener, sowie die innerhalb einer Transaktion benutzten Ressourcen (`XAResource`). Über *commit* wird die Transaktion, auf welcher diese Methode aufgerufen wird festgeschrieben und mit *rollback* zurückgerollt. Über *registerSynchronization* wird ein `Synchronization`-Objekt bei der Transaktion registriert. Die `Synchronization`-Schnittstelle erlaubt es, interessierte Komponenten über den Zustandswechsel einer Transaktion zu informieren. In der Vorbereitungsphase eines Commits wird *beforeCompletion*, nach dem Commit wird *afterCompletion* mit dem Status des Commits als Parameter aufgerufen.

Die `Xid` - Klasse ist ein eindeutiger Schlüssel innerhalb einer Transaktion. Es ist unterteilt in *formatId*, welches darüber entscheidet, wie die beiden anderen Werte zu interpretieren sind, *globalTransactionId*, welche die Transaktion an sich identifiziert und *branchId*, welche den Zweig einer Transaktion innerhalb des *Resourcemanagers* bezeichnet. Haben zwei `XAResource`-Objekte denselben *Resourcemanager* so haben sie innerhalb der Transaktion dieselbe *branchId*. Verweisen die zwei `XAResource`-Objekte auf unterschiedliche *Resourcemanager* dann sind die *branchId* unterschiedlich und somit auch die `Xid`-Objekte.

Mittels *enlistResource* aus `Transaction` können `XAResource`-Objekte aus den `ManagedConnection`-Objekten eines *Resource Adapters*, der Transaktion hinzugefügt werden. Mit *delistResource* können die `XAResourcen` wieder abgemeldet werden, d.h. sie sind innerhalb der Transaktion nicht mehr zugreifbar, es sei denn man führt wieder *enlistResource* aus.

Eine `XAResource` referenziert den *Resourcemanager* und ist Teil einer Transaktion. Über die `XAResource` werden die Steuerinformationen an den *Resourcemanager* übergeben um die Daten, welche über die Verbindungen zum *Resourcemanager* übertragen werden, den richtigen Transaktionen zuzuordnen. Mit dem Aufruf von *start* wird die `XAResource` mit der durch die übergebene `Xid` referenzierten Transaktion assoziiert. Alle Datenzugriffe über diese `XAResource` werden nun im *Resourcemanager* zu dem durch die `Xid` referenzierten Transaktionsbranch abgelegt, solange bis *end* aufgerufen wird. Die Methoden *prepare*,

`commit` und `rollback` werden für die Durchführung des *Two Phase Commit*-Protokolls benötigt. Eine `XAResource` behält ihre in einer Xid-Session gehaltenen Datenzugriffe solange, bis sie entweder ordnungsgemäß abgeschlossen (2PC), nach einer Fehlersituation über `recover` wiederhergestellt oder mittels `forget` zum löschen markiert wurden. Die Klasse `Xid` und `XAResource` sind Java-Umsetzungen des X/Open Standards für verteilte Transaktionen.

3.5 Zusammenspiel der Komponenten

3.5.1 Übersicht der Komponenten

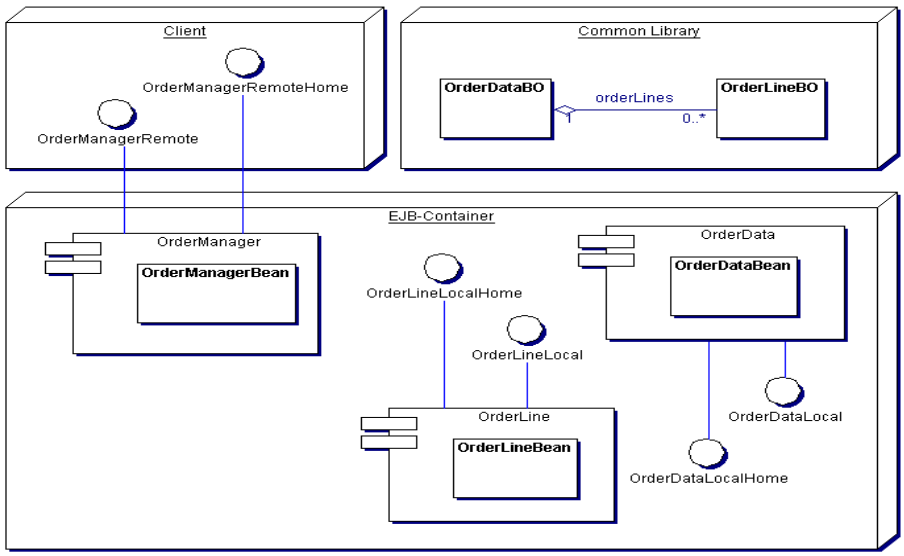

Abbildung 8 Komponenten der NewOrder-Transaktion

Nachdem im bisherigen Kapitel 3 die einzelnen Komponenten der Transaktionsverarbeitung innerhalb von J2EE-Systemen eingeführt wurden, soll nun das Zusammenspiel anhand eines Anwendungsszenarios (Abbildung 8) erläutert werden. Als Anwendungsszenario dient die Anlage eines neuen Auftrages (*NewOrder*-Transaktion) auf Basis des TERPH-Datenmodells. Im Vorfeld dieser Transaktion sind schon verschiedene andere Leseoperationen auf die Datenbank ausgeführt worden, um beispielsweise die Auftragserfassmaske für den Sachbearbeiter mit Daten zu füllen. Die Betrachtung beginnt nachdem alle Auftragsdaten erfasst wurden und die transienten Datenobjekte für die Auftragsdaten (`OrderDataBO` und `OrderLinBO`) vom Client über eine Methode von `OrderManagerRemote` an den Server

(EJB-Container) übergeben werden. Zur Förderung der Übersichtlichkeit wurden in den nachfolgenden Abbildungen Farben eingesetzt. Mit Blau sind die Klassen gekennzeichnet, welche vom Anwendungsentwickler stammen (oder aus seiner Entwicklungsumgebung). Gelb steht für die allgemeine Container-Infrastruktur. Orange für Container-Infrastruktur mit Schwerpunkt Transaktionsverarbeitung. Mit Grün ist der Persistenzbereich dargestellt.

3.5.2 Client ruft SessionBean

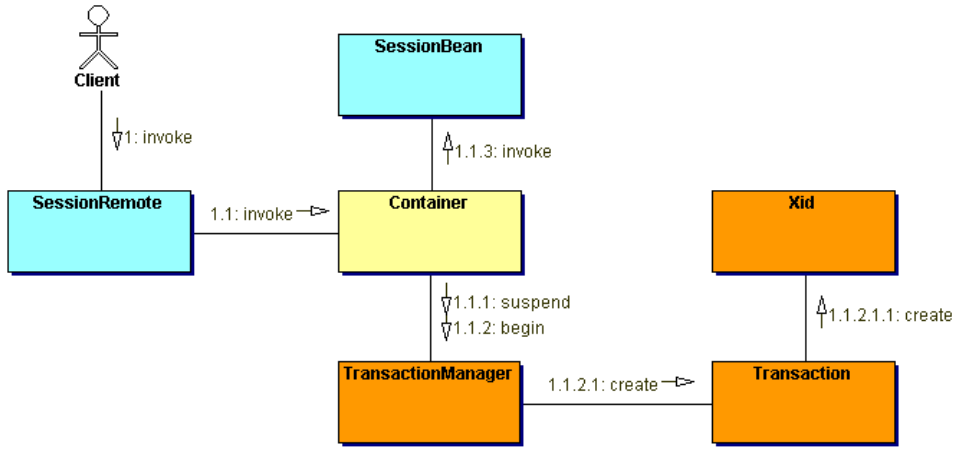

Abbildung 9 Client ruft SessionBean

Bevor der Client eine Methode auf dem `OrderManagerBean` (Stateless SessionBean) bzw. auf der `OrderManagerRemote`-Instanz aufrufen kann, muss der Client sich über das JNDI (Namensdienst der J2EE-Plattform) eine Referenz auf OrderManagerRemoteHome holen. Da der Client in einer anderen JVM ausgeführt wird als der EJB-Container muss über *`javax.rmi.PortableRemoteObject.narrow(Remote, Class)`* die richtigen Stubs und Ties an das Remote-Proxy gehängt werden. Danach ruft der Client auf `OrderManagerRemoteHome` (bzw. auf die Proxy-Implementierung) zur Erzeugung einer neuen `OrderManagerRemote`-Instanz ein *`create`* aus. Der Aufruf wird über Stubs/Ties an den J2EE-Container weitergegeben. Wenn keine Instanzen des OrderManagerBeans im Container-Pool vorhanden sind, wird eine neue Instanz (*`newInstance`*) erzeugt, der SessionContext gesetzt (*`setSessionContext`*) und *`ejbCreate`* des *`OrderManagerBean`* aufgerufen. Danach ist die Bean-Instanz bereit, um Aufrufe des Clients abarbeiten zu können. Der Container erzeugt für die `OrderManagerBean` noch eine Instanz, welche das `OrderManagerRemote`-Interface implementiert und die Client-Aufrufe an den Container und das Bean weiterleitet. Als Ergebnis erhält der Client eine Objektinstanz von `OrderManagerRemote`.

Der Client erzeugt ein Datenobjekt, welches einen Auftragskopf (`OrderDataBO`) und mehrere Auftragszeilen (`OrderLineBO`) enthält und ruft danach *`newOrder`* auf dem `OrderManagerRemote`-Objekt auf. Das Datenobjekt wird als Argument übergeben (Abbildung 9, 1). Der Aufruf wird über Stub/Tie an den J2EE-Container weitergegeben (Abbildung 9, 1.1). Der Container wertet den *Deployment Descriptor* für das `OrderManagerBean` aus, dort wird u.a. das Transaktionsattribut gelesen (in diesem Fall: Required). Der Container überprüft, ob der Aufruf schon mit einer Transaktion assoziiert ist. In diesem Fall hat der Client keine Transaktion gestartet und es ist demnach noch keine Transaktion mit dem Aufruf assoziiert. Der Container führt *`suspend`* auf dem Transaktionsmanager auf, um die mit diesem Thread assoziierte Transaktion vorübergehend auszusetzen (Abbildung 9, 1.1.1). Danach ruft er ein *`begin`* auf dem Transaktionsmanager auf, um eine neue Transaktion zu starten (Abbildung 9, 1.1.2). Der Transaktionsmanager erzeugt eine neue Transaktion (`javax.transaction.Transaction`), darüber hinaus assoziiert der Transaktionsmanager die neue Transaktion mit dem Aufruf und mit dem aktuellen Thread (Abbildung 9, 1.1.2.1). Die Transaktion erzeugt eine neue Transaktions-Identifizierung (`javax.transaction.xa.Xid`) und aggregiert diese (Abbildung 9, 1.1.2.1.1). Als Ergebnis ist eine neue Transaktion erzeugt worden mit dem Status `STATUS_ACTIVE`. Des weiteren ist die Transaktion mit dem ausführenden Thread assoziiert. Der J2EE-Container holt aus dem Pool eine freie Instanz des OrderManagerBeans und ruft darauf die Methode *`newOrder`* mit dem übergebenen Datenobjekt auf (Abbildung 9, 1.1.3).

3.5.3 SessionBean erzeugt eine neue Entität

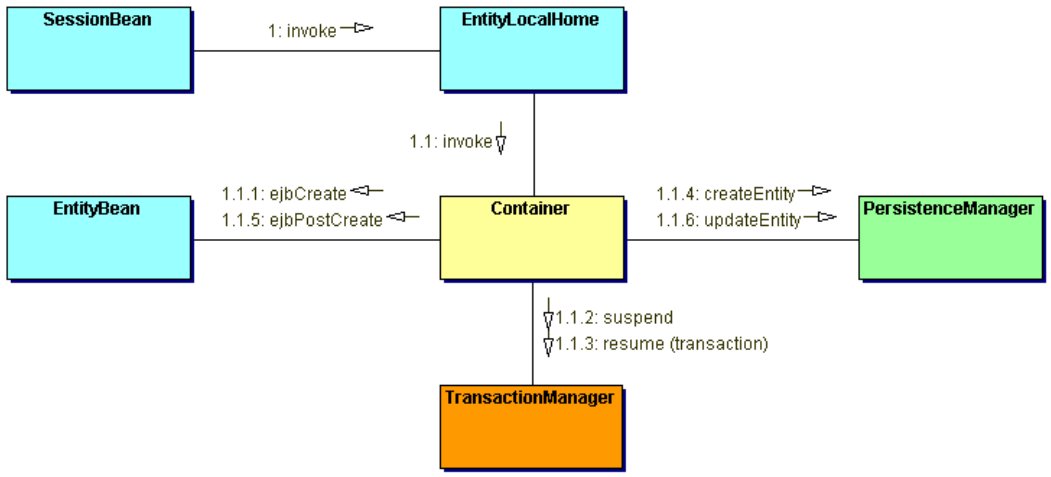

Abbildung 10 SessionBean erzeugt eine neue Entität (1)

Das OrderManagerBean holt sich über das JNDI eine Referenz auf das `EntityBean-LocalHome OrderLocalHome`. Da es innerhalb derselben JVM erzeugt wurde, muss kein Aufruf von `javax.rmi.PortableRemoteObject.narrow(Remote, Class)` erfolgen. Das OrderManagerBean ruft auf `OrderDataLocalHome` die Methode *`create`* auf und übergibt die Daten für den Auftragskopf (Abbildung 10, 1). Wenn keine freien Instanzen des OrderDataBeans im Container-Pool vorhanden sind, wird eine neue Instanz (*`newInstance`*) erzeugt und der `EntityContext` gesetzt (*`setEntityContext`*), sowie dem Pool hinzugefügt (Status: *pooled*). Bisher hat die Instanz des OrderDataBeans noch keine Verknüpfung mit einer Entität. Dies wird jetzt durchgeführt und dazu *`ejbCreate`* des OrderDataBeans aufgerufen (Abbildung 10, 1.1.1). Es wird der Primärschlüssel (*PrimaryKey)* als eindeutige Identifizierung der Entität (zumindest im Rahmen des eigenen Typs) erzeugt und dem OrderDataBean zugewiesen. Als Ergebnis ist die OrderBean mit der Entität verknüpft und befindet sich im Status *ready*.

Der Container wertet den *Deployment Descriptor* für das OrderDataBean aus. Dort wird u.a. das Transaktionsattribut gelesen (in diesem Fall: Required). Der Container überprüft, ob der Aufruf schon mit einer Transaktion assoziiert ist. In diesem Fall wurde dies schon bei dem Aufruf des OrderManagerBeans getan. Der Container führt *`suspend`* auf dem Transaktionsmanager auf, um die mit diesem Thread assoziierte Transaktion vorübergehend auszusetzen (Abbildung 10, 1.1.2). Danach holt er die mit dem Aufruf assoziierte Transaktion und ruft mit ihr als Parameter die Methode *`resume`* des Transaktionsmanagers auf (Abbildung 10, 1.1.3). Als Ergebnis ist die Transaktion des Aufrufs mit dem aktuellen Thread verknüpft.

Der Container ruft danach eine Methode (in Abbildung 10 mit createEntity bezeichnet) auf dem PersistenceManager auf, welche für das Erzeugen der Entität auf der Datenbank zuständig ist (Abbildung 10, 1.1.4 bzw. Abbildung 10, 1).

Der PersistenceManager benötigt eine Verbindung (`Connection`) zur Datenbank, um die Entität zu erzeugen. Diese Verbindung holt er sich über den `ConnectionManager` (*`allocateConnection`*), welche er auch an die `ManagedConnectionFactory` übergibt, die für das Erzeugen der `Connection` zuständig ist (Abbildung 11, 1.1).

Der `ConnectionManager` überprüft, ob es eine freie `Connection` gibt die *shareable* ist. Falls keine existiert, wird über *`createManagedConnection`* der `Managed-ConnectionFactory` eine neue `ManagedConnection` erzeugt (Abbildung 11, 1.1.1). Der `ManagedConnection` wird eine `XAResource` zugewiesen.

Der PersistenceManager ruft auf der erhaltenen `ManagedConnection` *`getXA-Resource`* auf, um die mit der `ManagedConnection` assoziierte `XAResource` zu erhalten (Abbildung 11, 1.2). Der PersistenceManager registriert mit *`enlist`* die erhaltene `XAResource` bei der aktiven Transaktion (Abbildung 11, 1.3). Die Transaktion fügt die `XAResource` zu der Liste der benutzten Ressourcen hinzu. Dabei wird mit der Methode

isSameRM(XAResource) der XAResource überprüft, ob der *Resourcemanager* für die übergebene XAResource schon vorhanden ist:

- Falls nein, wird eine neue Xid-Instanz erzeugt mit den Daten der Basis-Xid, aber einer neuen *branchId* für den *Resourcemanager* und der Liste von vorhandenen Xids für die Transaktion hinzugefügt. Auf der XAResource wird danach *start* aufgerufen und ihr als eindeutige Kennzeichnung für die Transaktion, die neu erzeugte Xid sowie TMNO-FLAGS übergeben (Abbildung 11, 1.3.1).
- Falls ja, wird die Xid-Instanz des *Resourcemanagers* aus der Liste der vorhandenen Xids für diese Transaktion geholt. Auf der übergebenen XAResource wird dann die Methode *start* mit der Xid-Instanz sowie dem Flag TMJOIN aufgerufen, welches dem *Resourcemanager* mitteilt, dass diese XAResource dem Transaktionsbranch hinzugefügt werden soll (Abbildung 11, 1.3.1).

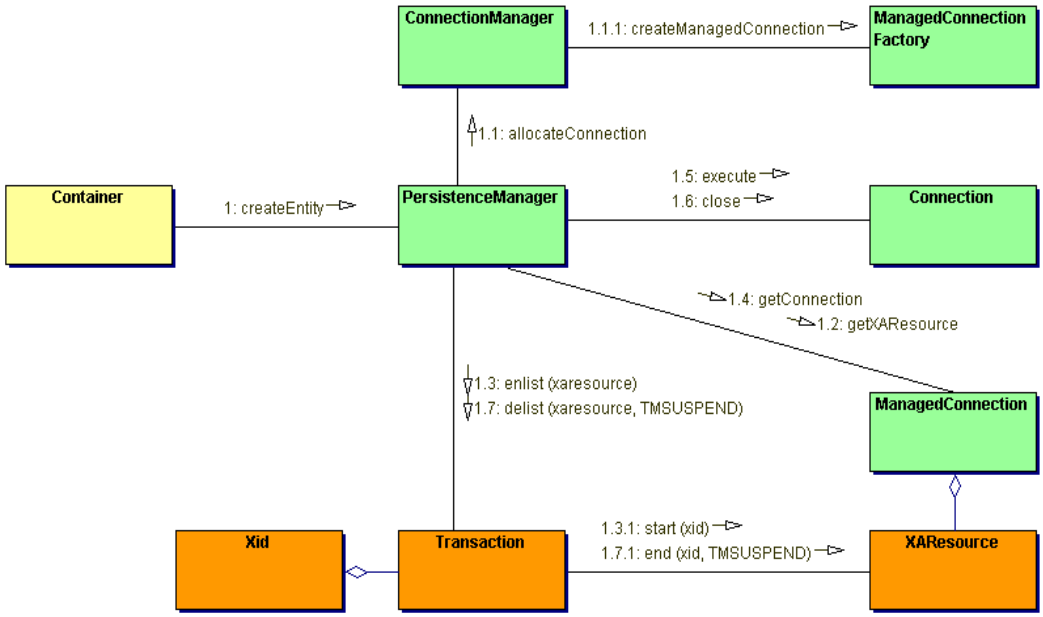

Abbildung 11 SessionBean erzeugt eine neue Entität (2)

In diesem Fall wird das erste Mal für diese Transaktion eine Verbindung geholt. Es ist deshalb keine andere XAResource mit demselben *Resourcemanager* verfügbar. Es gilt damit die erste Variante. Die ManagedConnection ist damit über die XAResource mit der Transaktion des Aufrufs assoziiert ist. Alle Interaktionen die über diese ManagedConnection oder ihre Connection-Objekte durchgeführt werden, werden im *Ressource Manager* dem Transaktionsbranch der entsprechenden Xid zugewiesen.

Der PersistenceManager holt sich nun nach erfolgreicher Ausführung von *enlist* mit *getConnection* eine Connection aus der ManagedConnection (Abbildung 11, 1.4). Falls er nicht selbst die Connection benutzt und nach Benutzung schließt, registriert

er vor dieser Aktion noch einen `ConnectionEventListener`, welcher auf solche Zustandsänderungen der Verbindung reagiert und die notwendigen Schritte durchführt. In diesem vereinfachten Fall handhabt der PersistenceManager die Verbindung selbst (ist selbst der `ConnectionEventListener`). Der PersistenceManager verfügt nun über eine mit einer Transaktion assoziierte `Connection`. Der PersistenceManager benutzt die Daten in der EntityBean-Instanz, um ein `PreparedStatement` für den Insert auf die Datenbank zu erstellen und führt dies über die `Connection` aus (Abbildung 11, 1.5). Der *Resourcemanager* erhält über die `Connection` das `PreparedStatement`. Aufgrund der Zuordnung zur `ManagedConnection` und der damit registrierten „aktiven“ `Xid` kann der *Resourcemanager* die interne Transaktion steuern. Der *Resourcemanager* führt das Einfügen der Daten durch das `PreparedStatement` mit der Isolationsstufe des *Resource Adapters* aus.

Der PersistenceManager schließt die `Connection` über *`close`* (Abbildung 11, 1.6), wodurch mit einem `ConnectionEvent` alle `ConnectionEventListener` über den Status-Wechsel informiert werden. Anschließend wird die `Connection` wieder in den `ConnectionPool` zurückgelegt. Der PersistenceManager wird als `ConnectionEvent-Listener` aufgerufen und ist dafür verantwortlich, die notwendigen Maßnahmen auf der aktiven Transaktion für diesen Statuswechsel auszulösen. Hauptsächlich muss der *Resourcemanager* darüber informiert werden, dass die Daten, die ab jetzt über dieselbe `Connection` kommen, nicht mehr auf denselben Transaktionsbranch gebucht werden dürfen, da sie gar nicht mehr zu dieser Transaktion gehören. Um dies der Transaktion mitzuteilen, ruft der PersistenceManager die Methode *`delist`* auf der aktiven Transaktion auf, und übergibt ihr die `XAResource` für die freigegebene `Connection`, sowie das Flag `TMSUSPEND` (Abbildung 11, 1.7). Dies deutet dem *Resourcemanager* an, dass dieselbe `XAResource` nochmals im Rahmen dieser Transaktion aktiviert werden könnte und lässt somit Platz für interne Optimierungen. Die Transaktion holt sich die `Xid`, welche für den *Resourcemanager* der übergebenen `XAResource` gültig ist und ruft dann *`end`* mit der `Xid` und dem übergebenen Flag `TMSUSPEND` auf(Abbildung 11, 1.7.1).

Der Container erhält nun nach dem *`createEntity`* Aufruf wieder die Kontrolle. Er führt *`ejbPostCreate`* auf dem OrderDataBean aus (Abbildung 10, 1.1.5). Die Entität ist innerhalb der Transaktion nun vorhanden, es können damit innerhalb von *`ejbPostCreate`* zusätzliche Relationen (z.B. Auftragszeilen) zu der Entität angehängt werden. Die abhängigen Entitäten müssen aber vorher ebenfalls angelegt werden. Danach veranlasst der Container den PersistenceManager die Entität zu aktualisieren (in Abbildung 10 mit updateEntity bezeichnet). Es werden wieder die Schritte aus Abbildung 11 ausgeführt, mit folgenden Unterschieden:

- `ManagedConnection` existiert schon im *Pool*, muss nicht neu erzeugt werden (1.1).
- Es wird eine `XAResource` gefunden, welche auf denselben *Resourcemanager* verweist (1.3.1). Es kommt damit die zweite Variante zum Zuge. Falls es eine bereits verwendete

`XAResource` ist wird auf diese *`start`* mit den Parametern `Xid` und `TMRESUME` ausgeführt. Falls es eine neue `XAResource` ist, welche sich nur auf dieselbe Transaktion bezieht, dann werden die Parameter `Xid` und `TMJOIN` übergeben.

- Das `PreparedStatement` enthält nun ein Update-Statement (1.5).

Im Anschluss erhält der Container wieder die Kontrolle über den Aufruf zurück. Er suspendiert nun die aktive Transaktion (über die Assoziation zum Aufruf behält er den Transaktionskontext). Danach wird die ursprüngliche Transaktion, welche in Abbildung 10 (1.1.2) suspendiert wurde, wieder aktiv gesetzt und mit dem Thread assoziiert.

3.5.4 Commit einer Transaktion beim Verlassen der SessionBean

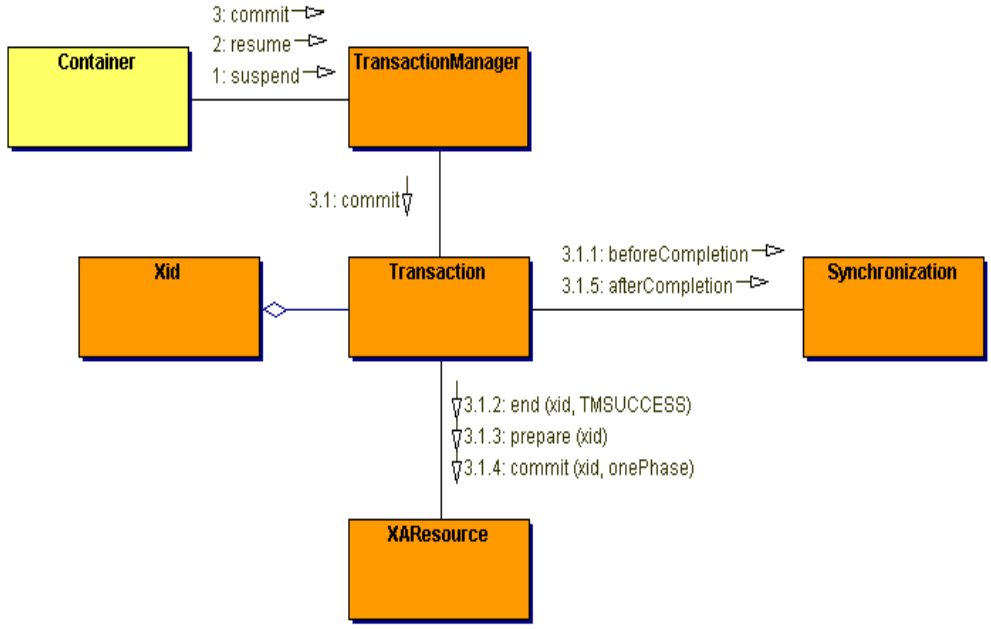

Abbildung 12 Commit einer Container Managed Transaction

Der Container setzt die Bearbeitung der mit diesem Thread assoziierten Transaktion, durch Aufruf von *`suspend`* auf dem Transaktionsmanager vorübergehend aus (Abbildung 12, 1). Danach setzt er über *`resume`* die mit dem Aufruf assoziierte Transaktion aktiv (Abbildung 12, 2). Im Anschluss initiiert der Container das Festschreiben der Transaktion mit dem Aufruf von *`commit`* auf dem Transaktionsmanager (Abbildung 12, 3). Der Transaktionsmanager delegiert den Aufruf von *`commit`* an die aktive Transaktion (Abbildung 12, 3.1).

Als erstes werden über *`beforeCompletion`* alle registrierten `Synchronization`-Objekte von der Transaktion über den bevorstehenden Commit benachrichtigt (Abbildung 12,

3.1.1). Als nächstes wir auf allen `XAResourcen` die innerhalb der Transaktion verwendet wurden die Methode *`end`* mit der jeweiligen `Xid`-Instanz und `TMSUCCESS` als Parameter aufgerufen (Abbildung 12, 3.1.2). Falls mehrere *Resourcemanager* an der Transaktion beteiligt sind, muss das *Two Phase Commit*-Protokoll für eine verteilte Transaktion eingesetzt werden. Falls nur ein *Resourcemanager* über eine oder mehrere `XAResourcen` an der Transaktion beteiligt ist, kann als Optimierung ein *One Phase Commit* durchgeführt werden.

Im Fall eines *Two Phase Commit* wird zunächst auf allen `XAResourcen` die Methode *`prepare`* mit der jeweiligen `Xid`-Instanz aufgerufen (Abbildung 12, 3.1.3). Dies veranlasst den *Resourcemanager* den durch die `Xid` referenzierten Transaktionsbranch gemäß dem *Two Phase Commit*-Protokoll zum Commit vorzubereiten. Der *Resourcemanager* gibt entweder `XA_OK` oder `XA_RDONLY` zurück. Bei einem Fehler wirft er eine entsprechende XAException. Nach diesem Aufruf ist der Transaktionsbranch im *Resourcemanager* bereit zum Commit. Im Fall eines *Two Phase Commit* wird nun auf allen `XAResourcen` die Methode *`commit`* aufgerufen, mit den jeweiligen `Xid`-Instanzen, sowie den OnePhase-Flag auf „false“ gesetzt (Abbildung 12, 3.1.4). Im Falle eines *One Phase Commit* wird das OnePhase-Flag auf „true“ gesetzt. Dies veranlasst den *Resourcemanager*, auf dem durch die `Xid` referenzierten Transaktionsbranch ein Commit auszuführen. Bei einem Fehler wird eine entsprechende `XAException` geworfen. Nach diesem Aufruf ist der Transaktionsbranch im *Resourcemanager* erfolgreich in der Datenbank festgeschrieben worden.

Nach dem Commit werden alle registrierten `Synchronization`-Objekte über *`afterCompletion`* (Abbildung 12, 3.1.5) von dem Ergebnis der Transaktion (`STATUS_COMMITED` oder `STATUS_ROLLEDBACK`) unterrichtet. Nach dem erfolgreichen Commit der Transaktion wird das `Transaction`-Object vom Transaktionsmanager zurückgesetzt und die Ressourcen freigegeben.

3.5.5 Rollback einer Transaktion beim Verlassen einer SessionBean

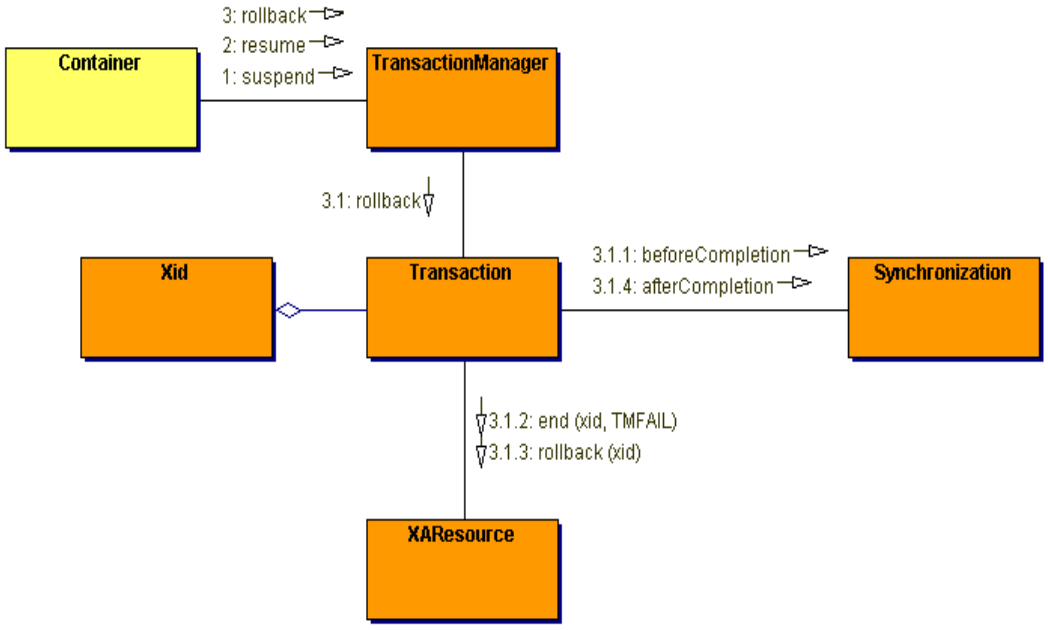

Abbildung 13 Rollback einer Container Managed Transaction

Der Container setzt die Bearbeitung der mit diesem Thread assoziierten Transaktion, durch Aufruf von *suspend* auf dem Transaktionsmanager vorübergehend aus (Abbildung 13, 1). Danach setzt er über *resume* die mit dem Aufruf assoziierte Transaktion aktiv (Abbildung 13, 2). Der Container prüft mit *getStatus* die aktive Transaktion erfolgreich auf `STATUS_MARKED_ROLLBACK` und führt daraufhin die Methode *rollback* des Transaktionsmanagers aus (Abbildung 13, 3). Der Transaktionsmanager delegiert den Aufruf von *rollback* an die aktive Transaktion (Abbildung 13, 3.1).

Als erstes werden über *beforeCompletion* alle registrierten `Synchronization`-Objekte von der Transaktion über den bevorstehenden Rollback benachrichtigt (Abbildung 13, 3.1.1). Als nächstes wir auf allen `XAResourcen` die innerhalb der Transaktion verwendet wurden die Methode *end* mit der jeweiligen `Xid`-Instanz und `TMFAIL` als Parameter aufgerufen (Abbildung 13, 3.1.2). Danach wird auf allen `XAResourcen` die Methode *rollback* mit der jeweiligen `Xid`-Instanz aufgerufen (Abbildung 13, 3.1.3). Dies veranlasst den *Resourcemanager* auf den durch die `Xid` referenzierten Transaktionsbranch einen Rollback durchzuführen. Bei einem Fehler wird eine entsprechende `XAException` geworfen. Nach diesem Aufruf ist die Transaktion im *Resourcemanager* zurückgerollt. Nach dem Rollback werden alle registrierten `Synchronization`-Objekte über *afterCompletion* (Abbildung 13, 3.1.5) von dem Ergebnis der Transaktion (`STATUS_ROLLED-BACK`) unterrichtet. Nach dem Rollback der Transaktion wird das `Transaction`-Object vom Transaktionsmanager zurückgesetzt und die Ressourcen freigegeben.

4 Transaktionsverarbeitung in J2EE Produkten

Im vorherigen Kapitel wurde die Funktionsweise von Transaktionsverarbeitung gemäß der J2EE-Spezifikation analysiert. Die gewonnenen Kenntnisse um die vorgeschriebene Funktionsweise liefert im folgenden die Basis, um die *Blackbox* der konkreten Implementierung der J2EE-Produkte zu entschlüsseln (Frage 1 aus Kapitel 1.1). Es wurden in dieser Arbeit zwei J2EE-Produkte exemplarisch ausgewählt. Zunächst wird der Opensource-Applicationserver JBoss vorgestellt, sowie das konkrete Zusammenspiel der JBoss-Komponenten, anhand des schon im vorherigen Kapitel benutzten Anwendungsszenarios beschrieben (Kapitel 4.1). Danach wird dasselbe Verfahren auf den kommerziellen Applicationserver Websphere von IBM angewendet (Kapitel 4.2).

4.1 JBoss Applicationserver

JBoss ist eine Opensource-Implementierung der J2EE-Spezifikation. Die aktuelle produktionsfähige Version ist 3.2.5, welche eine vollständige Implementierung der J2EE 1.3 Spezifikation ist. Der JBoss Applicationserver 4.0 hat vor kurzem den Kompatibilitätstest für die J2EE 1.4 – Zertifizierung erfolgreich bestanden [IW04]. Mittlerweile hat sich aus der Opensource-Bewegung die Firma JBoss Inc. entwickelt, welche die Entwicklung des JBoss Applicationservers und ergänzender Komponenten vorantreibt sowie Schulungen, Dienstleistungen und Dokumentationen zu JBoss anbietet. JBoss Inc. hat im Februar 2004 Venturekapital zum weiteren Ausbau ihrer Dienstleistungen in Höhe von ca. 10 Mio. USD erhalten.

4.1.1 JBoss Architektur und Komponenten

Die Architektur des JBoss Applicationservers basiert auf der Java Management Extension (JMX). Jedes Modul in JBoss ist ein eigenständiges JMX-Modul, welches separat verwaltet werden kann. Durch diese Eigenheit ist JBoss sehr flexibel und versteht sich selbst mehr als universelle Laufzeitumgebung für J2EE- und .NET-Anwendungen [JBoss04]. Durch den flexiblen JMX-Ansatz können auch selbst geschriebene Module einfach in JBoss integriert werden. Die wichtigsten Basismodule von JBoss werden im folgenden Text kurz erläutert.

JBossServer: Diese Komponente ist die Basis-Ablaufumgebung. Aufgebaut mittels des JMX-Ansatzes enthält der JBossServer neben den J2EE-Dienst-Modulen auch die Container für die EJBs. Zusätzlich zu den Anforderungen der Spezifikation fallen drei Punkte auf:

- EJBProxy-Klassen welche das Bindeglied zwischen den EJB-Schnittstellen und der Bean darstellen werden dynamisch zur Laufzeit erzeugt. Bei JBoss ist dies über ein *Dynamic Proxy*-Objekt realisiert, welches die benötigten Interfaces mittels Reflection einfügt.
- Spezifische Stub- und Tie – Klassen werden für die EJBs nicht benötigt, da der Aufruf als allgemeines Invocation-Objekt über den JBoss-ClientContainer an den Applicationserver übertragen wird. Es ist keine Ausführung eines EJB-Compilers notwendig, welcher die entsprechenden Klassen als zusätzlichen Entwicklungsschritt generiert.
- Hot-Deployment ermöglicht es, eine neue J2EE-Applikation (EAR) zu installieren, ein Update zu einer bestehenden Anwendung einzuspielen oder auch eine Konfigurationsänderung zu aktivieren, ohne den Server neu starten zu müssen. Das Intervall für die Suche nach neuen oder geänderten Dateien kann konfiguriert werden. Bei einem Update wird für ein neues EAR oder JAR automatisch das alte EAR/JAR deinstalliert und das Neue installiert.

In JBoss sind standardmäßig folgende Container definiert:

- Standard CMP 2.x EntityBean
- Clustered CMP 2.x EntityBean
- Standard CMP 2.x EntityBean with cache invalidation
- Instance Per Transaction CMP 2.x EntityBean
- Standard CMP EntityBean
- Clustered CMP EntityBean
- Instance Per Transaction CMP EntityBean
- Standard Stateless SessionBean
- Clustered Stateless SessionBean
- Standard Stateful SessionBean
- Clustered Stateful SessionBean
- Standard BMP EntityBean
- Clustered BMP EntityBean
- Instance Per Transaction BMP EntityBean
- Standard Message Driven Bean

Das Verhalten eines Containers lässt sich durch das einklinken von Interzeptoren (*Interceptors*) beeinflussen. Interzeptoren arbeiten nach dem *Command Pattern* [Gam96]. Interzeptoren implementieren eine spezielle `Interceptor` - Schnittstelle, welches nur eine öffentliche Methode definiert. Die `jboss-service.xml` Konfigurationsdatei definiert für jeden Container die Art und Reihenfolge der Interzeptoren, sowie noch weitere Optionen. Es ist so möglich in den Aufruf zwischen Client und Container beliebige Arten von Interzeptoren einzuklinken.

JBossMQ: Diese Komponente ist ein JMS-kompatibler *Messaging Service*. Die Ursprünge hat JBossMQ in einer frühen Opensource-Implementierung der JMS-Spezifikation unter dem Namen spyderMQ (April 2000). Derzeit unterstützt JBossMQ folgende Funktionalitäten:

- *Point-to-Point Message*-Topologie
- *Publisher/Subscribe Message*-Topologie
- *Durable/Transient Subscribers*
- *Resourcemanager* mit Unterstützung von verteilten Transaktionen
- Bei den Transportschichten werden folgende Varianten unterstützt:
  - RMI (*Remote Method Invocation*) : Es wird RMI als Kommunikationsinfrastruktur benutzt.
  - OIL (*Optimized Invocation Layer*) : Es werden direkt TCP/IP-Sockets benutzt.

JBossTX: Diese Komponente ist ein Transaktionsmanager gemäß JTA der J2EE 1.4 Spezifikation. Auf diese Komponente wird in Kapitel 4.1.2 detailliert eingegangen.

JBossCMP: Diese Komponente ist der *Persistence Manager* für *Container Managed Persistence (CMP)* und existiert schon seit JBoss Version 1.0 unter dem Name JAWS (*Just Another Web Storage*). Im Kern ist JBossCMP ein Objekt-Relationaler Mapper (ORM) welcher Objekte auf Tabellen in relationalen Datenbank-Managementsystemen (RDBMS) abbilden kann. JBossCMP unterstützt *Connection-Pooling* mittels dem „Minvera JDBC Connection Pooling“ Modul. Zusätzlich zur Spezifikation hat JBossCMP folgende Funktionalität:

- Erzeugen von Tabellen beim Deployment
- Abbildung von Java-Typen auf DB-Datentypen sind flexibel konfigurierbar (Mapping-Definitionen sind für viele Datenbanken vorhanden, u.a. Oracle, SQLServer, DB2, Sybase, PointBase, Cloudscape, HypersonicSQL, PostgresSQL, mySQL, MaxDB).
- Definition von Load-Groups. Pro Load-Group werden die Felder angegeben, welche beim Zugriff auf die EntityBean geladen werden sollen. So lässt sich die Performanz des Persistenzzugriffs durch das Laden der wirklich benötigten Attribute optimieren.

JBossSX: Diese Komponente ist ein Sicherheitsdienst, welches die *Java Authentification and Authorisation Services (JAAS)* unterstützt. Die J2EE-Spezifikation definiert ein deklaratives, auf Rollen basiertes Sicherheitsmodell. Jedoch wird nicht spezifiziert, wie dieses Modell implementiert werden soll. Die Implementierung von JBossSX basiert auf JAAS, geht aber in einigen Bereichen darüber hinaus.

JBossCX: Diese Komponente ist eine Implementierung der *Java Connector Architecture (JCA)*. Das JBossCX Modul interagiert stark mit dem JBoss Transaktionsmanager (JBossTX) und dem JBoss Sicherheitsdienst (JBossSX). Die Ressourcen sind über JNDI zugreifbar.

JBossWeb/Tomcat: Diese Komponente integriert die Servlet-Laufzeitumgebung Tomcat aus dem Apache-Projekt als Web-Container in den JBoss Applicationserver. Damit können in JBoss auch WAR-Dateien mit Web-Komponenten installiert werden und es wird eine Integration mit dem Apache-Webserver über Tomcat möglich. Tomcat enthält neben der Funktionalität für die Installation von WAR-Dateien und der stabilen und schnellen Ausführung von Servlets den Jasper JSP-Compiler, unterstützt Session-Clustering und bietet die Möglichkeit der Integration, sowie von Administration und Monitoring über JMX.

JBossIIOP: Diese Komponente implementiert einen Object Request Broker (ORB) gemäß CORBA Spezifikation der OMG.

JBossNaming: Diese Komponente stellt einen Namensdienst gemäß dem JNDI der J2EE-Spezifikation bereit.

4.1.2 Transaktionsmanager in JBoss

4.1.2.1 Überblick

Der JBoss-Container spricht seinen Transaktionsmanager über die standardisierten JTA-Schnittstellen an. Damit ist der Container unabhängig von der Implementierung des Transaktionsmanagers. Im Lieferumfang von JBoss sind zwei Transaktionsmanager enthalten. Es wäre aber auch denkbar, eigene Komponenten zu implementieren und JBoss als Transaktionsmanager bekannt zu machen [JBoss04]. Ebenso könnten Integrationsadapter für andere Transaktionsmanager geschrieben werden. Technisch denkbar wäre beispielsweise eine JTA-Integrationsadapter-Implementierung für den CICS-Transaktionsmonitor zu realisieren (zu Anbindungsmöglichkeiten an CICS vgl. [Hors00]).

Gemäß JTA muss der Transaktionsmanager die Schnittstelle `javax.transaction.TransactionManager` implementieren und sich beim Container-Startup unter dem JNDI-Namen `java:/TransactionManager` registrieren. Falls der Transaktionsmanager auch Kontext-Propagation über RMI bzw. JRMP unterstützen soll, müssen bei JBoss noch zwei weitere Schnittstellen implementiert werden. Da hierfür noch keine standardisierten Schnittstellen existieren, sind dies JBoss-Schnittstellen:

- `org.jboss.tm.TransactionPropagationContextFactory` wird benutzt, wenn ein Transaktionskontext über einen Remote-Call übertragen werden soll.
- `org.jboss.tm.TransactionPropagationContextImporter` wird benutzt, wenn ein Transaktionskontext von einem Remote-Call importiert werden soll.

4.1.2.2 Interposing

Verteilte Transaktionen können mehrere unterschiedliche JVMs bzw. Maschinen umspannen. Pro Knoten kann es mehrere Ressourcen mit zugehörigem *Resourcemanager* geben, welche an der Transaktion beteiligt sind. Im Normalfall müssten während des *Two Phase Commit*-Protokolls zu jeder Ressource viele langsame Aufrufe über Prozessgrenzen oder das Netzwerk gemacht werden. Um dieses inperformante Verhalten zu umgehen, wird das Verfahren *Interposing* benutzt. Dazu folgendes Beispiel:

Es gibt einen Server A und einen Server B. Die Transaktion wird von Server A gestartet. Auf Server A werden zwei Ressourcen in die Transaktion einbezogen, auf Server B wären es fünf. Ohne Interposing müsste jede einzelne der fünf Ressourcen einzeln über das Netz angesprochen werden. Mit Interposing delegiert der Transaktionsmanager des Server A die Koordination der Ressourcen auf Server B an den dortigen Transaktionsmanager. Dieser registriert sich dabei als Ressource bei der verteilten Transaktion anstelle der tatsächlichen Ressourcen von Server B. Für den Transaktionsmanager auf Server A bedeutet dies, dass die verteilte Transaktion nur noch die zwei lokalen Ressourcen beinhaltet, sowie eine Ressource auf Server B. Bei einem Prepare bzw. Commit ruft der Transaktionsmanager von Server A die entsprechenden Operationen auf den registrierten Ressourcen auf. Der Transaktionsmanager von Server B koordiniert für die jeweiligen Aufrufe seine lokalen Ressourcen und kumuliert das Ergebnis wieder in Richtung Server A. Der Transaktionsmanager auf Server A wird in einem solchen Szenario *Superior*-Transaktionsmanager, der Transaktionsmanager auf Server B *Subordinate*-Transaktionsmanager genannt. Der *Subordinate*-Transaktionsmanager kann jedoch wieder *Superior*-Transaktionsmanager für einen anderen Transaktionsmanager sein. Das Interposing lässt sich also hierarchisch fortsetzen.

4.1.2.3 Standard-Transaktionsmanager in JBoss

Der Standard-Transaktionsmanager in JBoss ist optimiert auf den Einsatz innerhalb derselben JVM. Spezifikationsgemäß unterstützt er nur das *Flat Transaction*-Modell und keine *Nested Transactions*. Er hat nach [JBoss04] zwei Einschränkungen:

- Für Transaktionen werden keine Logs geschrieben. Dies hat zur Folge, dass kein automatisches *Recovery* nach einem *Servercrash* erfolgen kann.
- JBoss unterstützt zwar Transaktionskontext-Propagation über Remote-Schnittstellen, jedoch nicht über JVM-Grenzen hinaus. Die Koordination der Transaktion und der zugehörigen `XAResourcen` muss deshalb innerhalb derselben JVM geschehen. Durch *Interposing* lassen sich Transaktionen über mehrere JVMs ausdehnen.

4.1.2.4 Tyrex Transaktionsmanager

Tyrex ist ein Transaktionsmanager auf Basis von CORBA, welche nicht unter den obigen Einschränkungen des JBoss Standard Transaktionsmanagers leidet. Es ist eine Implementierung der OTS Spezifikation und benutzt als Kommunikationsprotokoll IIOP [Tyr04]. Für JBoss wurde ein spezielles JMX-MBean (`org.jboss.tm.plugins.tyrex.TransactionManagerService`) zur Integration geschrieben, wie es oben schon für CICS vorgeschlagen wurde. Diese MBean macht es möglich, den Tyrex Transaktionsmanager als JBoss-Modul zu betreiben [JBoss04].

4.1.3 Zusammenspiel der Komponenten

Das im Kapitel 3.5 benutze Anwendungsszenario zum Anlegen eines neuen Auftrages (*NewOrder*-Transaktion), an dem das Zusammenspiel der einzelnen Spezifikationen und Komponenten dargestellt wurde, soll nun dazu dienen, die Arbeitsweise der JBoss-Komponenten zu beleuchten. Ausgangspunkt ist ein Client (Java-Applikation), welcher über JVM-Grenze einen Aufruf der *`newOrder`* - Methode auf dem Stateless SessionBean OrderManager durchführt. Dort werden innerhalb einer *Container Managed Transaction* die Entitäten für *OrderData* und *OrderLine* angelegt und danach zum Client zurückgekehrt.

Alle Aufrufe einer Java-Anwendung hin zu einem Container werden bei JBoss über ein *Dynamic Proxy* angenommen (erzeugt von der Klasse im Tag `<proxy-factory>` aus Anhang A, Listing A.1). Das *Dynamic Proxy*, welches über Reflection die entsprechende EJBObject- bzw. EJBHome-Schnittstelle implementiert, gibt den Aufruf an den `Client-Container` weiter. Dieser baut aus dem lokalen Methodenaufruf eine Instanz der Klasse `Invocation`, welches die Marker-Schnittstelle `java.io.Serializable` implementiert und deshalb über JVM-Grenzen übergeben werden kann. Es fügt die Methodensignatur des Aufrufs, die Parameter, sowie einige JBoss-spezifische Informationsattribute hinzu. Danach delegiert er das `Invocation`-Objekt an den ersten in der Konfiguration (`jboss-service.xml`) definierten Interzeptor (Anhang A, Listing A.1).

Für die EJBHome und das EJBObject gibt es unterschiedliche Konfigurationen der Interzeptoren. In der Standard-Konfiguration unterscheiden sie sich jedoch nur durch den ersten Interzeptor `org.jboss.proxy.ejb.HomeInterceptor` oder `org.jboss.proxy.ejb.StatelessSessionInterceptor`. In diesen Interzeptoren werden jeweils spezifische Aktionen durchgeführt. Beispielsweise werden im `HomeInterceptor` Methodenaufrufe behandelt, für die nicht an den Container gegangen werden muss, sondern die Arbeit

lokal erledigt werden kann (z.B. t*oString()*, *hashCode()*, *getEJBHandle()*). Diese „lokalen Methoden“ genannten Aufrufe, werden im `HomeInterceptor` direkt ausgeführt und kehren ohne Durchlauf der anderen konfigurierten Interzeptoren wieder zum Aufrufer zurück. Ein anderer Typ von Behandlung innerhalb der `HomeInterceptoren` ist das Abspeichern von zusätzlichen Informationsattributen in die `Invocation` (z.B. bei *remove(PrimaryKey)*).

Im `SecurityInterceptor` werden der *Principal* und die *Credentials* des aktiven Benutzers in die `Invocation` eingepackt. Im `TransactionInterceptor` wird überprüft, ob schon eine Transaktion (`UserTransaction`) mit dem aktuellen Thread assoziiert ist. Wenn vorhanden, wird diese dem `Invocation`-Objekt hinzugefügt. Der `Invoker-Interceptor` überprüft, ob die JVM des Aufrufers mit der Ziel-JVM des Aufrufs übereinstimmt. Falls dem so ist, wird das `Invocation`-Objekt über eine Speicherreferenz an den Container übergeben. Falls der Aufruf ein Remote-Aufruf über JVM-Grenzen ist, wird das `Invocation`-Objekt serialisiert und per RMI an die Container-JVM übertragen.

In der Container-JVM angelangt, wird das `Invocation`-Objekt deserialisiert und auf Basis der Informationen des `Invocation`-Objektes zum *Standard Stateless SessionBean*-Container (Anhang A, Listing A.2) delegiert. Für jeden EJB-Typ existiert eine Standard-Konfiguration an die der Aufruf delegiert wird. Dieses Standardverhalten kann über den JBoss-spezifischen *Deployment Descriptor* geändert werden. Zu jedem Eintrag von `EJBName` kann der Ziel-Container angegeben werden.

Der Container ruft nun zunächst die konfigurierten Interzeptoren auf, beginnend mit dem ProxyFactoryFinderInterceptor. Auf Basis der im `Invocation`-Objekt gespeicherten Informationen, wird die `ProxyFactory` zu der `Invocation` ermittelt und im Container gesetzt Die `ProxyFactory` ist EJB-typspezifisch (Stateless SessionBean, Stateful SessionBean, EntityBean) und wird später zur Erzeugung der EJBProxy-Objekte für das Bean benutzt. Danach geht das `Invocation`-Objekt an den nächsten konfigurierten Interzeptor. Im `LogInterceptor` wird jeder Aufruf geloggt, falls in der Konfiguration der Wert von `<call-logging>` auf den Wert „true“ gesetzt wird. Im `SecurityInterceptor` wird der Aufruf anhand des *Principals* und der *Credentials* im `Invocation`-Objekt auf seine Berechtigung hin überprüft.

Bei einer CMT – wie in diesem Fall – wird das `Invocation`-Objekt als nächstes an `TxInterceptorCMT` übergeben. Dieser Interzeptor stellt bei JBoss die Ansteuerung des Transaktionsmanagers dar. Er ist die Komponente, welche in Kapitel 3.5 als Container bezeichnet wurde. Hier wird überprüft, mit welchem Transaktionsattribut der Aufruf auszuführen ist, ob schon eine Transaktion mit dem Aufruf assoziiert ist und diese übernommen werden soll, oder wie in diesem Fall, keine Transaktion existiert und über den Transaktionsmanager eine neue Transaktion begonnen werden soll. Nachdem die Transaktion erzeugt (vgl. Kapitel 3.5.2) und mit dem aktuellen Thread assoziiert wurde, wird die Transaktion auch noch

in das `Invocation`-Objekt gesetzt. So ist sichergestellt, dass während des gesamten Aufrufs dieselbe Transaktion benutzt wird, zumindest solange nicht eine neue Transaktion in Folge des Transaktionsattributes *RequiresNew* begonnen wird. Wie in der J2EE-Spezifikation gefordert, unterstützt JBoss nur das *Flat Transaction*-Modell. Deshalb genügt ein einfaches Attribut an dem `Invocation`-Objekt vollauf, um die Transaktion durch den restlichen Aufruf zu schleusen, da sobald eine neue Transaktion begonnen wird, die alte Transaktion suspendiert wird bis die neue Transaktion beendet wurde. Danach wird die alte Transaktion wieder mit dem ausführenden Thread assoziiert (*`resume`*) und weitere Transaktionsoperationen ausgeführt.

Als nächstes folgt der `MetricsInterceptor`. Er sammelt Daten über den Aufruf (z.B. *Principal*, Transaktion, Zeit und Name des Messpunktes) und publiziert diese über eine JMS-Message an das Topic t*opic/metrics*. Hierüber können die JMS-Messages für weitere Auswertungen und Statistiken herangezogen werden.

Der `StatelessSessionInstanceInterceptor` propagiert den *Principal* an ein `Context`-Objekt und holt sich den richtigen Instanzen-Pool für den aufzurufenden Bean-Typ (in diesem Fall ist dies der Instanzen-Pool für OrderManagerBean). Über den gesetzten Kontext erhält der Aufruf einen Zugriff auf den Bean-Pool (vom Typ `org.jboss.ejb.plugins.StatelessSessionInstancePool`). Eine Konfiguration für einen Cache muss nicht erfolgen, da Stateless SessionBeans keine Identitäten haben (im Gegensatz zu Stateful SessionBeans oder EntityBeans) und somit immer direkt aus dem Pool genommen werden können. Ebenfalls benötigen wir keinen PersistenceManager, da Stateless-SessionBeans nicht persistiert werden.

Nachdem alle konfigurierten Interzeptoren durchlaufen wurden, wird als letztes wieder der Container über einen am Beginn registrierten `ContainerInterceptor` aufgerufen. Hier wird nun aus dem `Invocation`-Objekt das Reflection-Objekt der aufgerufenen Methode aus OrderManagerRemote geholt und die dazu passende Methode des OrderManagerBeans ermittelt. Danach wird eine Instanz des OrderManagerBeans aus dem Pool geholt und die ermittelte Methode mittels Reflection auf der OrderManagerBean-Instanz aufgerufen.

In der *`newOrder`* -Methode der OrderManagerBean-Instanz angekommen, werden mittels der im Aufruf übermittelten Datenobjekte (`OrderDataBO` und einer Liste von `OrderLineBOs`) neue Entitäten auf der Datenbank angelegt. Beim Aufruf von *`create`* auf dem `OrderDataLocalHome` wird, ähnlich wie im Fall des Clients, auf ein *Dynamic Proxy* zugegriffen, welche die relevanten Schnittstellen per Reflection implementiert. Von dieser Proxy-Instanz wird der Aufruf allerdings nicht an den `ClientContainer` weitergeleitet, sondern an das `LocalHomeProxy`. Die Aufgabe dieser Klasse ist dabei fast dieselbe: lokale Methoden für die keine Bean-Instanz von OrderDataBean benötigt wird sofort durchzuführen und ohne Lauf durch die restlichen Interzeptoren wieder zum Aufrufer zurückzukehren. In der

sich anschließenden `BaseLocalProxyFactory` wird das `LocalEJBInvocation`-Objekt erstellt und mit den notwendigen Werten belegt (ähnlich wie das `Invocation`-Objekt vom Client). Dann wird es an den `EntityContainer` übergeben, welcher zunächst die konfigurierten Interzeptoren auf dem `LocalEJBInvocation`- Objekt aufruft. Für den `ProxyFactoryFinderInterceptor`, `LogInterceptor`, `SecurityInterceptor`, `TxInterceptorCMT` und `MetricsInterceptor` wird dieselbe Logik durchlaufen wie bei dem vorherigen OrderManager-SessionBean-Aufruf.

Ein neuer Typ eines Interzeptors wird mit dem `EntityCreationInterceptor` aufgerufen. Im `EntityCreationInterceptor` wird die ankommende `LocalEJBInvocation` in zwei Aufrufe entlang der restlichen Interzeptoren aufgeteilt. Zuerst wird über den *`invokeHome`* - Zweig der Interzeptoren bis zum CMP-Proxy des OrderBeans (noch mit keiner Identität versehen) delegiert, um dort die *`ejbCreate`* - Methode aufzurufen. Danach wird, falls `<insert-after-ejb-post-create>` den Wert „false“ hat, der Insert über den `CMPPersistenceManager` getätigt. Wieder im `EntityCreationInterceptor` angelangt, werden über den *`invoke`* - Zweig die Interzeptoren unterhalb des `EntityCreationInterceptor` nochmals durchlaufen, bis hin zum CMP-Proxy des OrderBean (jetzt schon mit einer Identität versehen), um dort *`ejbPostCreate`* aufzurufen. Danach wird, falls `<insert-after-ejb-post-create>` den Wert „false2 hat, ein Update über den `CMPPersistenceManager` durchgeführt, andernfalls findet jetzt erst der Insert statt. Die Möglichkeit `<insert-after-ejb-post-create>` auf den Wert „true“ zu setzen, kann benutzt werden um die Performanz zu steigern, da anstatt zweier Datenbank-Aufrufen nur ein Aufruf durchgeführt wird.

Im `EntityLockInterceptor` findet das Setzen von Sperren (Locking) für die EntityBeans statt. Für den Aufruf über *`invokeHome`* findet kein Locking statt, da hier noch keine Identität mit einem Bean assoziiert wurde (Bean ist im Status *pooled*). Für den Aufruf über *`invoke`* findet hier die konfigurierte *Locking Policy* Anwendung. Es kann gewählt werden u.a. zwischen `NoLock`, `QueuedPessimisticEJBLock` und JDBCOptimisticLock. EntityBeans dürfen per Spezifikation nur von einem Thread gleichzeitig aufgerufen werden. Der `EntityLockInterceptor` hat deshalb auch die Aufgabe die Synchronisation von konkurrierenden Threads auf die Ressource EntityBean durchzuführen. `NoLock` kann folglich nur benutzt werden, wenn für jede Transaktion eine eigene Bean-Instanz benutzt wird. Hierfür existiert der EntityMultiInstanceInterceptor. Das Synchronisationsverfahren `QueuedPessimisticEJBLock` ist der Standard in JBoss und ist eine Variante der pessimistischen Sperrverfahren. Das `JDBCOptimisticLock` ist eine Variante der optimistischen Synchronisationsverfahren. Hierbei wird die Synchronisation über JDBC mit Hilfe der Datenbank sichergestellt. Es gibt dabei gemäß [JBoss04] folgende Strategien:

- *Fixed Group of TableField Strategy*: Eine festgelegte Gruppe von Tabellenfeldern werden am Anfang einer Transaktion gesperrt.

- *Modified Strategy*: Die modifizierten Felder bilden die Basis für die Entscheidung, ob eine nicht korrekt synchronisierte Sequenz besteht.
- *Read Strategy*: Alle gelesenen Felder werden hier für die Entscheidung, ob eine nicht korrekt synchronisierte Sequenz vorliegt, herangezogen.
- *Version-Column Strategy*: In die Tabelle der Entität wird ein zusätzliches Feld eingefügt, welches bei jedem Zugriff inkrementiert wird. Weicht das zu schreibende Feld von dem Inhalt auf der Datenbank ab, so wurde es von einer anderen Transaktion geändert. Es liegt somit möglicherweise eine nicht korrekt synchronisierte Sequenz vor.
- *Timestamp-Column Strategy*: Dies ist eine Implementierung des Zeitstempel-Verfahrens aus Kapitel 2.3.3.

Bisher ist noch keine Bean-Instanz mit dem Aufruf assoziiert. Dies wird im `EntityInstanceInterceptor` nachgeholt. Der `EntityInstanceInterceptor` propagiert den *Principal* an ein `Context`-Objekt und holt sich den richtigen Instanzen-Pool für den aufzurufenden Bean-Typ (OrderDataBean). Über das `Context`-Objekt erhält das `LocalEJBInvocation`-Objekt Zugriff auf den Bean-Pool, der vom Typ `org.jboss.ejb.plugins.EntityInstancePool` ist. Da EntityBeans mit einer Identität versehen werden, lohnt sich hier die Einrichtung eines zusätzlichen Caches zum Pool. Im Pool liegen EntityBeans ohne Identität, im Cache sind ausschließlich EntityBeans abgelegt, welche mit einer Identität versehen sind und für eine Wiederverwendung aufbewahrt werden (vgl. Kapitel 3.3.4). JBoss verfügt über eine sehr umfangreiche Möglichkeit, den Cache zu konfigurieren (vgl. Anhang A, Listing A.3), die jedoch nicht im Rahmen dieser Arbeit behandelt wird.

Der `EntityReentranceInterceptor` hat für *`invokeHome`* keine Funktionalität, sondern delegiert einfach zum nächsten Interzeptor. Für *`invoke`* stellt er sicher, dass eine Entität nicht in derselben Transaktion zweimal von unterschiedlichen Aufrufern berührt wird. Dies ist nur erlaubt wenn die Bean im *Deployment Descriptor* explizit als *reentrant* deklariert wurde. Ausgenommen sind Methoden wie *`getEJBHandle`*, *`getEJBHome`*, getPrimaryKey. Da ein Bean, welches *reentrant* deklariert ist, die Synchronisation von Threads und Beans unterminiert (es ist schwierig ein gültiges *Reentrant*-Szenario von einem illegalen Aufruf eines parallelen Threads zu unterscheiden) ist es von Vorteil, wenn so wenig Beans *reentrant* deklariert werden wie möglich.

Im `EntitySynchronizationInterceptor` wird sichergestellt, dass die Werte der EntityBean-Instanz mit der Datenbank synchronisiert werden. Es werden dazu die EntityBean-Methoden *`ejbLoad`* bzw. *`ejbStore`* aufgerufen. Der `EntitySynchronizationInterceptor` ist auch für die Umsetzung der einzelnen Commit-Optionen (Kapitel 3.3.4) zuständig. JBoss unterstützt, neben den in der EJB-Spezifikation geforderten Optionen A, B und C, auch noch eine Option D welche eine Abwandlung von Option A ist, jedoch die Daten nur für eine bestimmte Zeitspanne als gültig betrachtet. Nach Ablauf der Zeitspanne wird der

Zustand der Daten wieder mit der Datenbank synchronisiert. Um die Behandlung der Commit-Optionen durchführen zu können, registriert sich der `EntitySynchronizationInterceptor` als `Synchronization`-Objekt bei der Transaktion. Somit erhält er Nachricht über die Festschreibung einer Transaktion und kann damit die notwendigen Aktionen zu den Commit-Optionen auf dem Bean durchführen.

Im `JDBCRelationInterceptor` wird sich um die Relationen der Entität gekümmert (in diesem Beispiel sind dies die *OrderLine*-EntityBeans, welche zu der *OrderData*-EntityBean aggregiert werden). Die Informationen über die Relationen zwischen den einzelnen Entitäten, sind im *Deployment Descriptor* als *Container Managed Relation (CMR)* abgelegt.

Nachdem alle konfigurierten Interzeptoren durchlaufen wurden, wird als letztes wieder der EntityContainer über einen am Beginn registrierten `ContainerInterceptor` aufgerufen. Hier wird nun aus dem `LocalEJBInvocation`-Objekt das Reflection-Objekt der aufgerufenen Methode von OrderDataLocalHome geholt (hier ist dies die Methode *`create`*). Es wird die dazu passenden Methode des `CMPPersistenceManagers` ermittelt. Für den *`invokeHome`* - Zweig ist es *`createLocalHome`*, für den *`invoke`* - Zweig ist es postLocalHome. Der `CMPPersistenceManager` delegiert den Aufruf an den `EntityPersistenceStore` weiter und dieser an den `JDBCPersistenceStore`, welcher letztendlich das `PreparedStatement` erzeugt und die Entität auf der Datenbank anlegt. Innerhalb dieser Methoden werden auf der OrderDataBean für den *`invokeHome`* - Zweig die *`ejbCreate`* - Methode oder für den *`invoke`* - Zweig die *`ejbPostCreate`* - Methode aufgerufen. Der `CMPPersistenceManager`, zusammen mit dem `EntityPersistenceStore` und dem `JDBCPersistenceStore`, wurden im Kapitel 3.5 als Komponente PersistenceManager bezeichnet. Der Umgang mit dem `ConnectionManager`, sowie der `ManagedConnectionFactory`, `ManagedConnection` und `XAResource`, entspricht in JBoss dabei dem im Kapitel 3.5 erläuterten Funktionsprinzip. Ebenso folgt das anschließende Festschreiben der Transaktion beim Verlassen der OrderManagerSessionBean dem erläuterten Verfahren im Kapitel 3.5.4 bzw. 3.5.5.

4.2 IBM Websphere

4.2.1 Architektur und Überblick

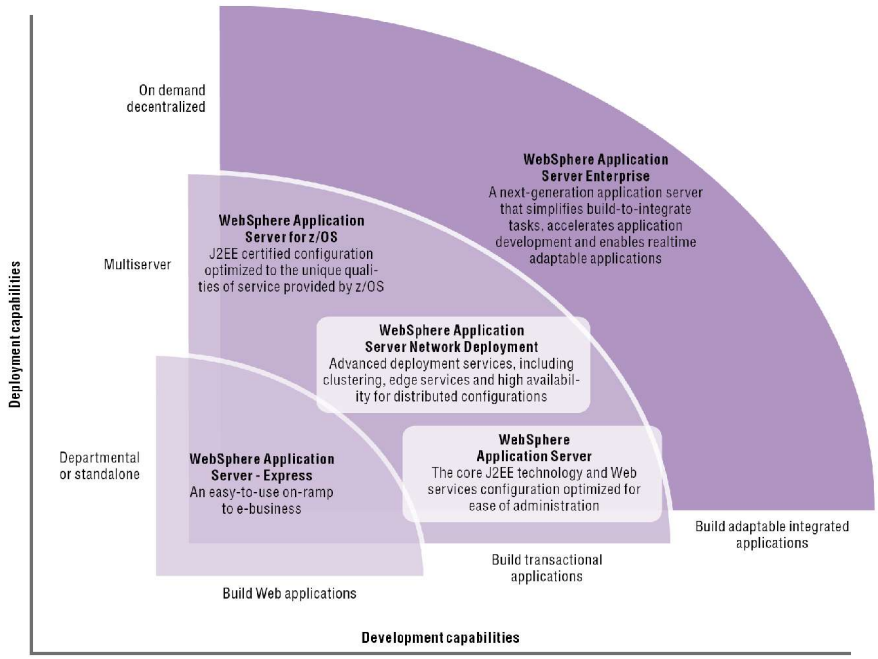

Abbildung 14 Varianten des Websphere Applicationserver (Quelle [WS04])

Der Applicationserver Websphere von IBM ist eine kommerzielle Implementierung der J2EE-Spezifikation. Websphere existiert in drei Varianten (Abbildung 14):

- **Websphere Application Server – Express:** adressiert hauptsächlich das Segment der dynamischen Webanwendungen.
- **Websphere Application Server (Multiserver und z/OS):** adressiert das Segment der Transaktionsanwendungen.
- **Websphere Application Server Enterprise:** adressiert das Segment der *Enterprise Application Integration (EAI)*, also der Integration von sämtlichen Unternehmensprozessen, Anwendungen und Backendsystemen auf eine Plattform.

Der Websphere Applicationserver beinhaltet folgende Funktionalitäten:

- Er unterstützt vollständig die J2EE 1.3 Spezifikation (in Einzelfällen auch schon J2EE 1.4).
- Er unterstützt Webservices und Webservice-Gateways.
- Er unterstützt Failover und Clustering, sowie adaptive Arbeitslastverteilung (Workload).

- Ein integrierter HTTP-Server beinhaltet Plugins für die Integration von externen HTTP-Servern.
- Er stellt eine Infrastruktur für Authentifizierung, Autorisierung und SingleSignOn bereit.
- Er unterstützt ein erweitertes, verteiltes Caching.
- Er bietet Integrationsmöglichkeiten für externe Performanzanalyse-Anwendungen.
- Er unterstützt die Integration mit der Systemmanagementsoftware Tivoli.

Zusätzlich zur J2EE-Spezifikation enthält der Websphere Applicationserver die folgenden Erweiterungen im Bereich des Programmiermodells:

- **Asynchrone Beans:** J2EE-Anwendungen können Funktionalitäten parallelisieren.
- **Startup Beans:** das Ausführen von Funktionalitäten beim Starten und Stoppen einer Anwendung wird ermöglicht .
- **Last Participant Support:** die Durchführung einer verteilten Transaktion erfordert eine Datenquelle, welche den XA-Standard unterstützt. In manchen Fällen ist es jedoch so, dass nicht alle an einer verteilten Transaktion beteiligten Ressourcen das XA-Protokoll unterstützen (z.B. ältere Legacy-Systeme, welche nun integriert werden sollen). Für diesen Fall kann trotzdem eine verteilte Transaktion ordnungsgemäß durchgeführt werden, solange es sich nur um eine solche „Nicht-XA-Konforme“ Ressource handelt und diese als letzte Ressource in einer verteilten Transaktion angesprochen wird.
- **Work areas:** bieten Funktionalität, um Information zwischen verteilten Anwendungen effizient auszutauschen.
- **Activity Session Service:** bietet die Möglichkeit mehrere lokale Transaktionen zu gruppieren.
- **Dynamic query service:** erlaubt es, SQL Abfragen zur Laufzeit zu erstellen und auszuführen.
- **Object Pools:** erlaubt das Caching für jegliche Form von Java-Objekten.
- **Container Managed Messaging:** automatisierte Unterstützung für Ausgangs- und Eingangsnachrichten.
- **Container Managed Persistence over anything:** erweitert das CMP-Modell auf nicht relationale Datenquellen. Jedes System, welches auf Basis eines eindeutigen Schlüssels die Funktionalität zum erzeugen, finden, aktualisieren und löschen von Daten hat, kann als CMP-Datenquelle angebunden werden.

4.2.2 Transaktionsmanagment in Websphere

4.2.2.1 Transaktionsmanager in Websphere

Der IBM Websphere Applicationserver basiert auf der CORBA-Implementierung von IBM. Die Kommunikation zwischen JVM-Prozessen geschieht über das IIOP-Protokoll. Auf Basis der CORBA-Infrastruktur wurde von der OMG der *Object Transaction Service (OTS)* spezifiziert [OTS03], das Java-Mapping für den OTS ist der *Java Transaction Service (JTS)*. Die JTS-Implementierung in Websphere wird dazu benutzt, CORBA-Ressourcen an Websphere-Transaktionen teilnehmen zu lassen. Der Transaktionsmanager, welcher in Websphere benutzt wird, basiert jedoch nicht auf CORBA sondern ist eine Java-Implementierung.

Standardmäßig sind in Websphere zwei Transaktionsmanager-Implementierungen verfügbar. Eine Implementierung unterstützt das Recovery von Transaktionen und transaktionalen Ressourcen, die andere nicht. Welcher Transaktionsmanager in der Server-Instanz benutzt wird, kann über die Konfigurationsdatei `implfactory.properties` angegeben werden. So wäre es denkbar, einen externen Transaktionsmanager in Websphere einzuhängen. Für Websphere genügt es jedoch nicht, nur die JTA-Schnittstellen zu implementieren wie bei JBoss. Websphere-Transaktionsmanager müssen die Schnittstelle `com.ibm.ws.Transaction.WebSphereTransactionManager` implementieren, welche `javax.transaction.TransactionManager` erweitert.

4.2.2.2 SessionBean als transaktionale Ressource

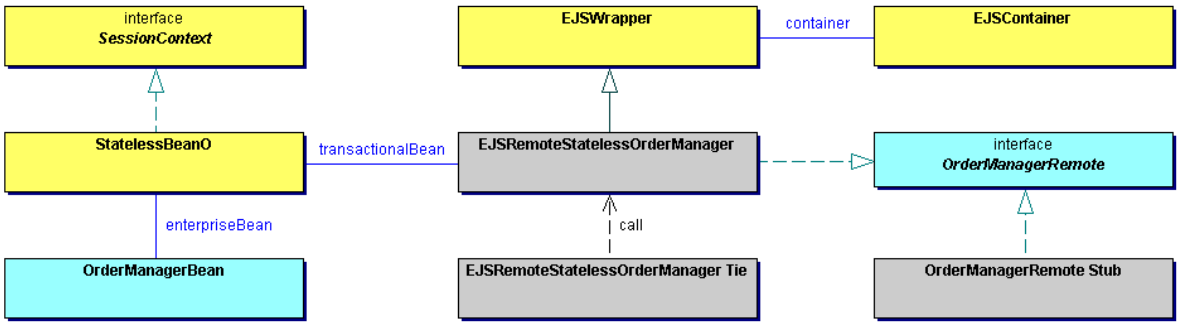

Abbildung 15 Integration von SessionBeans in Websphere

Anders als JBoss minimiert Websphere den dynamischen Aspekt und den damit verbundenen Einsatz von *Dynamic Proxies* oder der Reflection API selbst. Bei der Erzeugung des *Deployment Codes* werden aus der SessionBean und den Informationen des *Deployment*

Descriptors zusätzliche Klassen generiert, welche die Integration einer SessionBean in die Websphere-Laufzeitumgebung ermöglichen. Die in Abbildung 15 blau eingefärbten Klassen bzw. Schnittstellen `OrderManagerBean` und `OrderManagerRemote`, sind vom Entwickler der J2EE-Anwendung oder seiner Entwicklungsumgebung bereitgestellt worden. Websphere generiert (grau eingefärbt) in einem ersten Schritt die Klasse `EJSRemote-StatelessOrderManager`, welche `EJSWrapper` erweitert und das EJBRemote-Interface `OrderManagerRemote` implementiert. Websphere benutzt das Wrapper-Objekt als Klammer zwischen dem `EJSContainer`, welcher den Transaktionsmanager ansteuert und dem eigentlichen EnterpriseBean, dem `OrderManagerBean` bzw. seiner transaktionalen Hülle, dem StatelessBeanO. Das `StatelessBeanO` implementiert das `SessionContext`-Interface und stellt darüber dem assoziierten `OrderManagerBean` den `SessionContext` bereit.

In einem zweiten Schritt generiert der RMI-Compiler die für die Remote-Kommunikation notwendigen Stubs und Ties. Der `OrderManagerRemote_Stub` implementiert `OrderManagerRemote` und wird von der CORBA-Infrastruktur an den Client als Kommunikationsproxy geliefert. Aufrufe werden in CORBA-Nachrichten verpackt und an `EJS-RemoteStatelessOrderManager_Tie` weitergeleitet, welches den Aufruf entpackt und an eine Instanz der Klasse `EJSRemoteStatelessOrderManager` zur Abarbeitung delegiert.

4.2.2.3 EntityBean als transaktionale Ressource

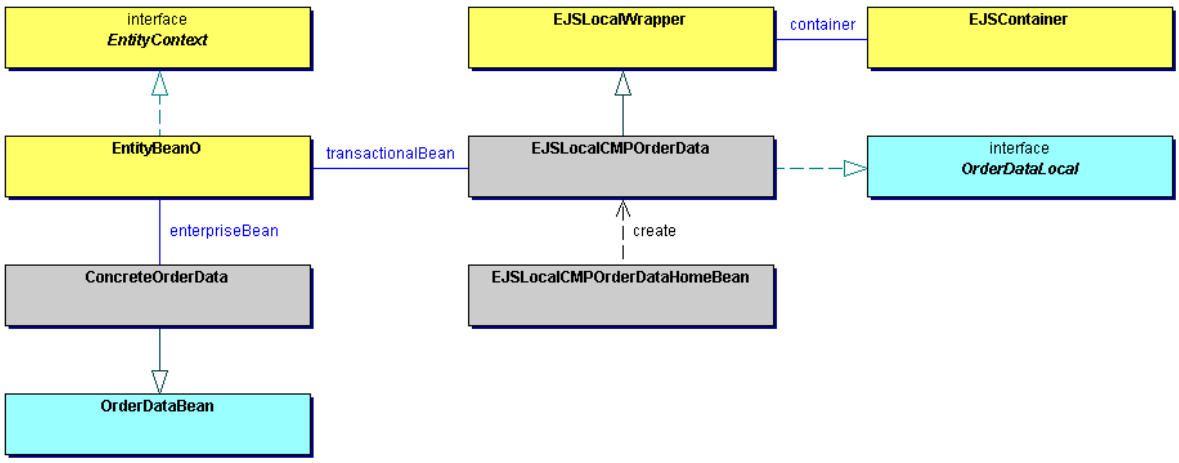

Abbildung 16 Integration von EntityBeans in Websphere

Die Integration von EntityBeans stellt sich in Abbildung 16 bis auf leichte Namensgebungsunterschiede gleich dar. Da in der betrachteten J2EE-Anwendung nur EntityBeans mit einer `EJBLocalObject`-Schnittstelle benutzt werden, erweitert die von Websphere generierte Klasse `EJSLocalCMPOrderData` die Container-Klasse EJSLocalWrapper. Das transaktionale Bean, welches mit `EJSLocalCMPOrderData` assoziiert ist, ist nun vom Typ `EntityBeanO` und implementiert die Schnittstelle `EntityContext`. Das `EntityBeanO` stellt dem EnterpriseBean `OrderDataBean` den `EJBContext` bereit.

Der einzige prinzipielle Unterschied stellt die Klasse `ConcreteOrderData` dar, welche von Websphere erzeugt wird als Erweiterung von `OrderDataBean`. Diese Klasse stellt die Integration in den PersistenceManager dar, welcher in Websphere nach dem Konzept eines *Resource Adapter* der JCA ausgelegt ist. Der gesamte Zugriff, einschließlich Schnittstellen und der JDBC-Statement-Ebene, werden von Websphere beim *Deployment* generiert. Die Anzahl der Klassen ist dementsprechend hoch und ist auf die wichtigsten Klassen beschränkt zusammen mit den relevanten Container-Klassen in Abbildung 17 dargestellt. Die grau eingefärbten Klassen sind wieder die von Websphere generierten Klassen. Die gelb eingefärbten sind die Infrastruktur-Klassen mit allgemeinem Zweck von Websphere selbst. Die orange eingefärbten Klassen sind ebenfalls Bestandteil der Infrastruktur, jedoch haben sie mit der Transaktionsverarbeitung im speziellen zu tun. Die grün eingefärbten Klassen haben insbesondere mit der Persistierung in einer Datasource (hier relationale Datenbank über JDBC) zu tun.

Die Klasse `ConcreteOrderData` stellt die Implementierung für die abstrakten Zugriffsmethoden auf die Klassenattribute der EntityBean bereit. Die Zugriffsmethoden werden dabei an die entsprechenden Werte der aggregierten `OrderDataCMPCacheEntryImpl` delegiert. Die Werte in `OrderDataCMPCacheEntryImpl` werden über `ConcreteOrderData` von dem `OrderManagerBean` befüllt. Bei der Erzeugung des Datensatzes auf der Datenbank werden die Daten mit Hilfe des `OrderDataCMPInjectorImpl` extrahiert und gemäß JCA in einem `Record`-Objekt abgelegt, und über eine `Interaction`-Instanz (`WSInteractionImpl`) ausgeführt. In der Klasse `OrderDataCMPFunctionSet` ist der Zugriff auf den relationalen *Resource Adapter* `WSRelationalRAAdapter` realisiert, sowie auf `WSRdbDatasource` und `WSJdbcConnection`, welche letztendlich die Verbindung zum herstellerspezifischen JDBC-Treiber der Datasource halten. Die `WSRdbManagedConnection` und die `WSRdbXAResourceImpl` sind Implementierungen der korrespondierenden Schnittstellen der JCA und JTA. Beide Klassen werden innerhalb von Websphere unter anderem zur Assoziierung von Datenbankressourcen, EnterpriseBean-Daten und dem Transaktionskontext benötigt und damit letztendlich zur Durchführung von Commits und Rollbacks auf dem *Resourcemanager* der Datenquelle.

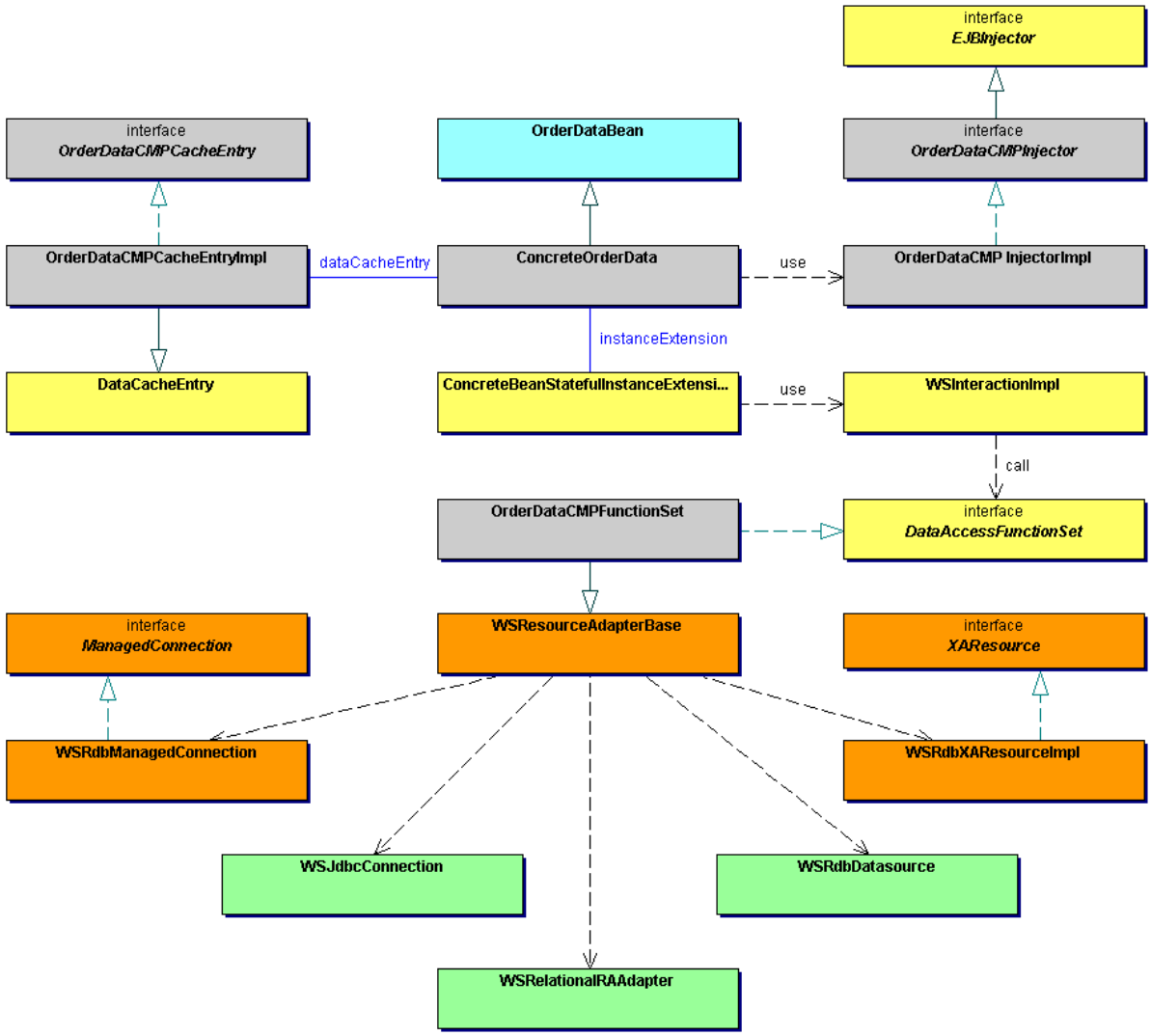

Abbildung 17 Integration von EntityBeans in den PersistenceManager

4.2.3 Integration von Websphere

Im Normalfall bedarf es recht wenig um auf einen Applicationserver von einer externen Java-Anwendung oder einem anderen Applicationserver-Produkt zuzugreifen. Der erste Schritt ist für die Erzeugung des `InitialContext`, der den Zugriff auf den Namensdienst des Applicationservers über JNDI bereitstellt, die entsprechenden Umgebungswerte zu setzen:

```
Hashtable environment = new Hashtable();
environment.put("java.naming.factory.initial","org.jnp.interfaces.NamingContextFactory");
environment.put("java.naming.factory.url.pkgs","org.jboss.naming:org.jnp.interfaces");
environment.put("java.naming.provider.url","localhost");
InitialContext initialContext = new InitialContext(environment);
```

Der zweite Schritt ist, dem Applicationserver A1 (z.B. JBoss) die notwendigen Klassen für den Zugriff auf den Applicationserver A2 (z.B. Bea Weblogic) zur Verfügung zu stellen. Viele Applicationserver-Hersteller haben dafür ein spezielles Client-JAR, welches alle relevanten Klassen und Schnittstellen beinhaltet.

Der dritte und letzte Schritt ist, ein weiteres Client-JAR für das auf Applicationserver A2 installierte EAR zu erzeugen, welches alle Schnittstellen-Klassen, Hilfsklassen und Übergabeparameter sowie Stub-Klassen des EARs beinhaltet. Dieses Client-JAR muss ebenfalls dem Applicationserver A1 zur Verfügung gestellt werden. Damit kann der Applicationserver A1 oder auch eine externe Java-Anwendung auf die Anwendung in Applicationserver A2 zugreifen.

Mit Websphere gestaltet sich insbesondere der erste und zweite Schritt etwas problematisch, da weder ein explizites Client-JAR für Websphere zu finden ist noch die Standard-Umgebungswerte für den `InitialContext` ausreichen. Durch konsequentes Testen konnten die fehlenden Werte sowie die notwendigen JARs ermittelt werden. Websphere benötigt für den ersten Schritt die folgenden Umgebungswerte für die Initialisierung des `InitialContext`-Objektes:

```
environment.put("java.naming.provider.url","corbaloc:iiop:localhost");
environment.put("java.naming.factory.initial",
                    "com.ibm.websphere.naming.WsnInitialContextFactory");
environment.put("java.naming.factory.url.pkgs",
                    "com.ibm.ws.client.applicationclient:com.ibm.ws.naming");

environment.put("com.ibm.websphere.naming.hostname.normalizer",
                    "com.ibm.ws.naming.util.DefaultHostnameNormalizer");
environment.put("com.ibm.websphere.naming.name.syntax","jndi");
environment.put("com.ibm.websphere.naming.namespace.connection","lazy");
environment.put("com.ibm.websphere.naming.jndicache.cacheobject","populated");
environment.put("com.ibm.websphere.naming.namespaceroot","defaultroot");
environment.put("com.ibm.ws.naming.wsn.factory.initial",
                    "com.ibm.ws.naming.util.WsnInitCtxFactory");
environment.put("com.ibm.websphere.naming.jndicache.maxcachelife","0");
environment.put("com.ibm.websphere.naming.jndicache.maxentrylife","0");
environment.put("com.ibm.websphere.naming.jndicache.cachename","providerURL");
```

Für den zweiten Schritt benötigt Websphere die JARs: `bootstrap.jar`, `idl.jar`, `ecutils.jar`, `ffdc.jar`, `iwsorb.jar`, `j2ee.jar`, `lmproxy.jar`, `ras.jar`, `marshall.jar`, `naming.jar`, `namingclient.jar`, `properites.jar`, `ibmorb.jar`, `txClientPrivate.jar`, `utils.jar`, `wsexception.jar`, `iwsorbutil.jar` und `core.jar`. Zusätzlich werden noch die DLLs aus dem `bin` -Verzeichnis des IBM-JDKs benötigt. Der einfachste Weg ist es als JVM das IBM-JDK zu verwenden,

dann sind die DLLs automatisch im Zugriff. Jedoch können die DLLs auch bei Benutzung eines Sun-JDKs über die Umgebungsvariable `java.library.path` beim Aufruf des Java-Interpreters übergeben werden.

4.2.4 Zusammenspiel der Komponenten

Im folgenden Kapitel wird das Zusammenspiel der vorgestellten Websphere-Komponenten anhand des schon in den vorherigen Kapiteln verwendeten Auftragsanlage-Szenarios (*NewOrder*-Transaktion) vorgestellt. Bei der Beschreibung, ebenso wie bei der Darstellung in den Abbildungen, wurden nur die Klassen berücksichtigt, welche zur Verarbeitung bzw. Weiterleitung des Aufrufs dienen oder originär mit der Transaktionsverarbeitung oder Persistenzanbindung zu tun haben. Aufgrund der großen Anzahl von Abstraktionsstufen und Hilfsklassen in Websphere wurden sinnvolle Gruppierungen zum Zwecke der besseren Übersichtlichkeit gebildet.

4.2.4.1 Client erzeugt SessionBean

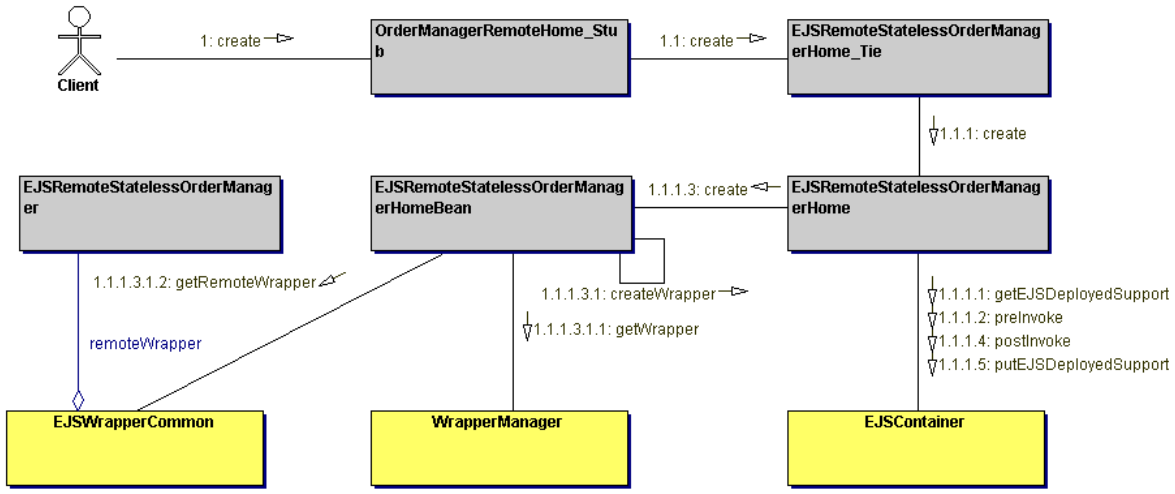

Abbildung 18 Client erzeugt SessionBean

Der Client initialisiert den `InitialContext` wie in Kapitel 4.2.3 erläutert. Nachdem über den Aufruf von *`narrow`* von `PortableRemoteObject` der CORBA-Stub des `OrderManagerRemoteHome` bereitgestellt ist, kann der Client darauf die Methode *`create`* aufrufen (Abbildung 18). Der Aufruf von *`create`* wird vom `OrderManager-RemoteHome_Stub` in eine CORBA-Nachricht verpackt (*marshalling*) und über IIOP an

den Server-ORB von Websphere übertragen. Dort wird der ankommende Aufruf an einen ORB-Worker-Thread zur Verarbeitung weitergeleitet. Eine Instanz des `EJSRemoteStatelessOrderManagerHome_Tie` entpackt die CORBA-Nachricht (*unmarshalling*) und ruft die Methode *`create`* auf einer Instanz der Klasse `EJSRemoteStatelessOrderManagerHome` auf. Diese Klasse erweitert, wie auch ihre korrespondierende Bean `EJSRemoteStatelessOrderManager`, die Websphere-Infrastruktur-Klasse `EJSWrapper` und stellen damit das Integrationsobjekt zwischen Anwendungsklassen und Container-Infrastruktur dar. Zunächst wird über *`getEJSDeployedSupport`* eine Instanz von `EJSDeployedSupport` geholt, welche ein Aufbewahrungsort darstellt für alle Objekt-Referenzen welche während der Verarbeitung innerhalb des Containers durch die einzelnen Abstraktionsschichten geschleust werden müssen. Dieses Objekt wird im folgenden auch bei fast jedem Methodenaufruf übergeben. Danach wird *`preInvoke`* auf dem `EJSContainer` ausgeführt.

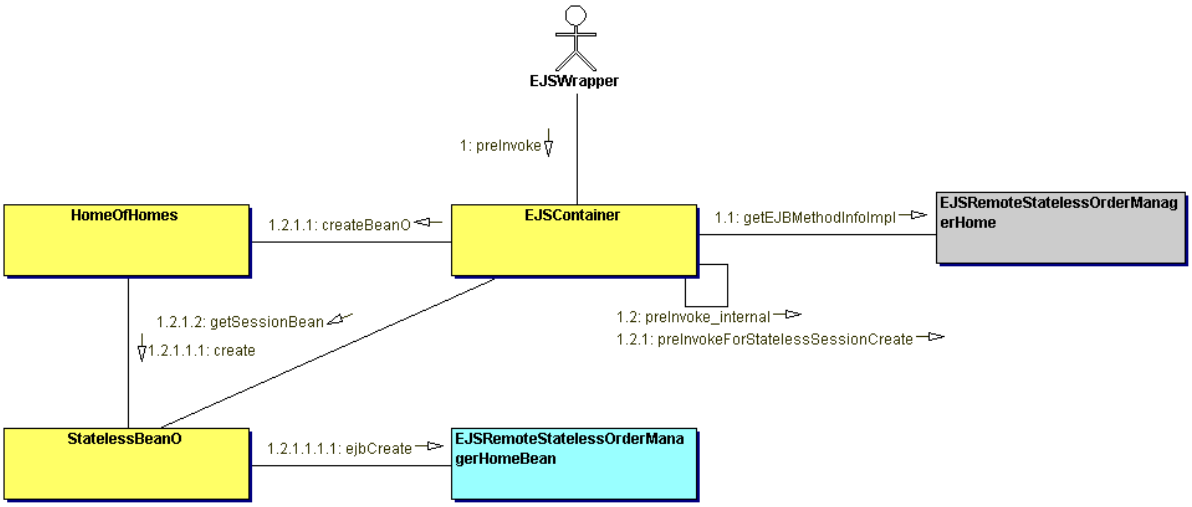

Abbildung 19 EJSContainer vor dem Erzeugen einer SessionBean

Die `EJSContainer`-Instanz ist für die Koordination einer gesamten Anwendung (entspricht einem EAR) zuständig. Der Aufruf von *`preInvoke`* auf dem `EJSContainer` durch den `EJSWrapper` (Abbildung 19) führt dazu, dass über *`getEJBMethodInfoImpl`* die Informationen des *Deployment Descriptors* zu dieser Methode geholt werden. Danach werden diese Informationen zusammen mit dem übergebenen `EJSRemoteStatelessOrderManagerHome` der Methode *`preInvokeForStatelessSessionCreate`* übergeben. Der `EJSContainer` benutzt diese Informationen, um über das *Factory*-Objekt ([Gam96]) `HomeOfHomes` mittels *`createBeanO`* eine Instanz der Klasse `StatelessBeanO` zu erzeugen. Bei der Erzeugung der `StatelessBeanO` durch das `HomeOfHomes` wird auch eine Instanz der `EJSRemoteStatelessOrderManagerHomeBean` erzeugt, welche das SessionBean-Interface implementiert und *`ejbCreate`* auf ihr aufgerufen. An-

schließend holt der EJSContainer über *getSessionBean* das EJSRemote-StatelessOrderManagerHomeBean aus der StatelessBeanO-Instanz und gibt es als Methodenergebnis dem EJSWrapper zurück.

In EJSRemoteStatelessOrderManagerHomeBean wird auf diesem Bean-Objekt die Methode *create* ausgeführt, welche intern an die Methode *createWrapper* weitergeleitet wird (Abbildung 18). In *createWrapper* wird über *getWrapper* des WrapperManager das zu diesem Bean korrespondierende EJSWrapperCommon-Objekt geholt. Das EJSWrapperCommon-Objekt beinhaltet alle Klassennamen für die einzelnen Implementierungen von EJBHome und EJBObject. Da es sich in diesem Fall um eine OrderManagerRemote-Schnittstelle handelt, wird auf dem EJSWrapperCommon-Objekt *getRemoteWrapper* aufgerufen. Als Ergebnis wird eine Instanz der Klasse EJS-RemoteStatelessOrderManager zurückgeliefert, welche über die CORBA-Infrastruktur wiederum dem Client zur Verfügung gestellt werden kann. Vorher wird allerdings noch von EJSRemoteStatelessOrderManagerHome die Methode *postInvoke* auf dem EJSContainer aufgerufen, welche im Fall des Erzeugens eines SessionBeans keine wichtigen Funktionalitäten mehr beinhaltet. Anschließend wird noch das EJSDeployed-Support-Objekt über *putEJSDeployedSupport* wieder an den EJSContainer abgegeben.

Der Client kann nun auf dem über die *create*-Methode des OrderManagerRemote-Home zurückgelieferte Instanz des OrderManagerRemote (genauer des Order-ManagerRemote_Stub) eine Methode aufrufen.

4.2.4.2 Client ruft SessionBean

Wie in Abbildung 20 zu sehen ist arbeitet dieses Szenario bis zur Delegation des Methodenaufrufs an das EJSRemoteStatelessOrderManager-Objekt deckungsgleich zum Erzeugen des SessionBeans aus dem vorherigen Kapitel, abgesehen davon, dass diesmal Stub und Tie vom konkreten EJBObject und nicht vom EJBHome sind. Das EJSRemote-StatelessOrderManager-Objekt ruft wieder die schon im vorherigen Kapitel besprochenen Methoden auf. Von der Methode *preInvoke* erhält das Objekt diesmal jedoch eine Instanz von OrderManagerBean zurück, auf der die Methode aufgerufen wird. Innerhalb der *preInvoke* - Methode des EJSContainers geschieht jedoch für den Aufruf eines SessionBeans sehr viel mehr als im Falle des Erzeugens eines SessionBeans. Dieser Sachverhalt wird in Abbildung 21 dargestellt.

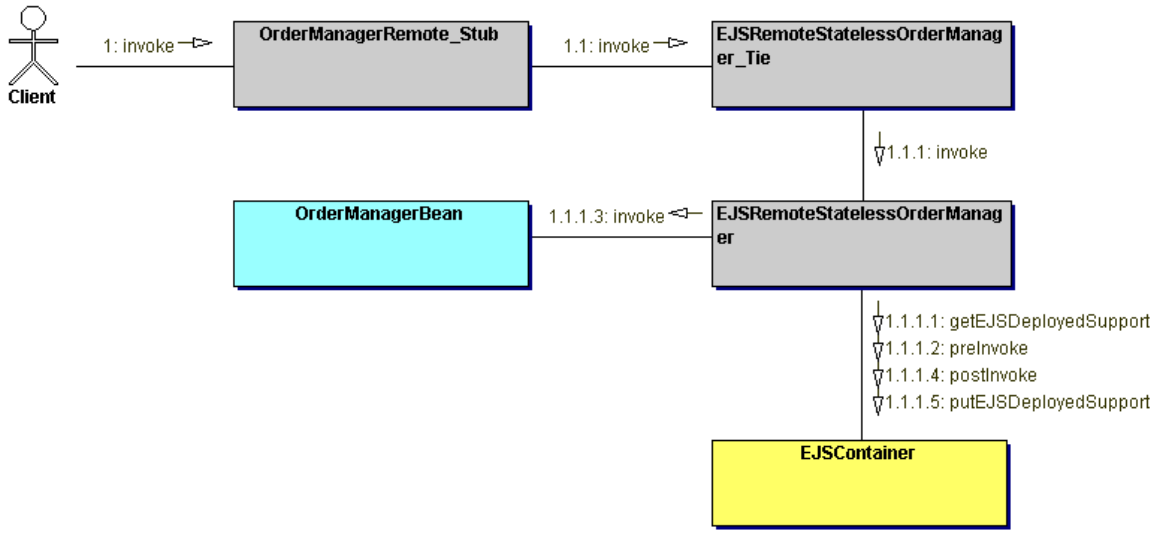

Abbildung 20 Client ruf SessionBean

Zunächst wird über `getEJBMethodInfoImpl` wieder die *Deployment Descriptor*-Informationen zu der aufgerufenen Methode ermittelt und zusammen mit der Instanz des `EJSWrappers` an die Methode `preinvoke_internal` übergeben. Mit dem anschließenden Aufruf von `preInvoke` auf dem `TransactionController` wird bei entsprechendem Transaktionsattribut der Methode eine Transaktion gestartet und mit der weiteren Methodenausführung assoziiert. Dies wird später im Detail betrachtet.

Über die Methode `getCurrentTx` wird eine neue `ContainerTx`-Instanz erzeugt und anschließend über die Methode `enlistWithTransaction` des `TransactionController` mit der aktiven Transaktion assoziiert. Auf dieser neu erzeugten `ContainerTx` wird die Methode `preInvoke` aufgerufen und anschließend über `setIsolationLevel` die Isolationsstufe aus der EJBMethodInfoImpl. Danach erfolgt die Aktivierung des Session-Beans über `activateBean` des `Activator`-Objektes, welcher die `ContainerTx` und die `BeanId` des `EJSWrappers` übergeben wird. Diese Parameter werden an eine Implementierung der `ActivationStrategy` übergeben. Für die `ActivationStrategy` gibt es verschiedene Implementierungen, da je nach konfigurierter Commit-Option der Zustand der Beans zwischen Transaktionen von der Datenbank aktualisiert werden muss. Die `ActivationStrategy` umfasst dementsprechend auch nicht nur die Aktivierung von EJBs und den damit verbundenen Aufruf von `ejbActivate`, sondern auch die Passivierung, die Erzeugung, das Löschen, das Commit und das Rollback.

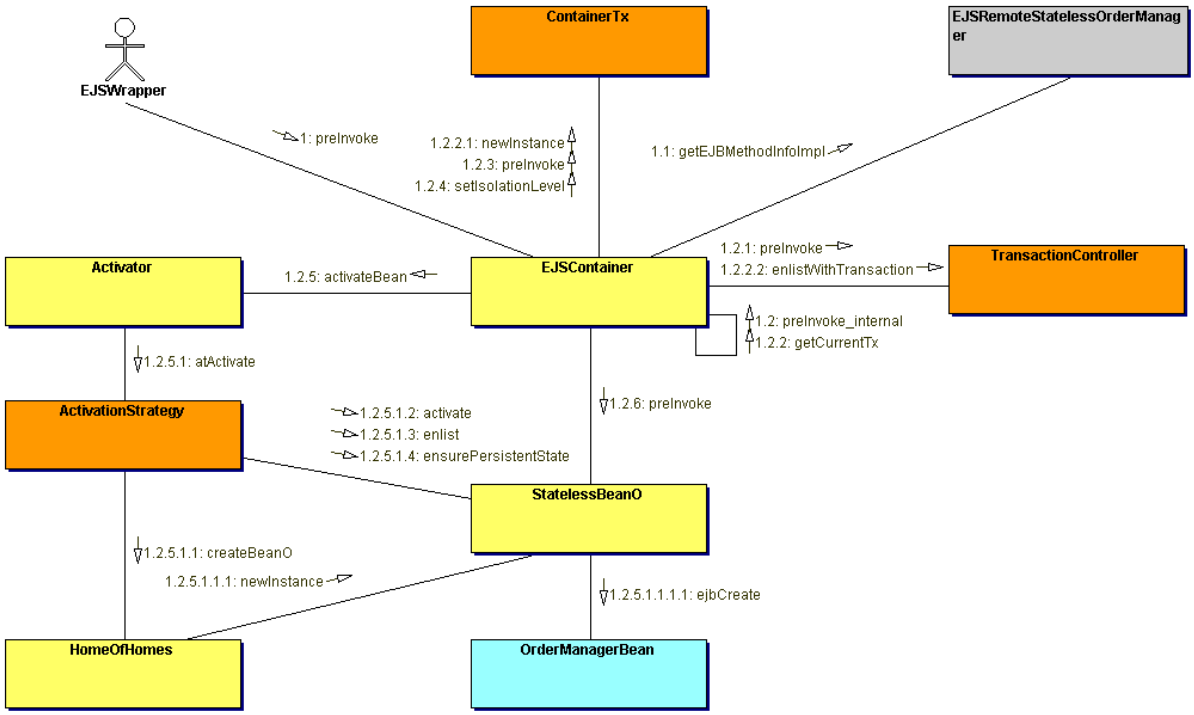

Abbildung 21 EJSContainer vor dem Aufruf einer SessionBean

Die `ActivationStrategy` erzeugt über *`createBeanO`* des `HomeOfHomes` eine neue Instanz von `StatelessBeanO`, welche beim Aufruf des Konstruktors auch eine neue Instanz von `OrderManagerBean` erzeugt. Nachdem sich die `StatelessBeanO` der EJB als `SessionContext` übergeben hat ruft sie *`ejbCreate`* auf dem EJB auf. Anschließend teilt die `ActivationStrategy` dem `StatelessBeanO` über die *`activate`*-Methode mit, in welchem Zustand sich das Bean befindet und dass *`ejbActivate`* auf `OrderManagerBean` aufzurufen ist. Da *Stateless SessionBeans* jedoch nicht aktiviert oder passiviert werden, ist die Methode des `StatelessBeanO` nicht implementiert. Mit *`enlist`* wird das `ContainerTx`-Objekt an die `StatelessBeanO`-Instanz übergeben. Der Aufruf von *`ensurePersistentState`* ist ebenso wie *`ejbActivate`* bei SessionBeans ohne Bedeutung. Abschließend wird auf dem `StatelessBeanO` noch *`preInvoke`* mit `EJSDeployedSupport` und `ContainerTx` aufgerufen, um die richtige Isolationsstufe zu setzen. Als Ergebnisobjekt wird von *`preInvoke`* das `OrderManagerBean` zurückgeliefert, welches bis an den aufrufenden `EJSWrapper` zurückgegeben wird.

Die zuvor nur grob in der Auswirkung dargestellte Funktionsweise des `TransactionControllers` soll an dieser Stelle genauer betrachtet werden (Abbildung 22). Der Aufruf von *`preInvoke`* des `EJSContainers` auf dem `TransactionController` (Abbildung 21, 1.2.1 und Abbildung 22, 1) führt zunächst dazu, dass eine aktive Transaktion über *`suspendLocalTx`* von dem ausführenden Thread deassoziiert und die Bearbeitung ausge-

setzt wird. Anschließend wird *preInvoke* auf einer Implementierung der TransactionStrategy aufgerufen. Wie schon im obigen Fall der ActivationStrategy, verbergen sich hinter den konkreten Implementierungen verschiedene Varianten, was vor dem ausführen einer SessionBean-Methode geschehen soll. Im Falle der TransactionStrategy korrespondieren diese Varianten zu den bekannten EJB-Transaktionsattributen (Kapitel 3.3.2). Da das verwendete SessionBean mit dem Transaktionsattribut *RequiresNew* versehen wurde, bedeutet es, dass eine Instanz der Klasse RequiresNew ausgeführt wird. Der Aufruf von *preInvoke* der RequiresNew-Klasse suspendiert die momentan ausgeführte globale Transaktion über *suspend* am WebsphereTransactionManager, erzeugt mit *begin* eine neue Transaktion und assoziiert sie mit dem aktiven Aufruf. Das Ergebnis der Ausführung, sowie die suspendierte Transaktion, werden in einer neu erzeugten Instanz von TxCookie abgelegt. Das TxCookie wird als Ergebnis an den TransactionController zurückgegeben.

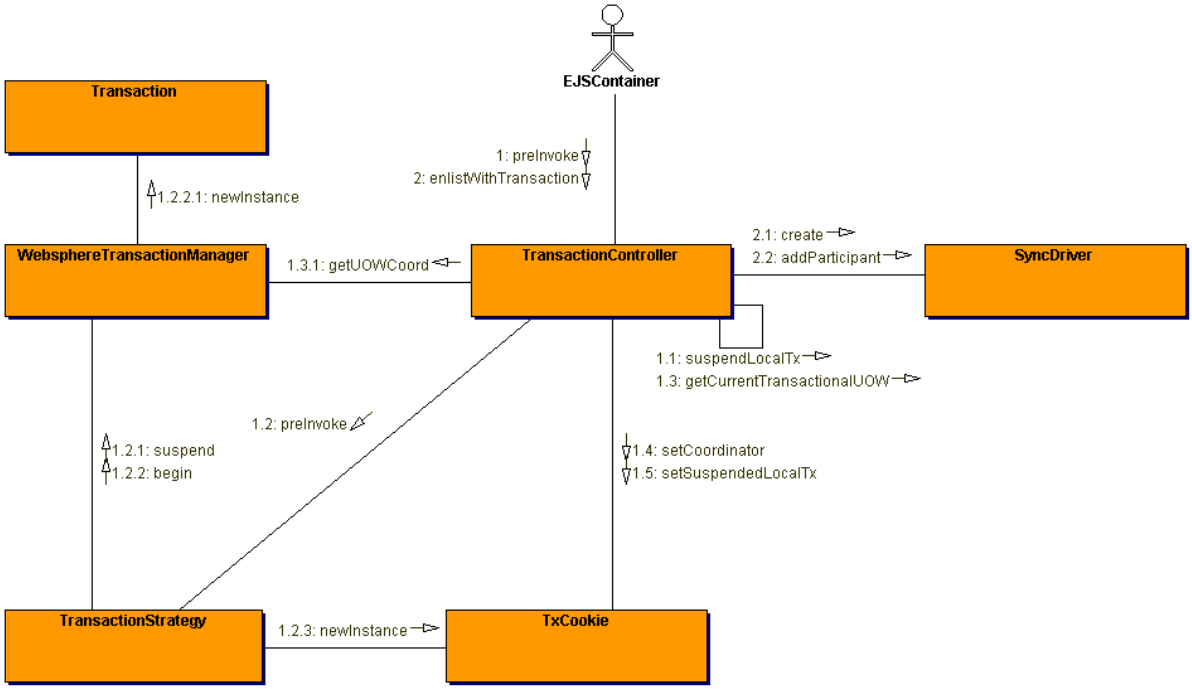

Abbildung 22 TransactionController des EJSContainer vor dem Aufruf einer SessionBean

Der TransactionController holt sich über *getCurrentTransactionalUOW* und *getUOWCoord* des WebsphereTransactionManagers die aktive Transaktion (eine Implementierung der JTA-Transaction-Schnittstelle und der Websphere-Schnittstelle UOWCoordinator), welche sich bei Websphere *Unit of Work Coordinator* nennt. Die Implementierung des Transaction-Interfaces hat dementsprechend noch einige zusätzliche Funktionalitäten, z.B. hält sie den RecoveryManager, die Registrierungstabelle der beteiligten Ressourcen und koordiniert Rollbacks, *One Phase Commits*, sowie *Two Phase Commits*. Das

`Transaction`-Objekt bzw. der `UOWCoordinator` wird auf dem `TxCookie` gesetzt ebenso wie die am Anfang suspendierte lokale Transaktion. Danach wird das `TxCookie` an den aufrufenden `EJSContainer` zurückgegeben.

Der Methodenaufruf „enlistWithTransaction“ (Abbildung 21, 1.2.2.2 und Abbildung 22, 2) ist etwas weniger aufwendig. In dieser Methode wird eine Instanz der Klasse `SyncDriver` erzeugt und die im Methodenaufruf übergebene `ContainerTx`-Instanz, über *`addParti-cipant`* dem `SyncDriver` hinzugefügt. Der `SyncDriver` ist das Bindeglied zwischen dem `UOWCoordinator` (`Transaction`) und den `Synchronization`-Objekten. Über die `Synchronization`-Schnittstelle werden Zustandsänderungen der Transaktion über *`beforeCompletion`* und *`afterCompletion`* zusammen mit dem Ergebnis verteilt.

4.2.4.3 Commit einer Transaktion beim Verlassen der SessionBean

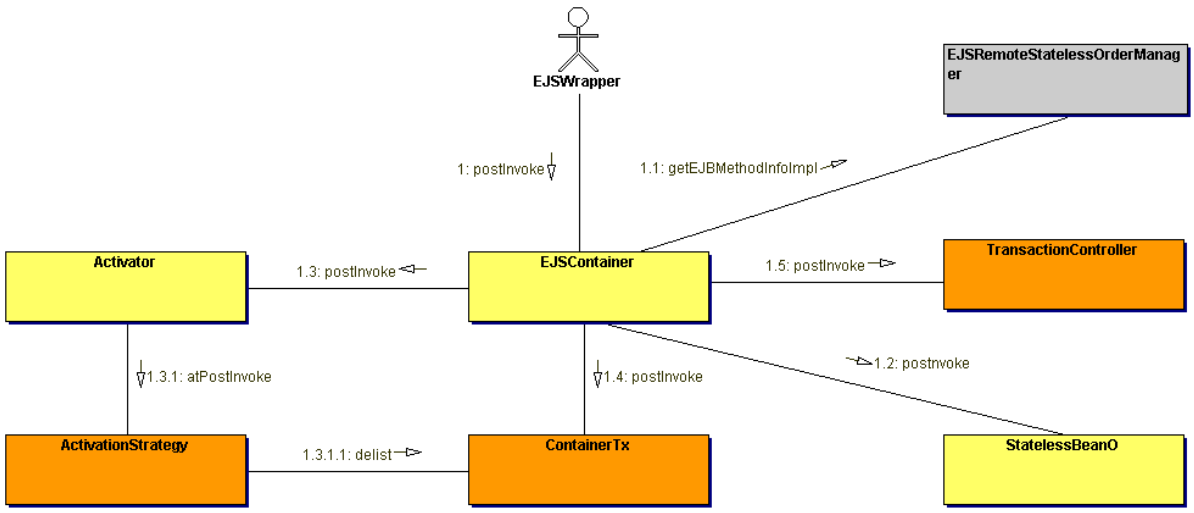

Abbildung 23 EJSContainer nach dem Aufruf einer SessionBean

Nachdem die Methode der SessionBean ausgeführt wurde, wird vom `EJSWrapper` (in dem betrachteten Fall eine Instanz von `EJSRemoteStatelessOrderManager`) die Methode *`postInvoke`* auf dem `EJSContainer` aufgerufen (Abbildung 23). Diese Methode wird sowohl bei erfolgreicher Ausführung des SessionBean-Aufrufs als auch bei einem Fehlerfall ausgeführt. Die Methode *`postInvoke`* handhabt sowohl das Commit als auch ein Rollback der mit dem Aufruf assoziierten Transaktion.

Zunächst wird auf den `StatelessBeanO` die Methode *`postInvoke`* aufgerufen, welches die Bean-Instanz wieder in den Objekt-Pool zurückgibt. Danach wird *`postInvoke`* auf dem `Activator` aufgerufen, welcher den Aufruf an die jeweilige `Activation-Strategy` delegiert. Die `ActivationStrategy` deregistriert die `StatelessBeanO`-

Instanz von `ContainerTx` über *`delist`*. Anschließend ruft der `EJSContainer` auf `ContainerTx`, welche über den `EJSDeployedSupport` geholt wurde, die Methode *`postInvoke`* auf. Hier wird jedoch nur ein Merker-Attribut gesetzt.

Als letztes wird *`postInvoke`* auf dem `TransactionController` ausgeführt, um die aktive Transaktion entsprechend den Informationen des *Deployment Descriptors* zu behandeln (Abbildung 23, 1.5 und Abbildung 24, 1). Dem `TxCookie` welches in der `EJBMethodinfoImpl` abgelegt ist, kommt in *`postInvoke`* des `TransactionControllers` zentrale Bedeutung zu. Hier sind Zustands-Attribute und Datenobjekte aus der vorherigen Ausführung von *`preInvoke`* gespeichert. Darüber hinaus sind auch Informationen über die Ausführung der SessionBean-Methode enthalten (z.B. aufgetretene Ausnahmebedingungen und welche Strategie für die Fehler angewandt werden soll).

Zunächst wird über *`getExceptionType`* ausgewertet, ob bei der Ausführung der SessionBean eine Ausnahmebedingung aufgetreten ist, welche zu einem Rollback der Transaktion führen sollte. Ist dies nicht der Fall wird die in *`preInvoke`* zuvor ausgeführte `TransactionStrategy` aus dem `TxCookie` geholt und auf ihr *`postInvoke`* aufgerufen.

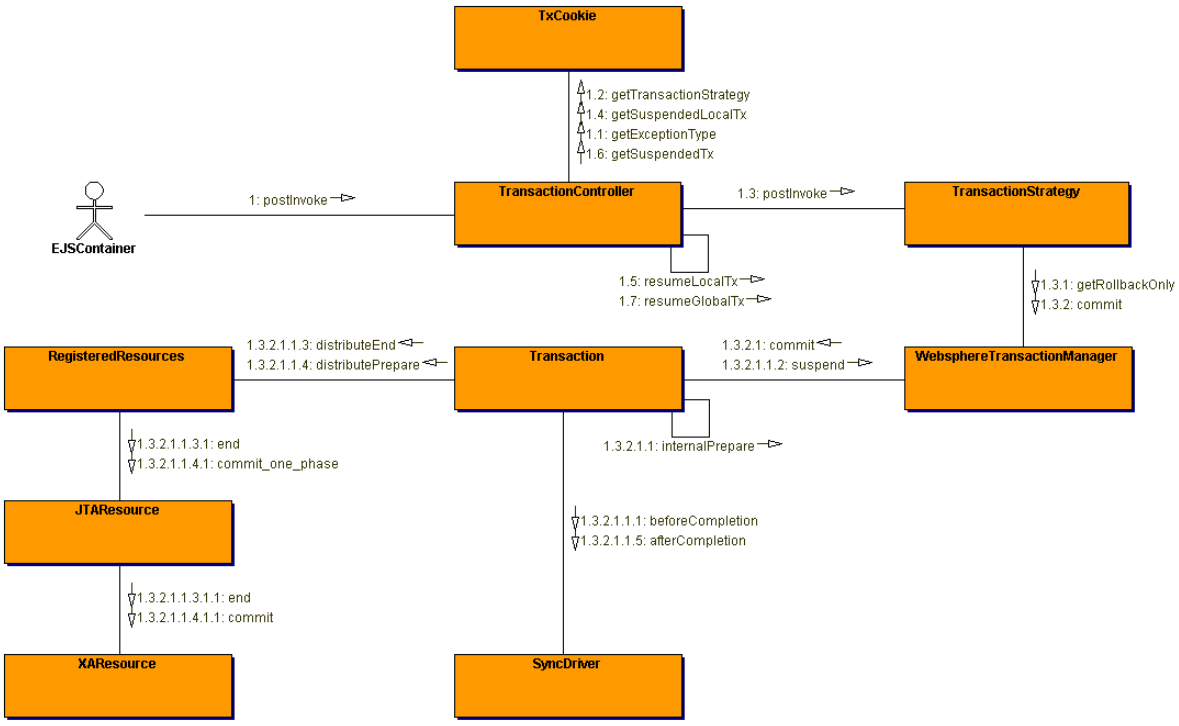

Abbildung 24 TransactionController des EJSContainer nach Aufruf einer SessionBean (One Phase Commit)

Die `TransactionStrategy` (in diesem Fall `RequiresNew`) überprüft zunächst durch Aufruf von *`getRollbackOnly`* auf dem `WebsphereTransactionManager`, ob auf der Transaktion ein Rollback durchgeführt werden muss. Ist dies nicht der Fall wird auf dem `WebsphereTransactionManager` die Methode *`commit`* aufgerufen. Der `Web-`

sphereTransactionManager delegiert den Aufruf an die aktive Transaktion (Transaction bzw. UOWCoordinator). Der UOWCoordinator führt nun ein potentielles *Two Phase Commit* durch, indem er zunächst *internalPrepare* aufruft. Der UOWCoordinator optimiert die Durchführung von Transaktionen mit nur einem beteiligten *Resourcemanager* (*One Phase Commit*). Wird während *internalPrepare* bemerkt, dass alle beteiligten Ressourcen zu demselben *Resourcemanager* gehören, dann wird ein bestimmtes Statusflag zurückgeliefert, welches dazuführt, dass kein *internalCommit* mehr ausgeführt wird (Abbildung 24). Wenn mehrere *Resourcemanager* an einer Transaktion beteiligt sind, muss ein *Two Phase Commit* durchgeführt werden und damit auch ein Aufruf von *internalCommit* des UOWCoordinators (Abbildung 25).

In *internalPrepare* werden zunächst alle registrieren Synchronization-Objekte von dem bevorstehenden Commit durch den Aufruf von *beforeCompletion* unterrichtet. Danach wird durch den Aufruf von *suspend* auf dem WebsphereTransactionManager sichergestellt, dass die Transaktion nicht mehr aktiv ist auf welcher jetzt ein Commit durchgeführt werden soll.

Als nächstes werden die an der Transaktion beteiligten Ressourcen von dem Ende der Transaktion unterrichtet. Dies geschieht über den Aufruf von *distri-buteEnd* auf der Helferklasse RegisteredResources. Die Klasse Registered-Resources ist dafür verantwortlich, die Anweisungen des UOWCoordinators an die beteiligten Ressourcen zu verteilen. Innerhalb von *distributeEnd* wird somit auf jeder beteiligten Ressource die Methode *end* zusammen mit dem entsprechenden Statusflag aufgerufen. Die JTAResource ist ein Hülle um die eigentliche XAResource. Die XAResource stammt aus der WSRdb-ManagedConnection des WSRelationalRA-Adapter (Kapitel 4.2.2.3).

Nachdem alle Ressourcen von dem Ende der Transaktion unterrichtet wurden, führt der UOWCoordinator die Methode *distributePrepare* auf RegisteredResources aus. Im Fall, dass alle Ressourcen zu demselben *Resourcemanager* gehören wird das *Two Phase Commit*-Protokoll dahingehend abgekürzt, dass nur die Methode *comit_one_phase* auf jeder JTAResource ausgeführt wird. Dies hat zur Folge, dass auf der XAResource nicht zunächst ein *prepare* erfolgt, sondern es wird sofort die Methode *commit* aufgerufen mit dem entsprechenden Statusflag, welches das *One Phase Commit* an die XAResource signalisiert. Nachdem das Commit auf allen Ressourcen durchgeführt wurde, wird das Ergebnis den registrierten Synchronization-Objekten über *afterCompletion* mitgeteilt.

Nach der Durchführung des Commits wird entweder die suspendierte lokale Transaktion aus dem TxCookie geholt und über *resumeLocalTx* wieder aktiviert (Abbildung 24, 1.4 und 1.5) oder die suspendierte globale Transaktion (Abbildung 24, 1.6 und 1.7), bevor die Ausführungskontrolle wieder an den EJSContainer zurückgeht.

Im Fall eines wirklichen *Two Phase Commits* genügt die oben dargestellte optimierte Variante nicht aus. Es muss ein vollständiges *Two Phase Commit*-Protokoll durchgeführt werden (Abbildung 25). Dies zeichnet sich dadurch aus, dass im Aufruf von *distributePrepare* auf RegisteredResources nicht der Aufruf von *commit_one_phase* erfolgt, sondern der Aufruf der Methode *prepare*, welche von der JTAResource an die XA-Resource delegiert wird. Nur wenn alle Instanzen von XAResource ein Erfolgsflag bei der Ausführung von *prepare* zurückgemeldet haben, wird vom UOWCoordinator die Methode *internalCommit* und damit auch *distributeCommit* aufgerufen (andernfalls *distributeRollback*). Die Methode *distributeCommit* führt auf jeder JTA-Resource die Methode *commit* aus, welche diesen Aufruf wiederum an die assoziierte XAResource delegiert. Nachdem das Commit auf allen Ressourcen durchgeführt wurde, wird das Ergebnis den registrierten Synchronization-Objekten über *afterCompletion* mitgeteilt.

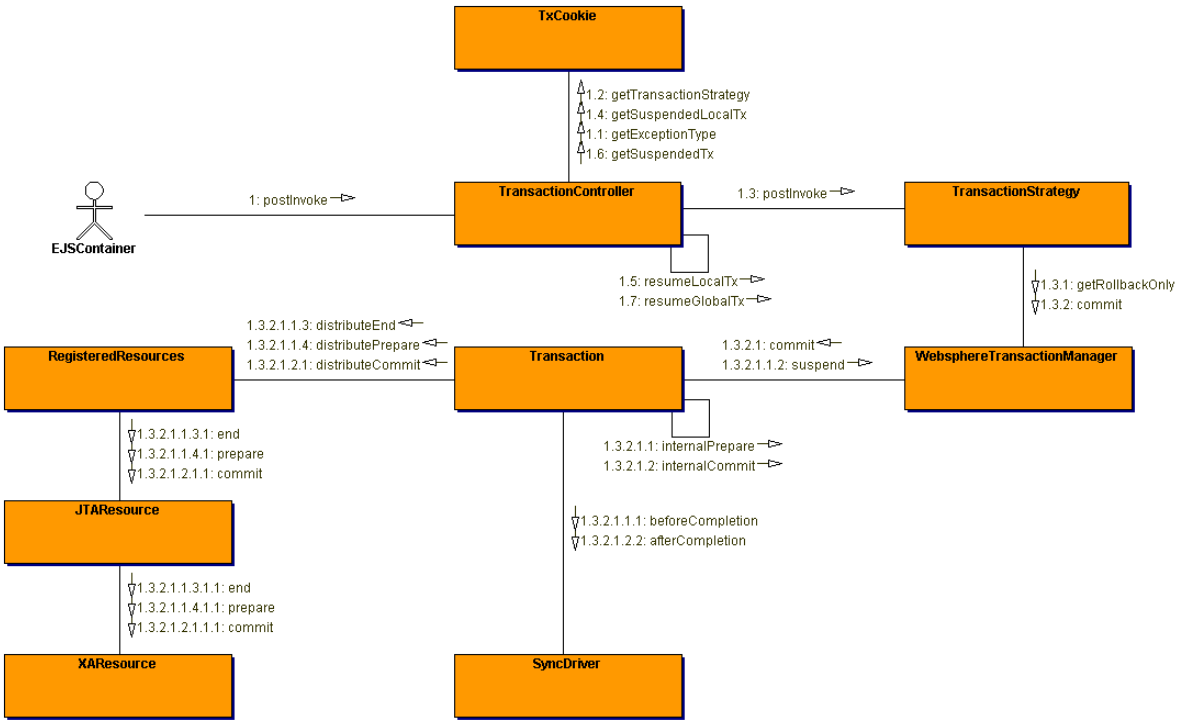

Abbildung 25 TransactionController des EJSContainer nach Aufruf einer SessionBean (Two Phase Commit)

4.3 Vergleich von JBoss und Websphere

JBoss als Opensource-Implementierung der J2EE-Spezifikation trennen nicht nur lizenzspezifische Aspekte von der kommerziellen Implementierung Websphere von IBM. Allein der Umfang der beiden Produkte unterscheidet sich recht beträchtlich. JBoss begnügt sich mit

knapp über 50 MB bei der Installation, Websphere schlägt mit 400 MB zu Buche, zuzüglich 60 MB für die Deinstallation. Auch von der Anzahl an Klassen liegen JBoss und Websphere weit auseinander. JBoss besteht aus knapp 4.300 Klassen, Websphere aus über 20.000.

Die Stärken von JBoss liegen klar in seiner Flexibilität und dem modularen Aufbau, welche einen einfachen Zugang zu Verständnis und Funktionalität ermöglicht. Die einfache Erweiterbarkeit durch eigene Komponenten und die Austauschbarkeit selbst von Kernkomponenten wie dem Transaktionsmanager ermöglichen die fallweise Anpassung auf konkrete Bedürfnisse des Anwenders. Beispielsweise lassen sich durch Implementierung eines eigenen Interzeptors und entsprechender Konfiguration des Containers sehr einfach zusätzliche Funktionalitäten in einen EJB-Aufruf einklinken. Websphere ist allein wegen seiner großen Anzahl von Klassen eine echte Herausforderung. Obwohl über verschiedene Konfigurationsdateien die Implementierungsklassen für einige Dienste (z.B. auch Transaktionsmanager) angegeben werden können, reichen die Möglichkeiten der Einflussnahme nicht an die von JBoss heran, was aber auch bestimmt nicht in der Absicht von IBM liegen dürfte.

Auffällig bei JBoss ist sein „dynamischer Aspekt“. Die Kommunikation zwischen Client und Server sowie die Abbildung von `EJBHome` bzw. `EJBObject` auf konkrete EJBs funktioniert bei JBoss über *Dynamic Proxies* und damit über die Reflection API. Obwohl in den letzten Releases der JVMs viel bezüglich der Performanz der Reflection API getan wurde, ist die Ausführungsgeschwindigkeit kleiner als bei einem herkömmlichen Methodenaufruf. Dies wird einer der Gründe sein warum Websphere konsequent den Gebrauch der Reflection API vermeidet. Während des *Deployments* wird für jedes Bean zusätzlicher Code generiert und kompiliert, welches die Kommunikation zwischen Client und Server, die Abbildung von `EJBHome` und `EJBObject` (`EJBWrapper` bei Websphere) auf konkrete EJBs bis hin zu den SQL-Statements für den Zugriff auf die Datenbank in normale Methodenaufrufe verpackt.

Ein zweiter sehr auffälliger Aspekt bei Websphere, v.a. in den zentralen Komponenten der Transaktionsverarbeitung, ist die große Menge von *Lines of Code (LOC)* für Fehlerbehandlung. Beispielsweise haben die beiden zentralen Klassen `RegisteredResources` und `TransactionImpl` (Implementierung des JTA-`Transaction`-Interfaces und von Websphere `UOWCoordinator`) jeweils ca. zweitausend LOC, wovon mehr als 80% für die Fehlerbehandlung zuständig sind. Dies bewegt sich bei JBoss ca. bei 40%.

Was noch bei dem Quellcode der `TransactionImpl`-Klasse besonders auffällt, ist der recht untypisch für Java verwendete Programmierstil, der sich auch von den restlichen Klassen deutlich abhebt (z.B. werden sehr kurze Anweisungen und viele GOTO-Verzweigungen verwendet). Dies legt die Vermutung nahe, dass die an dieser Stelle verwendeten Algorithmen nicht speziell für den Websphere-Transaktionsmanager entwickelt wurden. Eine naheliegende wenngleich keinesfalls schlüssige Erklärung wäre die Übernahme dieser Quellcode-Stellen von langjährig erfolgreich eingesetzten und verifizierten Algorithmen, evtl. sogar

aus Transaktionsmonitoren wie CICS. Eine Festigung dieser These Bedarf jedoch weiterer Nachforschungen.

5 Problematiken der Transaktionsverarbeitung

Mit dem Verständnis der Funktionsweise von Transaktionsverarbeitung, sowohl auf der Ebene der J2EE-Spezifikation, als auch konkreter Produkte, lässt sich nun die Frage nach der Transaktionssicherheit von J2EE-Produkten untersuchen (Frage 3 aus Kapitel 1.1). Um eine Antwort auf diese Frage zu finden, muss zunächst die Transaktionssicherheit selbst betrachtet werden und wie sie von einem System kompromittiert werden kann. Es stellen sich somit folgende Fragen:

- Was ist unter dem Begriff Transaktionssicherheit zu verstehen?
- Was gefährdet die Transaktionssicherheit bei J2EE-Systemen?
- Existieren Lösungsansätze für diese Problematiken und wie ist ihre Funktionsweise?
- Wie unterscheiden sich die Ansätze, wo liegen Vorteile und Nachteile?

Diesen Fragen sollen in diesem Kapitel untersucht werden.

5.1 Was ist Transaktionssicherheit?

Transaktionssicherheit verstehen wir als qualitative Eigenschaft eines transaktionsverarbeitenden Systems, um das gesicherte und isolierte Ausführen von Transaktionen zu ermöglichen. Ein transaktionsverarbeitendes System muss folgende Punkte hinsichtlich Transaktionssicherheit erfüllen:

1. Gewährleistung der ACID-Kriterien für alle Transaktionen (erfolgreich durchgeführte oder fehlgeschlagene) bei jeglichen Systemzuständen, d.h. selbst in Fehlerfällen muss das System die ACID-Kriterien für die momentanen, sich in Durchführung befindlichen Transaktionen sicherstellen. Dies kann auch bedeuten, dass das System eine fehlerhafte Transaktion zurückrollt und sich neu initialisieren muss. Auf gar keinen Fall dürfen jedoch die ACID-Kriterien unterlaufen werden.
2. Das System muss die Transaktion ohne störende Rahmenfaktoren welche das Ergebnis der Transaktion beeinflussende könnten ausführen, dies bedeutet:
   - parallel durchgeführte oder fehlgeschlagene Transaktionen dürfen sich nicht außerhalb der transaktional veränderten Datenbasis der betroffenen Datenbank beeinflussen.
   - Fehlerzustände und Fehlerbehandlungen des Systems dürfen keine inkorrekte Ausführung der momentanen oder nachfolgenden Transaktionen zur Folge haben. Die Transaktion muss, falls ihre korrekte Ausführung nicht mehr garantiert werden kann, zurückgerollt werden.
   - durchgeführte Transaktionen dürfen nachfolgende Transaktionen nur über die transaktional veränderte Datenbasis beeinflussen.

Dass ein transaktionsverarbeitendes System das Ziel haben sollte, Transaktionen erfolgreich auszuführen und nicht ständig Ausnahmesituationen und Fehlerzustände zu verursachen, wird an dieser Stelle weniger als Anforderung an die Transaktionssicherheit eines System als vielmehr generellen Anspruch an ein ausgereiftes Produkt erachtet.

5.2 Wie gefährdet ein System die Transaktionssicherheit?

Die erste Anforderung aus Kapitel 5.1 an ein transaktionsverarbeitendes System, dass Transaktionen unter Einhaltung der ACID-Kriterien ausgeführt werden müssen, wird als grundlegend angesehen und hier nicht näher betrachtet. Sehr viel tiefgreifender und schwerwiegender sind die Problematiken der Isolation von Transaktionen gegenüber nebenläufigen und nachfolgenden Transaktionen, sowie gegen das ausführende System wie es in der zweiten Anforderung aus Kapitel 5.1 definiert wird. [Borm01] führt dazu folgende Grade der Isolationsproblematik an:

- **Beeinflussung:** Transaktion T1 beeinflusst den globalen Systemzustand (in Java beispielsweise durch die Manipulation von statischen Variablen). Transaktion T2 wird anschließend im Rahmen dieses geänderten Systemzustandes ausgeführt, welches auch Einfluss auf das Ergebnis der Transaktion T2 hat.
- **Zuverlässigkeit:** das fehlerhafte Ausführen einer Transaktion T1 verursacht im System einen Fehlerzustand, so dass alle momentan aktiven Transaktionen Tx (x = {1..n}) fehlschlagen (beispielsweise in Java durch einen schwerwiegenden Fehler in einer JNI-Methode oder einer Systembibliothek oder durch Konsumierung des gesamten verfügbaren Speichers für die JVM).

Beide Problematiken sind verhängnisvoll. Eine Beeinflussung würde schon Systeme mit geringem Transaktionsvolumen katastrophal treffen, da Ergebnisse verfälscht werden. Die Zuverlässigkeit ist vor allem bei großen Systemen gefragt, da sich hier das Aufspüren von fehlgeschlagenen Transaktionen und die erneute Durchführung schnell zu einem erheblichen Zeitfaktor kumuliert, vom Vertrauensverlust des Anwenders gegenüber dem System ganz zu schweigen.

Das Problem trifft dabei nicht allein Anwendungen auf Basis der Programmiersprache Java. Jede Programmiersprache bzw. jede Laufzeitumgebung, welche die Nebenläufigkeit innerhalb von Betriebssystemprozessen unterstützt, trifft auf dieselben Isolationsproblematiken. Die Isolation von Anwendungen, welche in separaten Betriebssystemprozessen ausgeführt werden, wird durch das Betriebssystem in der Weise sichergestellt, dass für die nebenläufigen Anwendungen separate Speicherbereiche angelegt und die Zugriffe auf die Ressourcen streng

geregelt werden. Ein Fehler in einer Anwendung bezieht sich deshalb auch im Normalfall nur auf den eigenen Prozess. Eventuell noch geöffnete Ressourcen, werden vom Betriebssystem selbst durch entsprechende Mechanismen und Timer-Steuerungen aufgeräumt. Das Betriebssystem rechnet man in die Kategorie der *Trusted Applications.* Es hat einen langen Entwicklungs- und Testzyklus hinter sich und man darf ihm deshalb das Vertrauen entgegenbringen, dass es seine Aufgaben kennt und diese ordnungsgemäß verrichtet.

An dieser Stelle lässt sich also schon eine (theoretische) Lösung formulieren, wie eine sichere und isolierte Ausführung von Transaktionen durch ein Java-System erfolgen kann. Das transaktionsverarbeitende System startet für jede durchzuführende Transaktion einen neuen Betriebssystemprozess, das Betriebssystem - als *Trusted Application* - kümmert sich um die Isolation und der Transaktionsmanager um die korrekte Durchführung hinsichtlich der ACID-Kriterien. Obwohl durch seine Einfachheit bestechend hat dieser Ansatz einige Nachteile, welche ihn in der Praxis nicht anwendbar machen:

- **Performanz:** wie in [Bey04] ausgeführt, ist der Startvorgang einer JVM ein signifikanter Zeitfaktor, welcher in vielen Fällen um ein vielfaches länger sein dürfte als die Ausführungszeit einer einzelnen Transaktion. Das Starten von separaten Betriebssystemprozessen und JVMs für jede neue Transaktion hat bei Messungen die Performanz des Gesamtsystems um einen Faktor von mehr als 300 einbrechen lassen.
- **Durchsatz von Transaktionen pro Sekunde:** stark mit dem Problem der Performanz ist auch der Durchsatz des Systems verbunden. Eine schlechtere Performanz der Einzeloperation bedeutet eine schlechtere Performanz des Gesamtsystems und damit eine Senkung der Abarbeitungsgeschwindigkeit des Systems. Der Durchsatz ist in den angeführten Messungen in Abhängigkeit zur Anzahl der emulierten parallelen Benutzer um einen Faktor im Bereich von 200 bis über 300 eingebrochen.
- **Ressourcenverbrauch:** da für jede Transaktion ein neuer Betriebssystemprozess erzeugt und mit Ressourcen versehen wird, sowie eine neue Instanz der JVM mit allen System- und Anwendungsklassen geladen werden muss, hat dieser Ansatz einen höheren Ressourcenverbrauch als ein entsprechender Multi-Threading-Ansatz. Der Ressourcenverbrauch, sowie die Anzahl der parallel startbaren JVMs, hängt dabei stark von den maximalen Heap-Einstellung für die JVM ab (über Option -Xmx beim Starten der JVM anzugeben).
- **Management der Anwendungen innerhalb der Prozesse:** über Betriebssystemprozesse entkoppelte Anwendungen sind zwar hervorragend isoliert, es fällt jedoch schwer diese separaten, rasch fluktuierenden Prozesse zu verwalten und zu kontrollieren.
- **Kommunikation zwischen Transaktionen:** Für Transaktionen, welche nicht alle Informationen oder Funktionalitäten in sich selbst beinhalten und damit zur Erbringung der Gesamtleistung mit anderen Komponenten (welche evtl. Transaktionen starten) kollaborieren müssen, bringt die Isolation mittels Betriebssystemprozessen ein weiteres Problem mit sich. Kommunikation über Prozessgrenzen hinaus muss von einer JVM zur Anderen

serialisiert, übertragen und wieder deserialisiert werden. Der Overhead dieser Kommunikationsform senkt dabei die Performanz und den Durchsatz des Gesamtsystems.

Insgesamt gesehen ist dieser erste Lösungsansatz, obwohl theoretisch möglich, in der Praxis wenig erfolgversprechend. Es werden deshalb im folgenden Kapitel alternative Lösungsansätze betrachtet, ihr Funktionsprinzip erläutert, sowie die Vor- und Nachteile diskutiert.

5.3 Isolation mittels Prozessen

Die prozessorientierte Lösungsansatz ist dem zuvor beschriebenen Ansatz am ähnlichsten. Die Isolation der Transaktionen voneinander wird durch die Aufspaltung auf verschiedene JVMs in separaten Betriebssystemprozessen erreicht. Die *Persistent Reusable JVM (PRJVM)*, welche in [Borm01] und [Dill00] vorgestellt wird, ist nach diesem Prinzip in den IBM Labors in Hursley für die z/OS-Plattform entwickelt worden.

Funktionsprinzip:

Eine *Master*-JVM und mehrere *Worker*-JVMs teilen sich einen gemeinsamen Adressraum im Speicher (*System Heap*). Die *Master*-JVM ist dafür verantwortlich, dass der *System Heap* zur Verfügung steht und dass die gemeinsame Classloader-Umgebung bereit steht. Die *Worker*-JVMs verrichten die eigentlichen Transaktionen. Am Ende einer Transaktion werden jedoch die *Worker*-JVMs nicht zerstört und neu erzeugt, stattdessen werden die JVMs auf ihre *Reset*-Fähigkeit untersucht. Falls dies zutrifft, wird ein *Reset* auf der *Worker*-JVM durchgeführt, falls nein wird sie zerstört und neu erzeugt. Die *Reset*-Fähigkeit wird dadurch zu erreichen versucht, dass man den Heap aufsplittet in verschiedene Domänen (System, Middleware, Transient). Jeder Heap hat seinen eigenen Classloader.

System: hier werden Klassen-Objekte für die *shareable* Klassen angelegt. Das meiste geschieht dabei beim Starten der JVM. Diese Klassen haben eine Lebensdauer bis zum Ende der JVM. Es findet keine Garbage Collection statt.

Middleware: hier werden vorwiegend *non-shareable* Middleware-Klassen abgelegt und Middleware-Objektinstanzen. Der Zustand dieser Objekte bleibt über Transaktionsgrenzen hinaus erhalten. Es greift eine Standard Garbage Collection Policy (z.B. *mark/sweep*).

Transient: hier werden *non-shareable* Anwendungs-Klassen sowie Anwendungs-Objekte abgelegt. Die Allokationsrate ist hoch, die Lebensdauer ist kurz und nur auf die Transaktion bezogen. Die Garbage Collection Policy sollte ein schnelles Löschen dieser transaktionalen Daten erlauben.

Am Ende einer Transaktion wird versucht, ein *Reset* auf die JVM durchzuführen. Dies geschieht in der Weise, dass zunächst die JVM geprüft wird, ob ein *Reset* möglich ist:

- Die JVM darf keine ausstehenden Ausnahmebedingungen (*Exceptions*) haben, ansonsten kann die JVM nicht zurückgesetzt werden.
- Auf die Middleware-Objekte wird eine `tidyUp()`-Methode aufgerufen, welche von den Middleware-Klassen dazu genutzt werden kann, die Ressourcen und den inneren Zustand zwischen Transaktionen aufzuräumen. Die Middleware-Objekte müssen v.a. jegliche Referenzen zu Anwendungsobjekten entfernen, da sonst die JVM nicht zurückgesetzt werden kann. Falls dies nicht gelingt, muss der Wert „false“ zurückgegeben werden (JVM wird nicht zurückgesetzt), andernfalls „true“ (JVM kann zurückgesetzt werden).
- Innerhalb der JVM darf kein Benutzer-Thread mehr laufen.
- In den Anwendungsklassen darf kein *Unresettable-Event* aufgetreten sein.

Ist eine der Prüfungen negativ verlaufen, darf die JVM nicht zurückgesetzt werden. Sie muss dann neu erzeugt werden. Sind alle Überprüfungen erfolgreich verlaufen, ist die JVM im Zustand *Resettable*. Die konkrete *Reset*-Logik wird ausgeführt:

- Die Standard-Anwendungs-Classloader werden aus der Classloader-Hierarchie entfernt.
- Mit einem *ResetGC* wird der Transient-Heap wieder in seinen initialen (leeren) Zustand überführt.
- Neue Anwendungs-Classloader werden erzeugt und zur Classloader-Hierarchie hinzugefügt. Der Bytecode und JIT- kompilierte Code für *shareable* Anwendungsklassen ist schon vorhanden. Die entsprechenden Klassen werden mit „geladen aber noch nicht initialisiert“ markiert, d.h. statische Initialisierungsmethoden müssen beim ersten Zugriff ausgeführt und statische Variablen zurückgesetzt werden.
- Middleware-Klassen sind ebenfalls im Zustand „geladen aber noch nicht initialisiert“. Hier müssen dieselben Aktionen wie oben durchgeführt werden.

Damit die PRJVM ihre Performanz-Vorteile, i.S. eines *Reset* gegenüber einer Neuerzeugung einer JVM ausspielen kann, müssen die Anwendungen einigen Regeln genügen. Interessanterweise sind diese Regeln eine Untermenge der Reglementierungen aus der EJB-Spezifikation:

- Keine System-Eigenschaften (`System.setProperty()`) dürfen verändert werden.
- Keine Systembibliotheken (*native libraries*) dürfen geladen werden.
- Keine neuen Prozesse oder Threads dürfen erzeugt werden.

Im Vergleich dazu wird den Middleware-Klassen keine Restriktionen auferlegt, außer das Threads zu beenden sind und die Referenzen zu Anwendungsklassen gelöst werden müssen. Dieses Prinzip ist ebenfalls aus der EJB-Spezifikation bekannt.

Vorteile:

- Isolation von parallelen Transaktionen durch Prozessgrenzen. Bei Fehler in einem Prozess, wird durch das Betriebssystem die Isolation gewährleistet. Andere Prozesse werden dadurch nicht kompromittiert.
- Isolation von nachfolgenden Transaktionen durch *Reset*-Fähigkeit bzw. Neuerzeugung der JVM gewährleistet. Die Transaktion findet in jedem Fall einen „sauberen“ Anwendungs-Heap vor.
- Das Zurücksetzen der JVM liefert eindeutige Performanz-Vorteile gegenüber einer kompletten Neuerzeugung der JVM.

Nachteile/Probleme:

- Der Performanz-Gewinn hängt stark von den Anwendungen selbst ab. Wenn die ausgeführte Anwendung gegen die *Unresettable-Event*-Regeln verstößt, muss die komplette JVM neu erzeugt werden, was die Leistungsfähigkeit des Gesamtsystems beträchtlich in Mitleidenschaft zieht. Solche *Unresettable-Events* zu erzeugen ist dabei nicht sonderlich schwierig, da viele Implementierungen der Java-Klassen unter der Decke den System-Zustand manipulieren (z.B. `java.util.Calendar` setzt bei der ersten Verwendung eine Instanz von `java.util.Locale` in den *System Heap*).
- In [Borm01] wird schon darauf hingewiesen, dass der Ansatz der PRJVM auf kurzlebige Transaktionen zielt, welche keine Kommunikation außer der eigenen transaktionalen Datenquelle benötigt. Sobald eine Kommunikation über Prozessgrenzen hinzukommt, wird die Serialisierung/Deserialisierung des Aufrufs notwendig, welche den Performanz-Gewinn des PRJVM-Ansatzes ebenfalls stark negativ beeinträchtigt.
- Ein Betriebssystem-Kernel, welcher für die Isolation der Prozesse zuständig ist, dürfte auf kommerziellen Betriebssystemen gut getestet sein. Jedoch kann nicht davon ausgegangen werden, dass er fehlerfrei ist. Deshalb kann es Fälle geben, in welcher auch eine derartige Isolation nicht greift und eine Vielzahl von Prozessen – und damit Transaktionen – in Mitleidenschaft gezogen werden.
- Trotz der Einrichtung eines *Worker*-Pools aus JVMs, so dass nicht für jede Transaktion eine neue JVM gestartet werden muss, ist die maximal erreichbare parallele Abarbeitung limitiert durch den Faktor, wieviele JVMs sich parallel auf einem System starten lassen. Die Anzahl der maximal parallel lauffähigen JVMs ist aber um einige Zehnerpotenzen kleiner als die Anzahl von Threads in einer JVM.

Im Hinblick auf die EJB- bzw. J2EE-Spezifikationen ist das Prinzip der PRJVM integrierbar. Die Spezifikationen machen keine Vorschrift, wie die Abarbeitung von parallelen Anfragen durchgeführt werden soll. Eine mögliche Integration der PRJVM-Fähigkeiten könnte sein, dass es einen Thread-Worker-Pool gibt für nicht-transaktionale Arbeit und einen Pro-

zess-Worker-Pool für transaktionale Arbeit. Im Thread-Worker-Pool könnten rein lesende Aufrufe abgehandelt werden oder Aufrufe, für die die Performanz bei eingeschränkten ACID-Kriterien legitim ist, wie beispielsweise das Bereitstellen von Daten für Web-Anwendungen. Hierbei könnten im selben Thread der Aufruf eines Clients im Web-Container angenommen werden, an ein Session-EJB delegiert, welches wiederum einige EntityBeans benutzt, um die Anwendungsdaten aus der Datenbank zu holen. Anschließend wird das Resultat ebenfalls noch im selben Thread an den Benutzer zurückgeliefert.

Sobald transaktionale Arbeit vollbracht werden muss, sollte die Transaktion in einer Worker-Queue abgelegt werden, aus welcher der Prozess-Worker-Pool über einen geeigneten Mechanismus die Aufrufe entnimmt und in seinem eigenen Prozess durchführt. Wenn ein Stream-Handle für die Antwort an den Benutzer im Aufruf-Objekt der Worker-Queue mitgegeben wurde, kann im selben Prozess auch die Rückantwort an den Benutzer erfolgen, was einen weiteren „teuren“ Prozesswechsel unnötig macht.

Besonders praktikabel erscheint dieses Prinzip jedoch genau dann, wenn keine Rückantwort an einen Benutzer erfolgen muss, also in einem asynchronen Verarbeitungsschritt (z.B. über *Message-Driven Beans*), da hier der Performanz-Verlust beim Prozesswechsel zugunsten der strikten ACID-Einhaltung vernachlässigt werden kann.

5.4 Isolation mittels Classloader

Fast alle J2EE-Implementierungen arbeiten mit Multi-Threading für ihre Worker-Pools. Prozesse werden aus Gründen der Performanz nicht eingesetzt (gemeint ist hier die Performanz bei Kollaboration zwischen verschiedenen Worker-Prozessen). Die Isolation wird mittels Classloader-Hierarchien und Restriktionen in der EJB-Spezifikation adressiert.

Funktionsprinzip:

Der *Bootstrap*-Classloader ist für das Laden der System-Klassen zuständig, der *Standard Extension*-Classloader für die Standardbibliotheken. Für jedes EAR wird ein neuer Classloader benutzt. Dadurch wird eine Beeinflussung zwischen unterschiedlichen Anwendungen (EARs) vermieden. Selbst wenn in unterschiedlichen EARs dieselben Klassen benutzt und dort statische Werte zur Laufzeit in Anwendung A1 verändert werden, würde die Anwendung A2 davon nichts bemerken, da diese Veränderung auf unterschiedlichen Kopien der Klassen innerhalb des Classloaders stattfindet. Für jedes Modul des *Assembly Descriptors* (`application.xml`) wird ebenfalls ein neuer Classloader erzeugt, d.h. die Isolation besteht nicht nur zwischen Anwendungen (EARs), sondern auch zwischen den Modulen (JARs) innerhalb der eigenen Anwendung.

Des weiteren sind in der EJB-Spezifikation einige Restriktionen bzw. Garantien aufgeführt, welche die Isolation und Stabilität zur Laufzeit sicherstellen sollen:

- Keine Verwendung von statischen Klassenvariablen innerhalb von EJBs.
- Keine JNI-Aufrufe innerhalb von EJBs. JNI-Aufrufe dürfen nur vom EJB-Container durchgeführt werden (dies ähnelt anderen Ansätzen wie z.B. der PRJVM).
- Keine Erzeugung von Threads innerhalb von EJBs.
- Keine Referenz auf die Bean-Klasse darf an einen Client übergeben werden.
- Ein EJB wird nur von einem Thread gleichzeitig durchlaufen.
- Folgende Sicherheitsrestriktionen gelten innerhalb von EJBs:
  - java.security.AllPermission (verweigert)
  - java.awt.AWTPermission (verweigert)
  - java.io.FilePermission (verweigert)
  - java.net.NetPermission (verweigert)
  - java.util.PropertyPermission (nur lesen)
  - java.lang.reflect.ReflectPermission (verweigert)
  - java.lang.RuntimePermission (nur Einstellen eines Druckauftrags ist erlaubt)
  - java.lang.SecurityPermission (verweigert)
  - java.io.SerializablePermission (verweigert)
  - java.net.SocketPermission (nur *connect* erlaubt)

Die Restriktionen werden (außer den Sicherheitsaspekten) während der *Deployment*-Phase überprüft. Wenn diese verletzt werden, sollte ein *Deployment* verweigert werden. Die EJB-Spezifikation erlaubt jedoch v.a. im Bereich der Sicherheit die Lockerung dieser Restriktionen. Viele J2EE-Produkte unterstützen deshalb einen Modus, welcher auf strikte Einhaltung der Restriktionen prüft und einen Modus, welcher solche Fehler ignoriert.

Vorteile:

- Kein Performanz-Verlust für Kommunikation zwischen Prozessen, da der Applicationserver in einem Prozess ausgeführt wird. Um Ressourcen effizient zu nutzen, führen moderne Applicationserver bei jedem I/O-Zugriff ein *suspend* auf den laufenden Aufruf aus und ein *resume* wenn der I/O-Aufruf zurückkehrt. Dieser Ansatz würde immense Performanz-Einbußen bei einem prozessorientierten *Worker-Pool* mit sich bringen.
- Bei Entwicklung gemäß EJB-Spezifikation ist eine Modifikation des globalen Systemzustandes nicht möglich. Es wird also die Beeinflussung von parallelen Transaktionen adressiert.
- Die EJB-Spezifikation sieht Callback-Methoden vor (`Synchronization`-Objekte), welche genutzt werden können, um nach erfolgreicher Transaktion den Zustand des EJBs zurückzusetzen. Die Beeinflussung von nachfolgenden Transaktionen wird damit adressiert.

- Kritische Funktionalitäten (JNI, *Worker-Pool*, *Pooling*, Transaktionsmanager) obliegen der J2EE-Infrastruktur und werden vom EJB nur benutzt. Das Risiko durch fehlerhafte Implementierung von Bean-Klassen (welche nicht durch normale Testzyklen auffallen) kritische Fehlerzustände in der JVM zu produzieren, sinkt dadurch erheblich. Verantwortlich für die Isolation im Sinne der Zuverlässigkeit ist also die Implementierung des J2EE-Produktes. Ob man das J2EE-Produkt bzw. die JVM als genauso vertrauenswürdig wie ein Betriebssystemkernel erachtet, ist hierbei zwischen den einzelnen Vertretern noch strittig [Cza00].

Nachteile/Probleme:

- Die Standard-Konfiguration der meisten EJB-Container folgt nicht den starken Restriktionen der EJB-Spezifikation. Statische Klassenvariablen sind beispielsweise beim *Deployment* möglich. Dadurch lässt sich durch Misskonfiguration die Isolation beliebig unterlaufen.
- Die Verantwortung für das saubere und rückstandslose Aufräumen der Ressourcen wird dem Anwendungsentwickler überlassen. Er muss die entsprechenden Schnittstellen und Methoden korrekt implementieren. Nur der Aufruf und die Ausführung wird vom EJB-Container sichergestellt.
- Eine J2EE-Implementierung sowie eine JVM besteht aus mehr LOCs als ein normaler Betriebssystemkernel. Da die Fehlerwahrscheinlichkeit mit zunehmender Komplexität auch größer wird, sind Fehler welche kritische Systemzustände auslösen, bei J2EE- bzw. JVM-Implementierungen schon rein statistisch größer als bei einem Betriebssystem-Kernel. Jedoch erfolgen die meisten JVM-Implementierungen auch direkt von den Herstellern für ihr Betriebssystem (IBM für AIX, z/OS; Microsoft für Windows). Es stellt sich hier die berechtigte Frage, warum Hersteller von Betriebssystemen und JVMs nur fähig sein sollen auf Ebene des Betriebssystems die Isolation zu gewährleisten.

Einige weitere Beispiele, welche hier nur der Vollständigkeit halber erwähnt werden sollen, benutzen ebenfalls Classloader als Isolationsmittel:

- **Echidna:** Dies ist eine Klassen-Bibliothek, welche es mehreren Anwendungen erlaubt innerhalb derselben JVM zu laufen. Die durch eine Anwendung allokierten Ressourcen werden vollständig deallokiert bei Beendigung der Anwendung [Gorr04].
- **J-Kernel:** Es wird das Konstrukt der *Protection Domains* zu der Java-Programmiersprache hinzugefügt und eine starke Unterscheidung getroffen zwischen Objekten welche zwischen *Tasks* genutzt werden können und welche zu einem *Task* allein gehören müssen. Jede *Protection Domain* hat ihren eigenen Classloader. J-Kernel verfügt über eine saubere Beendigung der *Protection Domain*. Die durch eine Anwendung allokierten Ressourcen werden vollständig deallokiert [Haw98].

5.5 Isolation durch Modifikation der JVM

Einige Ansätze existieren bereits, wie eine Isolation innerhalb der JVM durchgeführt werden kann.

Funktionsprinzip:

[Cza00] führt ein Projekt an der Universität von Utah an, welches zwei unterschiedliche Ansätze für die Implementierung eines Prozessmodells in einer JVM durchgeführt hat [Back98].

GVM ist aufgebaut wie ein monolithischer Betriebssystemkernel und bietet vollständige Isolation und Kontrolle von Ressourcen zwischen Prozessen. Ein GVM-Prozess besteht aus einem *Namespace,* welcher mittels Classloader realisiert ist, einem Heap und einer Anzahl von Threads auf diesem Heap. Jeder Prozess hat seinen eigenen Heap und alle Prozesse können auf einen gemeinsamen Heap zugreifen. Für jeden Heap werden alle Referenzen zu und von anderen Heaps registriert und damit eine Art verteilte Garbage Collection realisiert.

Alta ist modelliert als Microkernel mit verschachtelten Prozessen. Der Vater-Prozess verwaltet alle Ressourcen für die Kind-Prozesse. Speicherverwaltung wird explizit durch ein einfaches *Allocator-Pays*-Schema unterstützt. Der Garbage Collector vermerkt den Eigentümer-Prozess wenn ein Objekt erzeugt wird. Da Alta *Cross-Process*-Referenzen erlaubt, wird jedes existierende Objekt dem Speichersegment des Vater-Prozesses hinzugefügt.

Der Ansatz an Isolation welcher in [Cza00] vorgeschlagen wird ist leichtgewichtiger als die beiden obigen Varianten. Anwendungen teilen dabei physikalisch denselben Heap, obwohl der Heap logisch getrennte Graphen von Objekten für unterschiedliche Anwendungen beinhaltet. Jede Klasse wird nur einmal geladen. Enthält die Klasse statische Elemente (statische Klassenvariablen mit statischen Initialisierungsmethoden) werden diese pro Anwendung kopiert. Dieses kopieren sowie modifizieren der Zugriffsmethoden kann durch Java-Bytecode-Manipulation oder durch Manipulation der JVM erzielt werden.

Vorteile:

- Im Gegensatz zu Classloader-basierten Isolationsansätzen werden Klassen nicht mehrfach geladen, dadurch entsteht ein besseres Speicherverhalten.
- Es ergeben sich Performanz-Vorteile, da Klassen nur einmal geladen werden müssen.
- Jede Anwendung hat seine eigene Kopie von statischen Variablen. Die Beeinflussung von parallelen Transaktionen wird damit adressiert.
- Gegenüber dem EJB-Ansatz bei welchem viel Verantwortung beim Anwendungsentwickler liegt, wird diese Form der Isolation automatisch von der JVM bereitgestellt. Dies macht sie zuverlässiger.

- Gegenüber Isolationsansätzen mittels Prozessen gibt es keine Performanz-Einbußen bei Kollaboration von Komponenten.

Nachteile/Probleme:

- Die Isolation von nachfolgenden Transaktionen ist mit diesem Ansatz nicht direkt gegeben.
- Die Isolation bzgl. Zuverlässigkeit wird in der JVM angesiedelt (vgl. Darstellung und Argumentation in Kapitel 5.4).

5.6 Isolation mit der Isolate API

An der Spezifikation der Isolate API haben sich zahlreiche Firmen (u.a. Sun, IBM, SAP, Oracle, Philips, HP) beteiligt. In ihr fließen Erfahrungen und Basis-Forschungsarbeit der letzten Jahre ein, die teilweise auch schon in den vorherigen Kapiteln vorgestellt wurden. Die Isolate API wird als JSR 121 im *Java Community Process (JCP)* standardisiert und liegt mittlerweile in der endgültigen Version vor. Die Isolate API ist speziell dafür gedacht, die Ausführung von isolierten parallelen Tasks zu ermöglichen. Welche Technik benutzt wird um die Isolation zu gewährleisten, ist nicht Gegenstand der Spezifikation. Die API ist unter dem Package `java.lang.isolate` angesiedelt und bietet Unterstützung für das erzeugen, starten, beenden, managen und kommunizieren mit anderen isolierten Java-Anwendungen.

Funktionsprinzip:

Java-Anwendungen welche voneinander isoliert ausgeführt werden sollen, werden innerhalb einer Instanz der Klasse `Isolate` ausgeführt (Abbildung 26). Ein `Isolate` kann kein Zugriff auf Objektinstanzen mit einem anderem `Isolate` teilen, ebenso hat jedes `Isolate` eine eigene Kopie der statischen Klassenvariablen (vgl. Kapitel 5.5).

Ein `Isolate` ist ein Konstrukt zwischen Thread und JVM. Wie ein Thread ermöglicht es nebenläufige Ausführung von Anwendungen innerhalb eines Prozesses, jedoch mit JVM Eigenschaften. Die ausgeführte Klasse bekommt ihren eigenen *System Level*-Kontext und ist von anderen parallel ablaufenden Java-Anwendungen unabhängig. Anders als eine JVM bietet ein `Isolate` jedoch eine API zum erzeugen, starten, beenden, managen und kommunizieren mit anderen `Isolate`-Objekten. Eine `Aggregate`-Instanz stellt in der Isolate-API eine Menge von `Isolate`-Objekten dar, welche einen gemeinsamen Erzeuger haben. Direkte Kommunikation ist für `Isolate`-Objekte nur innerhalb derselben `Aggregate`-Instanz erlaubt.

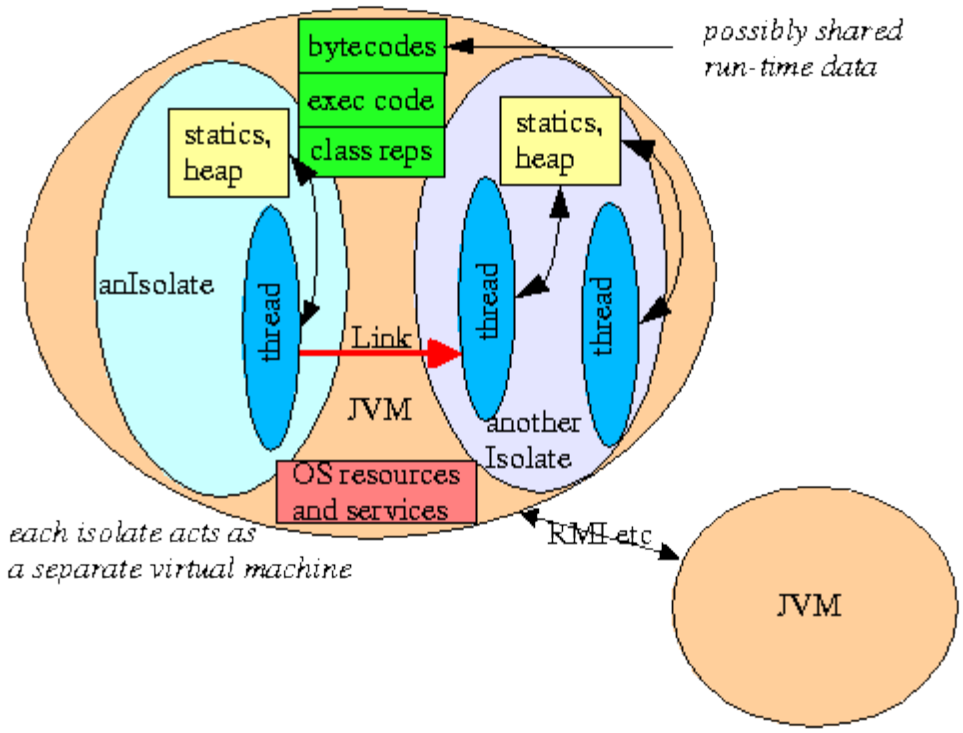

Abbildung 26 Architektur der Isolate API (Quelle [JSR121])

Die Isolate API ist vorgesehen, in drei unterschiedlichen Ansätzen implementiert zu werden:

1. **All-in-one:** Eine JVM wird auf einen Betriebssystemprozess (oder auf Hardware-Ebene) abgebildet. Die Isolation wird innerhalb der JVM implementiert. Dies soll in der neuen Version 1.5 der J2SE von Sun realisiert sein.
2. **One-to-One:** Eine JVM wird auf mehrere Betriebssystemprozesse abgebildet. Ein Prozess pro Isolate. Dieser Ansatz wurde im Rahmen der JanosVM [Janos04] realisiert. Auch die PRJVM ist für eine Implementierung dieses Ansatzes gut geeignet.
3. **Clustered:** *Isolates* befinden sich auf physikalisch verschiedenen Maschinen innerhalb eines Clusters mit einem konsistenten Bild der externen Umgebung (Dateisystem, Dienste,...).

Die folgenden Szenarien wurden für die Entwicklung der Spezifikation insbesondere betrachtet:

- **Unabhängige Anwendungen:** ein System könnte eine JVM beim booten des Systems oder zum Zeitpunkt der Benutzeranmeldung erzeugen und initialisieren. Ein Aufruf des `java`-Kommandos erzeugt dann nur noch ein neues `Isolate`, um die übergebene Anwendung auszuführen.

- **Unterstützung von Frameworks:** Frameworks zur Ausführung von Servlets, Applets u.a. können Optionen bieten, um diese innerhalb von getrennten `Isolate`-Instanzen auszuführen.
- **Server Forks:** ein Netzwerkdienst, welcher Anfragen von Clients entgegennimmt und an eine entsprechende Verarbeitungsinstanz (*Handler*) weitergibt, möchte die einzelnen Verarbeitungsinstanzen aus Stabilitätsgründen als `Isolate`-Instanzen ausführen.
- **Fehlertoleranz:** ein System kann mehrere Versionen von redundanten Kopien wichtiger Dienste in separaten `Isolate`-Instanzen halten, um die Zuverlässigkeit und Erreichbarkeit der Dienste zu erhöhen.
- **Ausführung im Cluster:** analog zum *Beowulf Cluster* können rechenintensive Programme im Cluster parallel ausgeführt werden.

Die Isolate-API definiert nur eine Basis für all diese möglichen Szenarien, ebenso macht es keine Aussagen über die Kontrolle von Ressourcen. Wie tief greifend eine Isolation angestrebt wird, obliegt der jeweiligen Implementierung der Spezifikation.

Die Isolate-API erlaubt das Erzeugen von *Isolates* auf unterschiedlichste Weise:

- Wie Java-Programme mittels Klassenname, Classpath und Argumenten.
- Mittels eines Isolate-Kontextes (basierend auf `java.util.prefs.Preferences`).
- Über einen Kontext auf Basis von JNDI (J2EE).

Alle `Isolate`-Instanzen welche in einer JVM ablaufen teilen einige wenige unveränderbare System-Eigenschaften, wie z.B. die Benutzerkennung, die Sicherheitsrichtlinie oder das Dateisystem. In der Isolate-API ist das Erzeugen der `Isolate` vom Starten der Instanz getrennt (wie auch bei Threads), dabei kann ein `Isolate` analog zu einer JVM entweder „normal“ (wie System.exit) oder „abrupt“ (wie System.halt) beendet werden. Eine JVM in der alle `Isolate`-Instanzen beendet wurden kann sich ebenfalls beenden, da die JVM keinen Nutzen mehr erfüllt.

Obwohl die Isolate-API dem Zweck dient, einzelne Anwendungen voneinander zu isolieren und somit der Gedanke an Kommunikation etwas seltsam erscheinen mag, beinhaltet die API auch einen Kommunikationsmechanismus. Mittels der Klasse `java.lang.isolate.Link` sowie ihre Helferklassen können `java.lang.isolate.IsolateMessage`-Objekte ausgetauscht werden. In der Praxis können `Link`-Implementierungen z.B. über RMI, CORBA, Dateisystem sowie *Shared Memory* realisiert sein. Mit diesem Kommunikationsmodells sind hauptsächlich zwei Fälle adressiert:

- Partitionierung von monolithischen Anwendungen welche Anwendungscode enthalten, der teilweise als zuverlässig eingestuft wird und teilweise nicht. Damit wird die Sicherheit und Stabilität der Programmausführung gesteigert.

- Anwendungs-Frameworks (z.B. Servlet-Container und EJB-Container) folgen einem Container-Manager-Modell. Dabei wird der Container als zuverlässig eingestuft, jedoch die ausgeführten Anwendungskomponenten nicht. Der Container und jede Komponente könnte innerhalb eines eigenen `Isolate` ausgeführt werden, so wäre der Container von seinen Komponenten isoliert, sowie ausgeführte Komponenten voneinander. Fehler in Komponenten können damit nicht zu globalen Fehlerzuständen führen.

Komponenten einer Anwendung sind häufig darauf angewiesen, in gewisser Weise miteinander zu kommunizieren. Im Fall einer Implementierung ohne Isolate-API würde dies über lokale Methodenaufrufe und globale Variablen im Rahmen der JVM erfolgen. Beide Mittel sind bei der Benutzung der Isolate-API nicht vorhanden. Diese nun durch traditionelle Interprozess-Kommunikation zu ersetzen (RMI, Sockets u. a.), würde einen gewaltigen Kommunikations-Overhead mit sich bringen. Die Isolate-API reduziert diesen Overhead in den folgenden Bereichen:

- Eine Kommunikation über Netzwerk ist nicht notwendig. Inter-Isolate-Kommunikation kann auf der Ebene einer JVM-Implementierung über *Shared Memory* durchgeführt werden.
- Keine Notwendigkeit der Serialisierung und Deserialisierung. Inter-Isolate-Kommunikation findet innerhalb einer physikalischen JVM statt.
- In manchen Fällen ist es möglich, den Overhead für das Kopieren von Objekten zwischen `Isolate`-Instanzen zu umgehen. Die JVM muss jedoch, obwohl auf der Implementierungsebene der Speicherbereich des Objektes physikalisch gemeinsam benutzt wird, gegenüber der Anwendung die Illusion eines logischen Kopierens erzeugen.
- In manchen Fällen ist es möglich das Kopieren von Daten auf Anwendungsebene dadurch zu eliminieren, dass Kommunikationsendpunkte zwischen `Isolate`-Instanzen übergeben werden (Pipes, Sockets, Streams u.a.). Falls diese Möglichkeit nicht zur Verfügung steht, müssen die Netzwerkdaten kopiert und zum jeweiligen `Isolate` weitergeleitet werden.

Die Isolate-API definiert kein eigenes Management-Protokoll. Es besteht jedoch die Möglichkeit, über die Java Management Extension (JMX) auch `Isolate`-Instanzen zu verwalten. Des weiteren kann jedes `Isolate` sich auf andere `Isolate`-Instanzen registrieren, um *lifecycle events* zu erhalten. Die Isolate-API garantiert innerhalb ihrer Spezifikation nicht das eine komplette Isolation für *Native Code (JNI)* existiert. Es wird der Implementierung überlassen, welchen Grad an Isolation sie unterstützt. Eine Anwendung kann diesen Grad der Isolation über den `Isolate`-Kontext (`java.lang.isolate.jni-isolation`) abfragen.

Vorteile:

- Standardisierte einfache API für die Isolation von Komponenten.
- Adressiert Isolation bzgl. Beeinflussung von parallelen Transaktionen.
- Adressiert Isolation bzgl. Zuverlässigkeit von parallelen Transaktionen.
- Isolation von nacheinander folgenden Transaktionen wird darüber unterstützt, dass die Erzeugung von `Isolate`-Instanzen leichtgewichtiger ist als die Erzeugung von JVMs. So wäre es denkbar, einen Pool von „sauber initialisierten“ `Isolate`-Objekten zu halten. Für jede Transaktion wird eine Instanz aus dem Pool genommen und nach Ausführung zurückgesetzt und wieder in den Pool gegeben.
- Optimierung von Inter-Isolate-Kommunikation hinsichtlich Performanz.
- Durch die unterschiedlichen Typen (All-in-One, One-to-One, Cluster) werden unterschiedlichste Implementierungen mit verschiedenen Merkmalen erlaubt.
- Für die Anwendungskomponente (z.B. Bean im Container) ändert sich nichts.

Nachteile/Probleme:

- Die Qualität der Isolation wird den einzelnen Implementierungen überlassen, beispielsweise ob die Isolation JNI-Aufrufe beinhaltet oder nicht.

5.7 Vergleich und Zusammenfassung

Es herrscht breiter Konsens darüber, dass die Isolation von nebenläufigen Java-Anwendungen und im speziellen die Gewährleistung von Transaktionssicherheit ein Muss-Kriterium für den Einsatz von Java in beliebigen transaktionsverarbeitenden Systemen darstellt. Darüber hinaus sind Performanz-Überlegungen und der Ressourcenverbrauch ein wichtiges Thema.

Die bisherigen Möglichkeiten über Threads Parallelausführung zu ermöglichen, erscheinen auf Grund der vielfältigen Möglichkeiten zur Beeinflussung als ungeeignet für Systeme, die eine hohe Transaktionssicherheit gewährleisten müssen. Mittels der J2EE/EJB-Spezifikation werden zwar Restriktionen definiert welche die einzelnen Komponenten deutlich voneinander isolieren, jedoch werden diese Restriktionen oft unterlaufen und es hängt viel Verantwortung an der Anwendungsentwicklung. Wünschenswerter erscheint ein Ansatz bei dem der Anwendungsentwickler von dieser Verantwortung entlastet wird und die JVM bzw. der J2EE-Container dies automatisch als Qualitätsmerkmal gewährleisten kann. Diese Thematik wurde von vielen unabhängigen Gruppen näher betrachtet und war Gegenstand von vielfältigen Basis-Forschungsarbeiten. Die unterschiedlichen Ansätze wurden in diesem Kapitel dargestellt und ihre Vorteile und Nachteile untersucht. Grundlegend bleibt der Eindruck, dass in jedem

der vorgestellten Ansätze noch Arbeit investiert werden muss, doch die Ansätze selbst – so gegensätzlich sie manchmal sein mögen – konzeptionell machbar und ausbaufähig erscheinen.

Viele Vertreter der vorgestellten Ansätze haben an der Spezifikation der erst kürzlich fertiggestellten Isolate-API mitgewirkt. Die Isolate-API definiert eine Basis API für die Isolation von Komponenten. Sie kann sowohl von Anwendungsentwicklern als auch Serverframework-Herstellern benutzt werden, um einzelne Komponenten voneinander zu isolieren. Die Güte der Isolation kommt dabei stark auf die Implementierung selbst an. Die möglichen Implementierungstypen sind dabei so offen gehalten, dass jeder der oben diskutierten Ansätze darauf abbildbar ist.

Die Technologie der PRJVM geeignet sich gut für eine *One-to-One*-Implementierung auf Basis von Betriebssystemprozessen. Die in [Cza00] vorgestellte Modifikation der JVM und die Separation von statischen Klassenvariablen ist geeignet für die Implementierung einer All-in-One Variante. Diese Variante ist bereits für die kommende Version 1.5 der J2SE von Sun angekündigt. Dies ist mit ein Indiz dafür, welche hohe Priorität diesem Thema eingeräumt wird.

Auch wenn es bestimmt noch einige Zeit dauert bis die ersten Implementierungen als ausgereift und stabil gewertet werden können, sind die vorgestellten Ansätze und Ergebnisse äußerst vielversprechend. Die konzeptionelle Abstraktionsbasis für unterschiedlichste Implementierung ist mit der Isolate-API ebenfalls gegeben, so dass in absehbarer Zeit nichts mehr gegen einen Einsatz von Java und J2EE in transaktionsverarbeitenden Systemen jeglicher Art sprechen dürfte.

6 Entwurf eines Testsystems für J2EE

Mit dem Wissen um die Problematik von Transaktionssicherheit wird nun der Frage nachgegangen, wie es möglich ist Aussagen über die Transaktionssicherheit eines J2EE-Produktes zu machen (Frage 3 aus Kapitel 1.1), dies sogar auf die J2EE-Anwendungsentwicklung zu erweitern (Frage 5 aus Kapitel 1.1) sowie eine Vergleichsmetrik für J2EE-Produkte zu schaffen (Frage 4 aus Kapitel 1.1). Der erste Schritt hierzu ist eine Problemanalyse (Kapitel 6.1) in der untersucht werden soll, mit welchem System man Antworten zu diesen Fragen erhalten kann. Anschließend wird der Entwurf des Transaction Testsystems (TTS) vorgestellt (Kapitel 6.2), das Datenmodell (Kapitel 6.3) und die Konsistenzbedingungen (Kapitel 6.4). Zuletzt wird die Bewertungsmetrik definiert (Kapitel 6.5).

6.1 Problemanalyse

Dem Thema Transaktionssicherheit wurde in den vorherigen Kapitel ausführlich nachgegangen. Doch obwohl die Ergebnisse sehr vielversprechend sind, bleibt vieles eine Implementierungsangelegenheit. Hier schließt sich unmittelbar die Frage an, wie die Qualität der Implementierung sichergestellt werden kann. Darüber hinaus wird es doch noch etwas Zeit in Anspruch nehmen, bis die ersten J2EE-Produkte auf Basis der Isolate-API verfügbar und ausgereift sind. Dies wirft die Frage auf, mit welcher Basis ein Entscheidungsträger die Faktoren Kosten und Leistung eines J2EE-Produktes abschätzen und somit zu einer substanziellen Entscheidung kommen kann, welches J2EE-Produkt für sein Anwendungsszenario am besten geeignet ist. Des weiteren muss auch sichergestellt werden, dass die Anwendungskomponenten stabil und zuverlässig funktionieren, was durch den relativ komplexen, entkoppelten und verteilten Entwicklungsprozess von J2EE-Anwendungen recht schwierig ist.

Für die Untersuchung wann ein J2EE-Produkt für ein Anwendungsszenario einsetzbar ist wird eine formalere Definition benötigt. Hierbei besteht ein Anwendungsszenario aus einer Menge von Anwendungsfällen. Jeder Anwendungsfall definiert eine Reihe von Inputparametern, eine durchzuführende Funktionalität und das erwartete Resultat. Zusätzlich können Rahmenparameter, über die Menge der parallelen Transaktionen, Gesamtmengen von Transaktionen sowie die maximal geduldete Durchlaufzeit, definiert werden.

Ein System ist dann für ein Anwendungsszenario einsetzbar, wenn alle Anwendungsfälle unter Berücksichtigung der Rahmenparameter ein korrektes Ergebnis liefern, also für jeden Anwendungsfall die Inputparameter auf das erwartete Ergebnis abgebildet werden.

Es gibt prinzipiell drei Varianten, wie eine Aussage getroffen werden kann, ob ein System für ein Anwendungsszenario einsetzbar ist:

1. **Formale Verifikation des Quellcodes:** das Grundprinzip ist, die Methoden des Sourcecodes auf mathematisch handhabbare Konstrukte zurückzuführen. Mit Hilfe von Zusicherungen, über die Leistung einzelner Methoden, werden schrittweise alle Methoden, welche an einer Funktionalität beteiligt sind, betrachtet. Die Summe der Zusicherungen liefert schließlich das Ergebnis, ob die Inputparameter in das erwartete Ergebnis überführt werden können oder nicht [Whi00].
   Der Vorteil dieses Ansatzes ist es, dass es ein mathematisch korrektes Ergebnis liefert. Leider ist dieses Verfahren in der Praxis kaum anwendbar, da Kosten und Zeit für eine formale Verifikation eines Sourcecodes astronomisch sind (beispielsweise kann ein Experte ca. 1000 bis 2000 Zeilen Quellcode pro Jahr verifizieren [Kla03]). Hinzu kommt noch, dass der Kunde, welcher sich einen Eindruck über die Eignung des J2EE-Produktes für sein Anwendungsszenario verschaffen möchte, im Normalfall nicht über dessen Sourcecode verfügt. Damit scheitert die formale Verifikation schon oft an den Grundvoraussetzungen.
2. **Experten-Empfehlung:** eine Gruppe von Experten oder Einzelpersonen haben sich (durch Quellcode-Analyse oder Erfahrung) einen Eindruck über die Leistungsfähigkeit konkreter J2EE-Infrastrukturen gemacht und können daraus die Eignung für gewisse Anwendungsszenarien ableiten. Dies ist der in der Praxis gängigste Weg. An dieser Stelle sei dahingestellt, ob es sich um Experten des Kunden handelt oder um externe Experten und auf welche Weise das Expertenwissen erlangt wurde. Problematisch ist dieser Weg dahingehend, dass die „sogenannten“ Experten über einen profunden Erfahrungsschatz verfügen müssen, da es mit jeder Version eines J2EE-Produktes wieder Unterschiede gibt. Auch kann jeder Experte nur über einen gewissen Ausschnitt der am Markt verfügbaren J2EE-Produkte und *Resource Adapter* sprechen. Die Empfehlung ist deshalb weniger auf das bestmögliche Ergebnis ausgerichtet, sondern eher umgekehrt, auf ein dem Experten bekanntes Produkt und ob es die gestellten Anforderungen erfüllt. Hinzukommt, dass die Entscheidung des Experten subjektiv ist und in der Regel auf unvollständigem Wissen beruht. Darüber hinaus lässt sich damit nur die prinzipielle Eignung einer J2EE-Infrastruktur für ein Anwendungsszenario herleiten, nicht ausgeschlossen ist die falsche Anwendung.
3. **Evaluierung durch Test der J2EE-Infrastruktur gegen die Anwendungsszenarien:** die Anwendungsszenarien werden auf Testfälle abgebildet. Für die Anwendungsfälle werden Funktionalitäten realisiert und auf der J2EE-Infrastruktur installiert. Die Testfälle testen nun die realisierten Funktionalitäten, auf die korrekte Abbildung der Inputparameter auf die Ergebnismenge. Je nach Rahmenparameter müssen die Testfälle in einem komplexeren Rahmen, mit einer Vielzahl von nebenläufigen Transaktionen, ebenfalls korrekt sein. Dieser Weg wird in der Praxis hin- und wieder, zumindest auf eine durch Experten-

Empfehlung vorselektierte kleine Menge, von J2EE-Produkten angewendet, um aus einer Menge von zwei bis drei J2EE-Infrastrukturen die geeignete Plattform zu evaluieren. Problematisch ist, dass Projekte generell und Softwareprojekte im speziellen meistens unter starkem Zeit- und Budgetdruck stehen. Es ist deshalb äußerst selten, dass in einem Projekt die Zeit und das Budget investiert wird, tatsächlich praxisrelevante Anwendungsszenarien zu testen. Statt dessen wird häufig nur ein sogenannter *Proof of Concept* für die Schlüsselaspekte der angedachten Architektur durchgeführt, und selbst dies ist schon selten. Wenn die Entscheidung für eine bestimmte J2EE-Infrastruktur endlich gefallen ist und die Anwendungsentwicklung beginnt, werden wiederum häufig die ausführlichen Anwendungstests, zugunsten von neuen Funktionalitäten oder aggressiven Zeitplänen vernachlässigt (äußerst treffend in [DeMa97] nachzulesen). Eine Faustformel besagt, dass für die Realisierung eines Testfalls nochmals genauso viel Zeit veranschlagt werden muss, wie für die Entwicklung der Funktionalität selbst. Hinzukommt, dass für den Test von EJBs eine einfache Testklasse im Rahmen eines Modultests nicht ausreicht, da dies die Komplexität der Komposition von EJBs innerhalb einer Transaktion oder gar die Container-Interaktion ganz außer Acht lässt. Es ist also eine gewisse Infrastruktur notwendig, welche auch zunächst entwickelt werden muss. Dies ist oft nicht durch das Budget oder den Projektplan abgedeckt oder wird, um Verzüge innerhalb des Projektes zu kaschieren, stillschweigend aus dem Projektumfang definiert. Dies ist eine äußerst bedenkliche Situation, da, wie in [Kla03] dargestellt, die Kosten für die Fehlerbehebung drastisch ansteigen, je später sie entdeckt werden. Dies ist auch einer der Gründe weshalb J2EE-Projekte in der Vergangenheit so häufig gescheitert sind.

Zusammenfassend ist zu bemerken, dass der Ansatz der formalen Verifikation in der Praxis kaum anwendbar ist. Der Ansatz der Experten-Empfehlung ist zwar der Häufigste, jedoch aber auch der Fehlerträchtigste. Der dritte Ansatz des Tests von J2EE-Infrastrukturen und Anwendungen, welcher, abgesehen von der formalen Verifikation, noch die besten Ergebnisse liefern würde, kommt aus Zeit oder Budgetgründen meistens zu kurz. Wird der dritte Ansatz nicht mit dem notwendigen Ernst betrieben, liefert er leider genauso unvollständige Aussagen über eine J2EE-Infrastruktur wie der zweite Ansatz. Jedoch steckt in diesem Ansatz das Potential, bei richtiger Anwendung eine brauchbare Aussage über die Einsetzbarkeit eines J2EE-Produktes liefern zu können. Es wird deshalb im folgenden der dritte Ansatz einer genaueren Untersuchung unterzogen. Die Probleme, welche in der Praxis häufig entstehen, sind folgendermaßen zu kategorisieren:

1. **Fehlendes Wissen über J2EE:** es wird ein fundiertes Wissen über J2EE an sich benötigt, um eine J2EE-Anwendung und aussagefähige Testfälle erstellen zu können.
2. **Fehlende Kenntnis der einzelnen am Markt verfügbaren J2EE-Produkte:** es wird ein fundiertes Wissen über die jeweiligen J2EE-Implementierungen benötigt, um ein Testsys-

tem aufzusetzen und darauf aussagekräftige Tests durchführen zu können. Um eine gewisse Objektivität zu erlangen, müssen möglichst viele J2EE-Produkte getestet werden.

3. **Fehlendes Budget um Testsystem aufzusetzen:** um eine repräsentative Evaluierung durchführen zu können, müssen für eine Vielzahl von J2EE-Produkten separate Testsysteme aufgesetzt werden. Dies bedeutet die Bereitstellung von Hardware, die Installation des Betriebssystems, sowie die Besorgung von Lizenzen und das Setup der J2EE-Infrastruktur. Darauf müssen für die verschiedenen kritischen Anwendungsszenarien Testfälle und Funktionalitäten realisiert werden. Darüber hinaus muss es eine Test-Infrastruktur geben, welche die Tests multiplizieren kann, um Last und die parallele Verarbeitung zu testen. Dies ist ein immenser Kosten- und Entwicklungsaufwand für ein einzelnes Projekt.
4. **Fehlende Zeit:** häufig existieren aggressive Zeitpläne, da das Unternehmen in Konkurrenz mit internen oder externen Mitbewerbern steht und die schnelle Bereitstellung von Funktionalitäten über Wettbewerbsvorteile entscheidet. Ein langwieriger Ansatz des Tests von J2EE-Produkten kann deshalb oft nicht erfolgen. Auch der Test der entwickelten J2EE-Anwendung kommt oft zu kurz und wird oft nur funktional oder im Rahmen eines Modultests durchgeführt.

Die Loslösung der Entwicklung eines solchen Testsystems von dem tatsächlichen Bedarf der Entscheidungsfindung innerhalb eines Projektes, scheint der Schlüssel für eine Lösung der oben dargestellten Probleme zu sein. Bei Sammlung der Testergebnisse und deren Veröffentlichung müsste nur einmal in den Test einer J2EE-Infrastruktur investiert werden, während die Anwendungsfälle wiederverwendet werden könnten. Hieraus lassen sich die folgenden Rahmenanforderungen ableiten:

- Das Testsystem besteht aus einer Anzahl von Anwendungsszenarien (Funktionalitäten, Testfälle und Daten) dem Testframework, welches die Tests auf dem Testsystem durchführt und einem Metrik-Modul, welches die Tests auswertet.
- Anwendungsszenarien müssen flexibel hinzugefügt werden können. Die Transaktionseigenschaften und die Einbettung in eine Transaktion müssen einfach veränderbar sein. Variationen von Funktionalitäten in verschiedenen Transaktionssituationen müssen einfach testbar sind.
- Das Testsystem sollte sich strikt an die Minimalanforderungen der J2EE-Spezifikation halten. Damit ist gewährleistet, dass die Testfälle auf einem breiten Spektrum an J2EE-Produkten lauffähig sind.
- Die Metriken müssen ein einheitliches Bewertungssystem für alle J2EE-Produkte haben und pro Anwendungsszenario Aufschluss darüber geben, ob der Test erfolgreich war. Das Anwendungsszenario muss vermerken, mit welchen Rahmenbedingungen es durchgeführt wurde (Last, Isolation, Typ der nebenläufigen Transaktionen).

- Das Testsystem muss ein Datenmodell sowie Funktionalität für die Anwendungsszenarien beinhalten, aber es muss auch ermöglichen, leicht für andere Datenmodelle und Funktionalitätstests verwendet zu werden (z.B. im Rahmen der Anwendungsentwicklung eines Unternehmens).
- Zumindest die Umsetzungen der Anwendungsszenarien müssen öffentlich zugänglich sein. Nur so kann gewährleistet werden, dass zum einen Fehler in den Implementierungen entdeckt werden können und zum anderen Anwender angeregt werden, auch ihre Erweiterungen oder Verbesserungen der Anwendungsszenarien öffentlich zu machen. Ein geeignetes Mittel wäre die Freigabe des Quellcodes für die Anwendungsszenarien als Opensource [OS04].

6.2 Das Transaction Test System (TTS)

Aus den Anforderungen an das Testsystem wurde im Rahmen dieser Arbeit das *Transaction Testsystem (TTS)*, bestehend aus dem *TestController-System (TCS)*, dem *System under Test (SUT)* und einer Reihe von Test- und Fehlerfällen, entwickelt. Die detaillierten Ergebnisse der Anforderungsanalyse sowie des Entwurfs des TTS sind im Anhang B und C dargestellt.

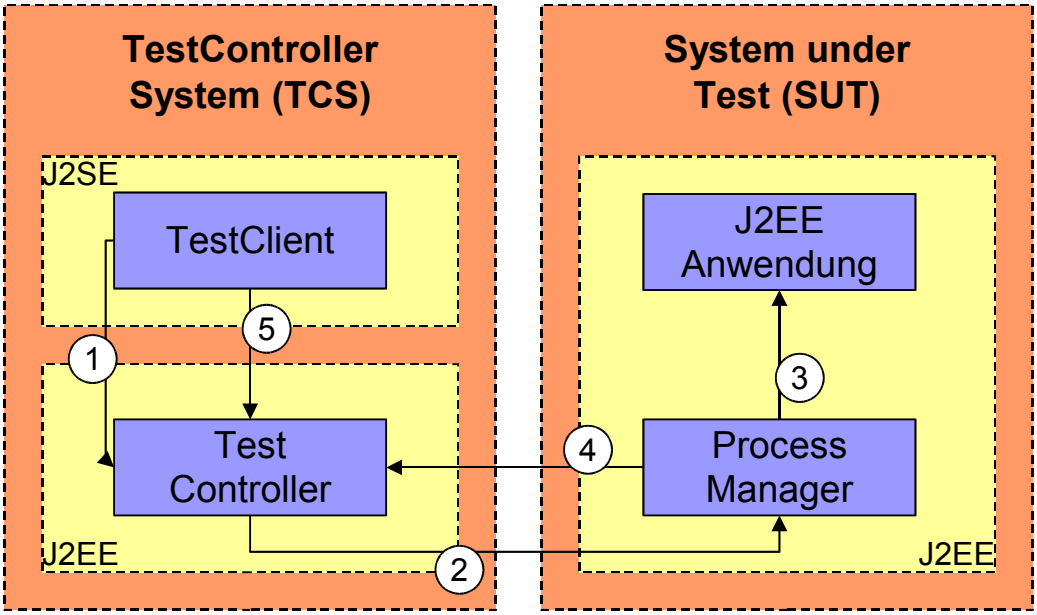

Abbildung 27 Architektur des Transaction Testsystem (TTS)

Das TCS beinhaltet den `TestClient`, welcher ein Tester benutzt um eine Testsuite, bestehend aus mehreren Testfällen, zu erstellen und dem `TestController` zu übergeben (Abbildung 27, 1). Der `TestController` ist ein Dienst innerhalb eines J2EE-Containers

und stellt Schnittstellen bereit, zum Ausführen und Interagieren mit einer Testsuite. Der `TestController` verwaltet die Testprozesse zu einem Testfall und übergibt diese zur Ausführung auf dem SUT dem `ProcessManager` (Abbildung 27, 2). Ebenfalls steuert er die Basislast gegen das SUT während der Ausführung eines Testprozesses.

Der `Process-Manager` ist die Integrationsschicht zwischen dem TTS und einer beliebigen zu testenden J2EE-Anwendung (Abbildung 27, 3). Während der Ausführung des Testprozesses kommuniziert der `ProcessManager` an bestimmten Synchronisationspunkten mit dem `TestController` (Abbildung 27, 4), um sich mit den parallel ausgeführten Testprozessen desselben Testfalls zu synchronisieren. Die Synchronisation wird vom `TestController` durchgeführt. Wenn ein Testprozess den für ihn vorgesehenen Synchronisationspunkt erreicht hat, wird der Methodenaufruf vom `TestController` intern blockiert, bis die restlichen Testprozesse ebenfalls ihre Synchronisationspunkte erreicht haben.

Wenn alle Testprozesse ihren Synchronisationspunkt erreicht haben, kehren die Testprozesse in einer definierten Reihenfolge zurück. Bei der Rückkehr der Testprozesse können Fehlerfälle im SUT ausgelöst werden. Bei Fehlern der Fehlerkategorie ERROR (z.B. `NullPointerException`), darf nur der fehlerauslösende Testprozess fehlschlagen und ein Rollback durchführen, alle anderen Testprozesse müssen erfolgreich sein. Bei der Fehlerkategorie FATAL (z.B. *Systemcrash*), laufen alle aktiven Testprozesse auf einen Fehler. Das System muss evtl. neu gestartet und auf dem *Resourcemanager* ein *Recovery* durchgeführt werden.

Nach der Ausführung eines Testprozesses, startet das TCS die Überprüfung der Korrektheit der Ergebnisse des Testprozesses sowie die Konsistenzprüfung der Anwendungsdatenbank. Wenn das SUT durch die Ausführung eines FATAL-Fehlers zeitweise nicht verfügbar ist, wird die Validierung solange verzögert. Zuletzt werden die Daten für die Testmetrik und die Vergleichsmetrik ausgewertet und in die Metrik-Datenbank des TCS geschrieben.

Der `TestClient` kann jederzeit beim TCS den Status und die Ergebnisse der übergebenen Testsuite abfragen (Abbildung 27, 5).

6.3 Datenmodell und Mengengerüst

Als Basis dient das in Abbildung 28 dargestellte TERPH-Datenmodell der *Terrific Housewares AG*, da es von der TPC auch speziell dazu herangezogen wurde, ein praxisrelevantes auftragsverarbeitendes System zu simulieren. Prinzipiell sind zwei Ansätze denkbar für die Realisierung der einzelnen *Warehouses*:

- Möglichkeit 1: die Daten aller Warehouses sind in einer zentralen Datenbasis abgelegt. Es finden nur Transaktionen über einen *Resourcemanager* statt.
- Möglichkeit 2: die Daten sind pro Warehouse in einer separaten Datenbasis abgelegt. Es finden also Transaktionen sowohl über einen *Resourcemanager* als auch über Mehrere statt.

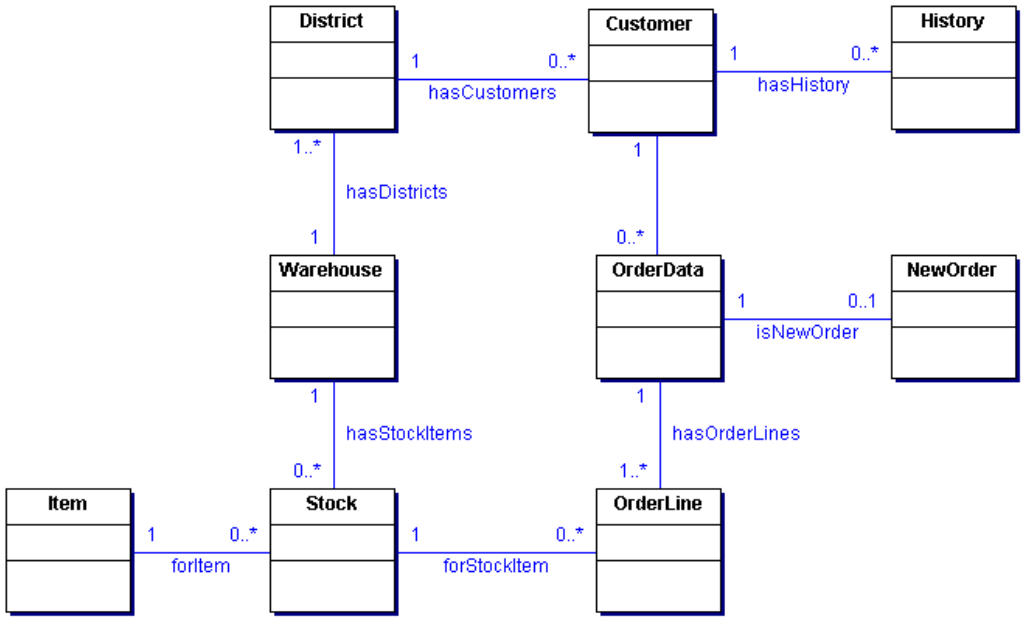

Abbildung 28 Datenmodell von TERPH

Für diese Arbeit genügt es wenn das TTS die Möglichkeit 2 vorsieht. Die konkreten Tests werden mit der Möglichkeit 1 durchgeführt. Die mengenmäßige Skalierung der Datenbasis erfolgt über die Anzahl der *Warehouses*. Da das hauptsächliche Interesse an der Transaktionssicherheit liegt, genügt eine Skalierung von zwei *Warehouses* (vgl. Tabelle 15).

Tabelle 15 Mengengerüst für die Anwendungsdaten

Entität	***Menge pro Entität***	***Gesamtmenge***
Warehouse	2	2
District	10 pro Warehouse	20
Customer	3000 pro District	60000
Items	100000	100000
Stock	100000 pro Warehouse	200000
Order	3000 pro District	60000
OrderLine	Random(5,15)[1] pro Order	600000

1 Random(x,y) ist eine Funktion auf Basis der `Random`-Klasse von Java. Die Funktion wird benutzt um eine zufallsbedingte Auswahl einer Zahl zwischen x und y zu simulieren.

6.4 Konsistenzkriterien

Das vorgestellte Datenmodell muss die folgenden Konsistenzkriterien erfüllen:

1. Die Datensätze von *District*, *OrderData* und *NewOrder* müssen der Bedingung genügen:

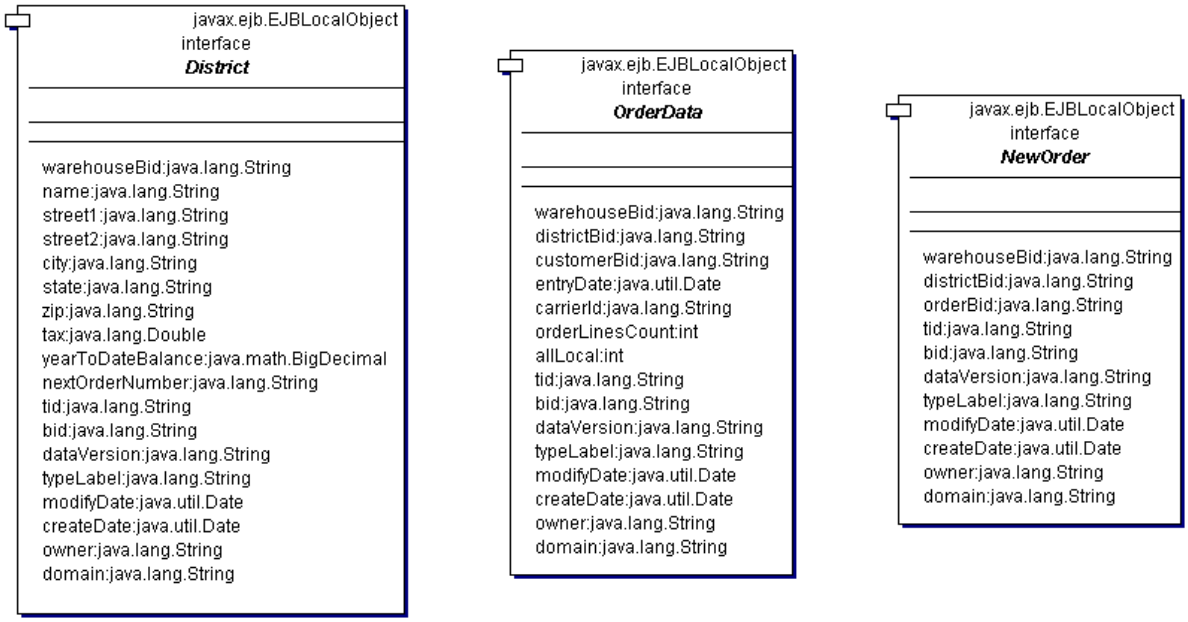

Abbildung 29 Entitäten zu District, OrderData und NewOrder

```
District::nextOrderNumber - 1 =
              max(OrderData::bid) = max(NewOrder::bid)
```

mit den Beziehungen:

```
District::warehouseBid =
OrderData::warehouseBid =
NewOrder::warehouseBid

District::bid =
OrderData::districtBid =
NewOrder::districtBid
```

Die Attribute `nextOrderNumber` und `bid` sind zwar im Datenmodell (Abbildung 29) als `String` modelliert, beinhalten jedoch nur numerische Werte.

2. Die *NewOrder*-Datensätze müssen der Bedingung genügen:

```
max(NewOrder::bid) - min(NewOrder::bid) + 1 =
     [Anzahl Datensätze von NewOrder für diesen District]
```

mit den Beziehungen:

```
NewOrder::warehouseBid =
NewOrder::warehouseBid

NewOrder::districtBid =
NewOrder::districtBid
```

3. Die Datensätze von *OrderData* und *OrderLine* (Abbildung 30) müssen der Bedingung genügen:

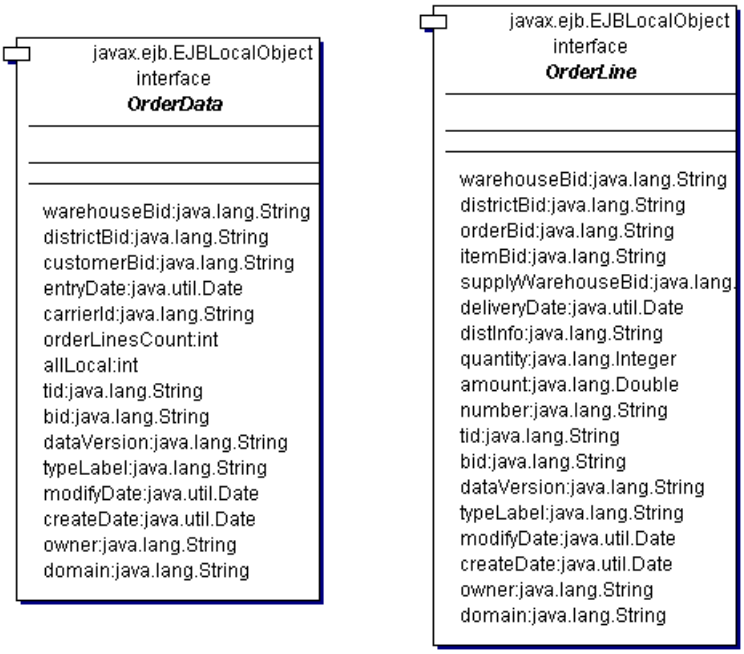

Abbildung 30 Entitäten für OrderData und OrderLine

```
sum(OrderData::orderLinesCount)
     = [Anzahl der Datensätze von OrderLine pro District]
```

mit den Beziehungen:

```
OrderData::warehouseBid =
OrderLine::warehouseBid

OrderData::districtBid =
OrderLine::districtBid
```

4. Für jeden Datensatz in *OrderData* gilt:

   `OrderData::carrierId` ist auf den Wert „null“ gesetzt, genau dann, wenn der einen korrespondierenden *NewOrder* Datensatz hat mit den Beziehungen:

   ```
   OrderData::warehouseBid =
   NewOrder::warehouseBid

   OrderData::districtBid =
   NewOrder::districtBid

   OrderData::bid =
   NewOrder::orderBid
   ```

5. Für jeden Datensatz in *OrderData* gilt:

   das Feld `OrderData::orderLinesCount` muss gleich der Anzahl der Datensätze von *OrderLine* sein, welche die Beziehung haben:

   ```
   OrderData::warehouseBid =
   OrderLine::warehouseBid

   OrderData::districtBid =
   OrderLine::districtBid

   OrderData::bid =
   OrderLine::orderBid
   ```

6. Für jeden *OrderLine* Datensatz gilt:

   das Feld `OrderLine::deliveryDate` muss auf „null“ gesetzt werden, genau dann, wenn ein korrespondierender *OrderData* Datensatz existiert mit dem Wert „null“ für `OrderData::carrierId`. *OrderData* und *OrderLine* müssen die folgenden Beziehungen erfüllen:

   ```
   OrderData::warehouseBid =
   OrderLine::warehouseBid

   OrderData::districtBid =
   OrderLine::districtBid

   OrderData::bid =
   OrderLine::orderBid
   ```

6.5 Definition der Bewertungsmetrik

Die Testmetrik beschreibt die Messwerte, welche bei einem Test ermittelt werden und welche Aufschluss über den Testerfolg liefern. Für die Bewertungsmetrik der Tests durch das TTS wird zweistufig vorgegangen. Eine Detailmetrik gibt jeden Testfall auf das genaueste wieder (anhand dreißig Werten aus den Bereichen technische Fehler, fachliche Fehler, Ausführungsdauer, Konsistenz und Anzahl modifizierter Datensätzen). Eine konsolidierte Vergleichsmetrik aus nur fünf Werten eignet sich für den Vergleich zwischen J2EE-Produkten oder unterschiedlichen Produktkonfigurationen. Die Definition der Detailmetrik ist in Tabelle 16 dargestellt.

Tabelle 16 Definition der Detailmetrik

ID	***Feld***	***Beschreibung***
1	TestSuiteBid	Id der `TestSuite`, welche der `TestClient` als Identifizierung für die `TestSuite`-Konfiguration festgelegt hat.
2	TestCaseBid	Id des `TestCase`, welche der `TestClient` als Identifizierung für die `TestCase`-Konfiguration festgelegt hat.
3	TestConfigurationBid	Id der `TestConfiguration`, welche der `TestClient` als Identifizierung für die `TestConfiguration` festgelegt hat.
4	TestRun-Handle	`Handle` der `TestRun`-Instanz, sodass die Bewertung einem konkreten Testprozess zugewiesen werden kann.
5	RequestCase	Klassenname des `RequestCase`.
6	EvaluationCase	Klassenname des `EvaluationCase`.
7	ErrorCase	Klassenname des `ErrorCase`.
8	RunInfinite	Bezeichnet ob die `TestRun`-Instanz endlos ausgeführt wird (z.B. Basislast-TestRun). Werte: true/false.
9	LoopCount	Anzahl der Schleifen-Durchläufe.
10	StateOfRequest	Status der Ausführung des `RequestCases`.
11	StateOfEvaluation	Status der Ausführung des `EvaluationCases`.
12	CountOf RequestExecptions	Anzahl der aufgetretenen `Exceptions` während des Requests (dürfte nur auftreten bei `TestRun` mit `ErrorCase`, oder wenn eine `TestConfiguration` des `TestCases` einen Fatal-Fehler verursacht).

ID	*Feld*	*Beschreibung*
13	CountOf EvaluationErrors	Anzahl der technischen Fehler bei der Ausführung des `EvaluationCases`.
14	CountOf EvaluationFailures	Anzahl der fachlichen Fehler (Verstöße gegen die erwarteten Werte) bei der Ausführung des `EvaluationCases`.
15	TestTime (Begin / End)	Angabe der Zeitstempel für die Testdurchführung. (Zeitpunkt des Testbeginns / Zeitpunkt des Testendes).
16	DurationOfTest	Dauer der Ausführung des Tests (TCS).
17	ProcessTime (Begin / End)	Angabe der Zeitstempel für die Testprozess-Durchführung. (Start-Zeitpunkt des Prozesses / Ende-Zeitpunkt des Prozesses).
18	DurationOfProcess	Dauer der Ausführung des Testprozesses (SUT).
19	Duration OfBusinesslogic	Dauer der Ausführung der Businesslogik.
20	Interrupts OfBusinesslogic	Anzahl der Unterbrechungen der Businesslogik.
21	TestForWait	Angabe ob auf Wait (Sperre) getestet werden sollte. Werte: true/false.
20	ShallWaitOccure	Angabe ob ein Wait (Sperre) auftreten soll (Annahme).
21	WaitOccured	Angabe ob ein Wait (Sperre) aufgetreten ist (Ergebnis).
22	DurationOfWait	Dauer des Waits (Sperre).
23	ConsistencyOk	Angabe ob die Durchführung der Konsistenztests nach der Ausführung erfolgreich verlaufen ist. Werte: true/false.
24	CountOfEntities BeforeTest	Anzahl der Entitäten der zu testenden J2EE-Anwendung, vor der Ausführung des Tests (Information über `SnapShotStatistic`).
25	CountOfEntities AfterTest	Anzahl der Entitäten der zu testenden J2EE-Anwendung nach der Ausführung des Tests (Information über `SnapShotStatistic`).
26	CountOf CreatedEntities	Anzahl der durch den Testprozess erzeugten Entitäten der zu testenden J2EE-Anwendung (Information über `SnapShotStatistic`).
27	CountOf ModifiedEntities	Anzahl der durch den Testprozess modifizierten Entitäten der zu testenden J2EE-Anwendung (Information über `SnapShotStatistic`).

Mittels dieser Detailmetrik ist eine genaue Analyse und Bewertung des ausgeführten Testfalls möglich. Sie eignet sich gut für eine Tiefenanalyse der J2EE-Anwendung bzw. der J2EE-Produktkonfiguration. Als Vergleichsmetrik für J2EE-Produkte ist sie jedoch zu ausführlich. Die Vergleichsmetrik ist in Tabelle 17 dargestellt.

Tabelle 17 Definition der Vergleichsmetrik

ID	***Feld***	***Beschreibung***
1	Test	Zusammenführung der Id aus der `TestSuite` (1) und `TestCase` (2). Format: <1>-<2>.
2	Ergebnis	Zusammenführung und Kumulierung der Felder StateOfEvaluation (11) und ConsistencyOk (23). Für die Bestimmung des kumulierten Wertes wird das schlechteste Ergebnis einer `TestConfiguration` des `TestCases` herangezogen. Format: <11>/<23>.
3	Ergebnisdetails	Zusammenführung und Kumulierung der Felder CountOfRequestExecptions (12), CountOfEvaluationErrors (13) und CountOfEvaluationFailures (14). Für die Kumulierung wird eine einfach Aufsummierung der Werte pro Feld durchgeführt. Format: <12>/<13>/<14>.
4	Durchschnittliche Dauer (mit Basislast)	Ausgabe des Wertes für DurationOfBusinesslogic (18) geteilt durch LoopCount (9) kumuliert auf Ebene des `TestCases`. Kumulation erfolgt durch einfache Aufsummierung der Werte pro Feld (ohne Datensätze mit WaitOccured=true).
5	Durchschnittliche Dauer (ohne Basislast)	Für alle TestRun-Instanzen welche RunInfinite != true haben werden die Felder DurationOfBusinesslogic (18) geteilt durch LoopCount (9) kumuliert auf Ebene des `TestCases`. Kumulation erfolgt durch einfache Aufsummierung der Werte pro Feld (ohne Datensätze mit WaitOccured=true).

Mittels dieser Vergleichsmetrik lässt sich eine Ergebnismatrix aufstellen, welche für jedes getestete J2EE-Produkt und jeden durchgeführten TestCase die Werte der Vergleichsmetrik beinhaltet. Eine Summierung der Ergebnisse aller TestCases für ein J2EE-Produkt als abschließende Zeile erleichtert den Überblick.

7 Testdurchführung und Ergebnisse

Um auf die Frage eine Antwort zu finden, wie es um die Gewährleistung der Transaktionssicherheit bei J2EE-Systemen steht, wurden Tests auf Basis des Transaction Testsystem (TTS) durchgeführt. Darüber hinaus stand die Eignung des TTS sowohl für die Auswahl einer J2EE-Infrastruktur anhand von Anwendungsszenarien als auch die Sicherstellung von stabilen und zuverlässigen Anwendungskomponenten auf dem Prüfstand.

7.1 Testdurchführung

Im folgenden wird der Aufbau des Testsystems sowie die verwendeten Testszenarien und Rahmenbedingungen erläutert.

7.1.1 Testaufbau

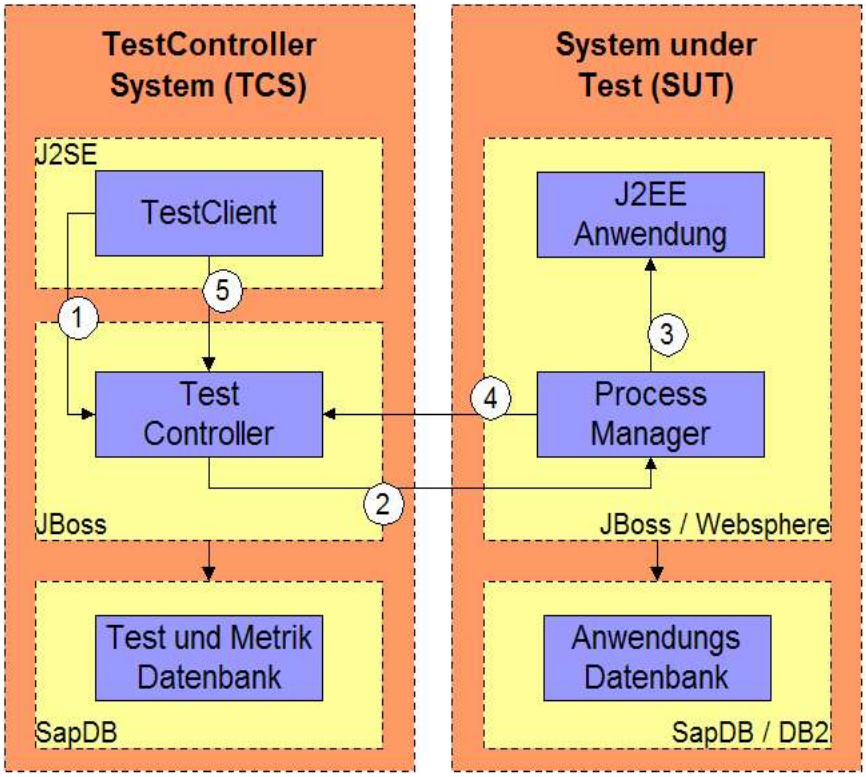

Abbildung 31 Testaufbau

Die Tests wurden auf zwei verschiedenen Konfigurationen von J2EE-Infrastruktur-Komponenten für das SUT durchgeführt:

Testsetup 1, bestehend aus einer kompletten Opensource Implementierung mit dem JBoss Applicationserver 3.2.3 und der Datenbank MaxDB 7.4.3.30.

Testsetup 2, bestehend aus einer kommerziellen J2EE-Implementierung mit dem IBM Websphere Applicationserver 5.1 und der Datenbank DB2 8.1, ebenfalls von IBM.

Die JVM (IBM), das Betriebssystem (Windows 2000 Advanced Server) und die Hardware wurden über die gesamte Testdurchführung als statisch betrachtet, um eine Vergleichbarkeit der Performanzmessungen sicherzustellen.

Der `TestController` des TCS wurde auf einer JBoss-Infrastruktur betrieben. Laufzeitdaten der Tests sowie die Metrikdaten wurden in einer MaxDB-Datenbank gespeichert. Der `TestClient` stellte eine separate Java-Anwendung dar. Die Kommunikation zwischen `TestClient` und `TestController` sowie `ProcessManager` und `TestController` (Abbildung 31 , Punkt 1, 5 und 4) fand immer über RMI statt. Die Kommunikation zwischen `TestController` und `ProcessManager` geschah bei Testsetup 1 (JBoss) ebenfalls über RMI, bei Testsetup 2 (Websphere) jedoch über das IIOP der IBM Orb-Implementierung. Im TCS wird ein optimistisches Synchronisationsverfahren für den Zugriff auf die Daten der Test- und Metrikdatenbank verwendet. In Anlehnung an die TPC-C-Spezifikation und an produktive Systeme, wird für den Zugriff auf die Daten der Anwendungsdatenbank im SUT ein pessimistisches Sperrverfahren eingesetzt.

7.1.2 Testablauf

Die Testdurchläufe werden alle nach demselben Verfahren durchgeführt (Abbildung 31). Der `TestClient` erstellt die relevanten `TestSuite`-Objekte und übergibt sie zur Abarbeitung dem `TestController` des TCS. Dieser startet die Testprozesse, übergibt sie an das SUT zur Ausführung und koordiniert die Abarbeitung. Nach der Ausführung eines Testprozesses startet das TCS die Überprüfung der Korrektheit der Ergebnisse sowie die Konsistenzprüfung der Anwendungsdatenbank. Zuletzt werden die Daten für die Detailmetrik und die Vergleichsmetrik ausgewertet und in die Metrik-Datenbank geschrieben.

Bei der Testdurchführung von Fehlerfällen aus der Kategorie FATAL wurde im TTS berücksichtigt, dass das SUT nach Ausführung dieses Fehlerfalls möglicherweise nicht mehr ansprechbar ist. Der `ProcessManager` im SUT verfügt über eine Methode, die eine Prüffunktionalität ausführt, welche auch Datenbankzugriff miteinschließt. Das TCS bedient sich dieser Funktionalität, um die Validierung der einzelnen Testprozesse so lange zu verzögern,

bis die Prüfmethode das richtige Ergebnis zurückliefert. Erst dann werden die Testprozesse auf korrekte Ausführung und auf die Datenbankkonsistenz, nach erfolgter Ausführung des Testprozesses, überprüft.

7.1.3 Verwendete Testszenarien und Fehlerfälle

Für die Durchführung der Tests wurden die folgenden Testfälle (detailliert in Anhang B.7) auf Basis der Anforderungsanalyse betrachtet:

- Atomarität.
- Atomarität bei *Nonfatal*-Fehlern (Kategorie ERROR, vgl. Tabelle 18).
- Atomarität bei *Fatal*-Fehlern (Kategorie FATAL, vgl. Tabelle 19).
- Isolation von Schreib / Schreib-Konflikten.
- Isolation von Schreib / Schreib-Konflikten bei *Nonfatal*-Fehlern (Kategorie ERROR, vgl. Tabelle 18).
- Isolation von Schreib / Schreib-Konflikten bei *Fatal*-Fehlern (Kategorie FATAL, vgl. Tabelle 19).

Tabelle 18 Bei der Testdurchführung berücksichtigte Fehler der Kategorie ERROR

Fehlerfall-Id	***Beschreibung des Fehlerfalls***
FinderForPrimaryKey HasNoResult	Suche nach einem Primärschlüssel lieferte keine Entität zurück.
FinderForCompoundKey-HasNoResult	Suche nach einem zusammengesetzten Fremdschlüssel lieferte keine Entität zurück.
DuplicateRecordsFound	Suche nach einem eindeutigen Schlüssel (z.B. Primärschlüssel oder eindeutiger Fremdschlüssel) lieferte mehrere Entitäten zurück.
RuntimeExceptionForData	Bei der Auswertung der Daten ist ein Laufzeitfehler aufgetreten (z.B. `NullPointerException`).

Tabelle 19 Bei der Testdurchführung berücksichtigte Fehler der Kategorie FATAL

Fehlerfall-Id	***Beschreibung des Fehlerfalls***
DatabaseNotAvailable	Die Verbindung zur Datenbank wurde beendet.
RuntimeException-ForMemory	Eine Transaktion beansprucht den gesamten Heap der JVM für sich (z.B. durch eine rekursive Endlosschleife). Es kommt zu einem *OutOfMemory*-Fehler.
RuntimeException ForContainer	Während der Ausführung einer Transaktion wird der Container abrupt beendet.
ShutdownOfContainer	Während der Ausführung einer Transaktion wird der Container durch ein *Shutdown*-Kommando kontrolliert beendet.
RuntimeException ForMachine	Während der Ausführung einer Transaktion wird die Maschine anormal beendet (Stromausfall, Hardwaredefekt).

7.1.4 Verwendete Businesslogik

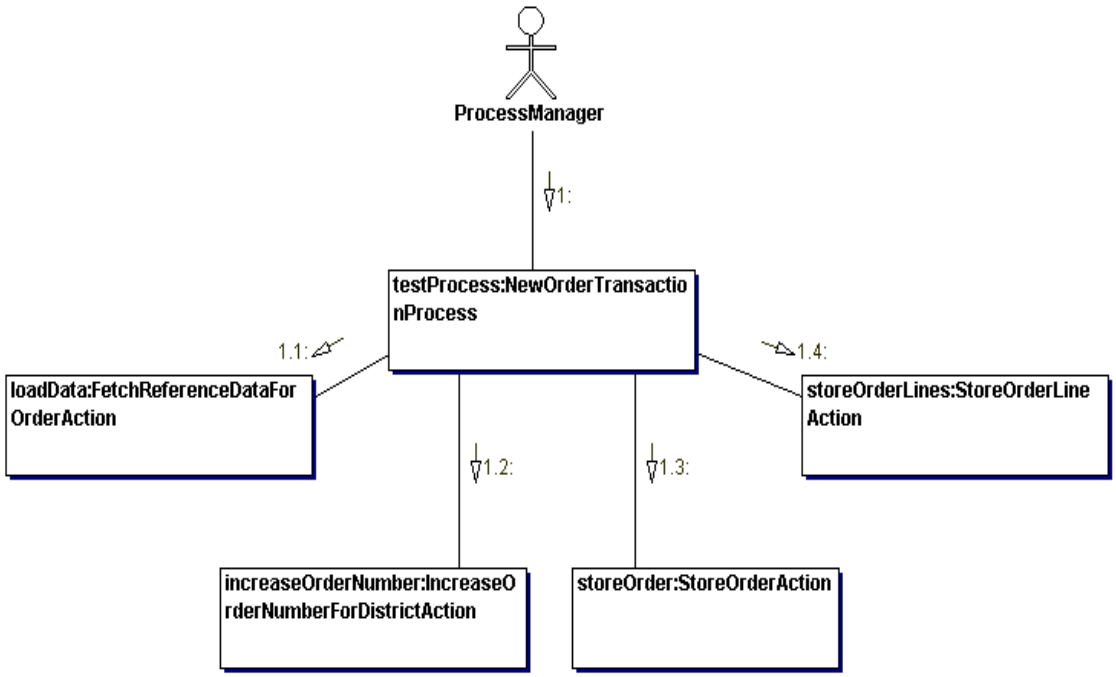

Abbildung 32 Kollaborationsdiagramm für den NewOrderTransaction-Prozess (Auftragsanlage)

Als Businesslogik für die Durchführung der Testfälle wurde die Auftragsanlage (*NewOrderTransaction*-Prozess, vgl. Abbildung 32) eingesetzt, welche schon während der Untersuchung der J2EE-Plattform in Kapitel 3 und der J2EE-Produkte in Kapitel 4 verwendetet

wurde. Der *NewOrderTransaction*-Prozess besteht aus den folgenden Aktionen (*Actions*), welche alle innerhalb einer Transaktion ausgeführt werden:

- **FetchReferenceDataForOrderAction:** Referenzdaten, wie die betroffene *Customer*-Entität sowie der *District*, werden auf der Datenbank anhand ihres fachlichen Schlüssels geladen. Die Referenzen werden im `Context`-Objekt für den *NewOrderTransaction*-Prozess abgelegt.
- **IncreaseOrderNumberForDistrictAction:** die Auftragsnummer im *District* des `Context`-Objektes wird als zu benutzende Auftragsnummer in den `Context` gesetzt. Die Auftragsnummer des *District* wird inkrementiert.
- **StoreOrderAction:** für das vom TCS übergebene Auftragskopf-Datenobjekt (`OrderDataBO`) wird ein neues `OrderData`-EntityBean erzeugt.
- **StoreOrderLineAction:** für jedes Auftragszeilen-Datenobjekt (`OrderLineBO`), welches vom TCS übergeben wurde, wird ein neues `OrderLine`-EntityBean erzeugt.

7.1.5 Testkategorien der Rahmenbedingungen

Um Informationen über das Verhalten der J2EE-Infrastrukturen unter steigender Last und verschiedenen Konfigurationsoptionen zu erhalten, wurden alle Testszenarien wiederholt ausgeführt, unter Verwendung von unterschiedlichen Rahmenparametern. Die Rahmenparameter wurden in den folgenden Testkategorien zusammengefasst:

1. **No Baseload with check with error:** alle Testszenarien ohne Fehler und mit Fehler aus der Kategorie ERROR werden verwendet. Die Tests werden ohne Basislast durchgeführt. Jede Transaktion des Testfalls wird auf korrekte Ausführung und Konsistenz der Datenbank überprüft.
2. **Baseload 10 with check with error:** alle Testszenarien ohne Fehler und mit Fehler aus der Kategorie ERROR werden verwendet. Die Tests werden mit einer Basislast von zehn parallelen Transaktionen (*NewOrderTransaction*-Prozess) durchgeführt. Jede Transaktion des Testfalls wird auf korrekte Ausführung und Konsistenz der Datenbank überprüft.
3. **Baseload 10 without check with error:** alle Testszenarien ohne Fehler und mit Fehler aus der Kategorie ERROR werden verwendet. Die Tests werden mit einer Basislast von zehn parallelen Transaktionen (*NewOrderTransaction*-Prozess) durchgeführt. Jede Transaktion des Testfalls wird auf korrekte Ausführung und Konsistenz der Datenbank überprüft, mit Ausnahme der Basislast-Transaktionen.
4. **Baseload 100 without check with error:** alle Testszenarien ohne Fehler und mit Fehler aus der Kategorie ERROR werden verwendet. Die Tests werden mit einer Basislast von hundert parallelen Transaktionen (*NewOrderTransaction*-Prozess) durchgeführt. Jede

Transaktion des Testfalls wird auf korrekte Ausführung und Konsistenz der Datenbank überprüft, mit Ausnahme der Basislast-Transaktionen.

5. **Baseload 10 without check with fatal:** alle Testszenarien ohne Fehler und mit Fehler aus der Kategorie FATAL werden verwendet. Die Tests werden mit einer Basislast von zehn parallelen Transaktionen (*NewOrderTransaction*-Prozess) durchgeführt. Jede Transaktion des Testfalls wird auf korrekte Ausführung und Konsistenz der Datenbank überprüft, mit Ausnahme der Basislast-Transaktionen.

7.1.6 Problembericht

Bei Websphere traten nach der Durchführung von FATAL-Testfällen, die einen Neustart des Websphere-Prozesses erforderten, Probleme bei der Wiederherstellung der Verbindung (Reconnect) vom TCS zum neu gestarteten SUT auf. Der Orb-Implementierung der IBM JVM war es nicht möglich, eine erneute Verbindung an den Orb des SUT herzustellen (Fehler in der `connect`-Methode). Es wurde darauf geachtet, dass keine Nutzung von fehlerhaften bzw. nicht mehr verfügbaren Referenzen im TTS vorkommen. Dazu wurde das TCS so realisiert, dass in einem solchen Fehlerfall nochmals ein neuer `InitialContext` erzeugt und die Referenz auf das `EJBHome` und `EJBObject` neu aus dem Namensdienst des SUT geholt wird. Doch trotz dieses Verfahrens trat im Falle von Websphere der beschriebene Fehler auf, welcher erst durch einem Neustart des TCS-Prozesses umgangen werden konnte. Danach konnte wieder eine neue Verbindung zum SUT aufgebaut werden.

Dies komplizierte die Auswertung der Testszenarien mit Fehlern aus der Kategorie FATAL, da für die Auswertung transiente Informationen im TCS gehalten wurden, welche nach dem Neustart des TCS-Prozesses nicht mehr zur Verfügung standen. Der TCS-Implementierung wurde daraufhin eine *Restart*-Funktionalität hinzugefügt, welche den transienten Zustand des TCS als serialisiertes Java-Objekt auf das Dateisystem schreibt und beim Neustart dieses Java-Objekt zur Initialisierung verwendet. Dadurch wurde es möglich, die FATAL-Fehlerfälle auch auf Websphere durchzuführen und auszuwerten.

Da JBoss nicht IIOP sondern RMI zur Kommunikation benutzt, war das Testsetup mit JBoss als SUT-Infrastruktur nicht von diesem Problem befallen. Die Wiederherstellung der Verbindung funktionierte im Falle von JBoss in allen Fehlerfällen.

7.2 Testergebnisse

Die Auswertung und Diskussion der Testergebnisse wurde in zwei Bereiche aufgeteilt:

1. Ergebnisse über die J2EE-Infrastruktur, welche mit der Annahme einer funktionsfähigen und zuverlässigen Anwendung im Sinne der Transaktionssicherheits-Problematiken arbeitet (Kapitel 7.2.1).
2. Ergebnisse der Anwendungstests, welche Aussagen über die Eignung des Testframeworks für die Gewährleistung von stabilen und zuverlässigen Server-Anwendungen liefern, v.a. im Sinne der Transaktionssicherheit (Kapitel 7.2.2).

7.2.1 J2EE Infrastruktur

7.2.1.1 Ergebnisse zur Transaktionssicherheit

Die Testergebnisse zur Transaktionssicherheit sind in Abbildung 33 grafisch aufbereitet. Auf der y-Achse ist die Anzahl der durchgeführten Testfälle aufgetragen. Die x-Achse ist in fünf Bereiche aufgeteilt, die den Testkategorien aus Kapitel 7.1.5 entsprechen. Jeder Bereich ist mit der Zahl der enthaltenen Testfälle versehen (Abbildung 33, *Total of executed Testcases*) und in vier Säulen mit folgender Bedeutung unterteilt:

1. Summe, der für diese Testkategorie erfolgreich durchgeführten Testszenarien für Websphere (Abbildung 33, *Testcases executed in Websphere with Success*).
2. Summe, der für diese Testkategorie erfolgreich durchgeführten Testszenarien für JBoss (Abbildung 33, *Testcases executed in JBoss with Success*).
3. Summe, der für diese Testkategorie nicht erfolgreich durchgeführten Testszenarien für Websphere (Abbildung 33, *Testcases executed in Websphere with Failures*).
4. Summe, der für diese Testkategorie nicht erfolgreich durchgeführten Testszenarien für JBoss (Abbildung 33, *Testcases executed in JBoss with Failures*).

JBoss und Websphere erwiesen sich beide, im Rahmen der durchgeführten Testfälle, als stabil und für die Transaktionsverarbeitung geeignet. Bei allen Testkategorien, bis auf Kategorie (2), wurden alle Testszenarien mit dem erwarteten Ergebnis durchgeführt sowie mit einem positiven Konsistenztest der Datenbasis abgeschlossen. Die Testkategorie (2) zeigt das Verhalten der Systeme bei zu vielen blockierenden Sperren. Diese wurden durch die Validierung des Testergebnisses und der Durchführung des Konsistenztests sowohl der Transaktionen der Testfälle als auch der Basislast-Transaktionen ausgelöst. Das Endergebnis gegenüber einem Anwender stellt sich gleich dar: kein Testfall ist erfolgreich durchgeführt worden in dieser Testkategorie. Websphere (mit DB2) verfügt jedoch über einen besseren und stabi-

leren Locking-Algorithmus als JBoss (mit MaxDB), da Websphere mehr Testfälle ausführt, bevor das System komplett sperrt und der Testlauf abgebrochen wird. Die Testkategorie (3) ist identisch mit (2), jedoch werden nur noch die Transaktionen der Testfälle der Validierungs- und Konsistenzprüfung unterzogen, nicht mehr die Basislast-Transaktionen.

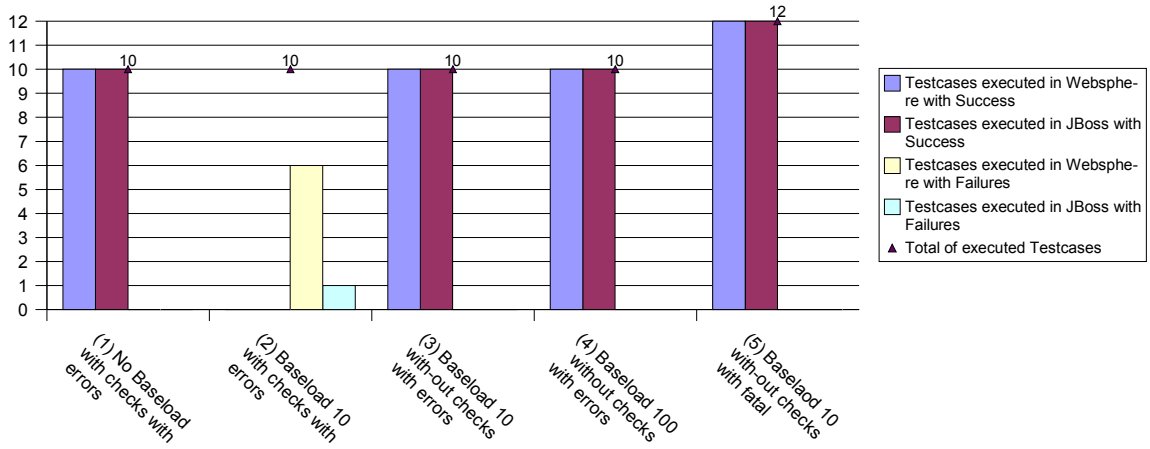

Abbildung 33 Testergebnisse zur Transaktionssicherheit. Dargestellt ist die Anzahl der durchgeführten Testfälle (y-Achse) pro Testkategorie (x-Achse).

Die Bestandteile der Testkategorien sind: No Baseload = ohne Basislast, Baseload 10(0) = Basislast von zehn (hundert) parallelen Transaktionen, with checks = Überprüfung aller Transaktionen, without checks = ohne Überprüfung der Basislast-Transaktionen, with errors = Fehler der Kategorie ERROR, with fatal = nur Fehler der Kategorie FATAL.

Aus den Problemen bei Testkategorie (2) sowie dem Erfolg von (3) und (4) lässt sich schlussfolgern, dass die Validierungslogik eine enorme Anzahl von Datenbank-Ressourcen durch Sperren blockiert, obwohl nur Daten gelesen wurden. Bei genauerer Betrachtung wird recht schnell klar, dass die Ausführung der Validierungslogik über dieselbe *Datasource*-Definition wie die CMP-Zugriffe diese Probleme verursacht haben. Sowohl der Applicationserver als auch die Datenbank haben die CMP-Transaktionen sowie die *Readonly*-Validierungs-Transaktionen gleich behandelt, als würden Daten verändert werden. Dadurch wurden exklusive Sperren für die Daten gesetzt und vielfach auf denselben Ressourcen, sodass der Durchsatz der Abarbeitung der Transaktionen drastisch einbrach, da die Transaktionen sequentiell ausgeführt werden mussten. Oft stellte sich auch eine *Deadlock*-Situation ein, sodass ganze Testfälle sich gegenseitig blockierten und entweder durch *Deadlock*-Erkennung oder durch die Überschreitung der Transaktionszeit (*Timeout*) rückgängig gemacht wurden.

Als Folgerung aus dieser Problematik lässt sich ziehen, dass Leseoperationen über *Datasource*-Definitionen ausgeführt werden sollten, welche als *Readonly*-Verbindungen konfiguriert sind und idealerweise mit einem geringeren Isolationsgrad auskommen (*Read Commited* anstelle von *Repeatable Read*). So kann sowohl der Applicationserver Datenbankzugriffe optimieren (bei *Readonly*-Transaktionen ist beispielsweise kein Aufruf von `ejbStore` not-

wendig) als auch die Datenbank das Setzen von Sperren effizienter handhaben (weniger Sperren und für die *Readonly* Transaktionen nur *Readonly*-Sperren). Dies führt zu einem erheblichen Anstieg der Parallelität und damit des Gesamtdurchsatzes an Transaktionen, welche das J2EE-System in einem bestimmten Zeitfenster durchführen kann.

7.2.1.2 Ergebnisse zur Performanz

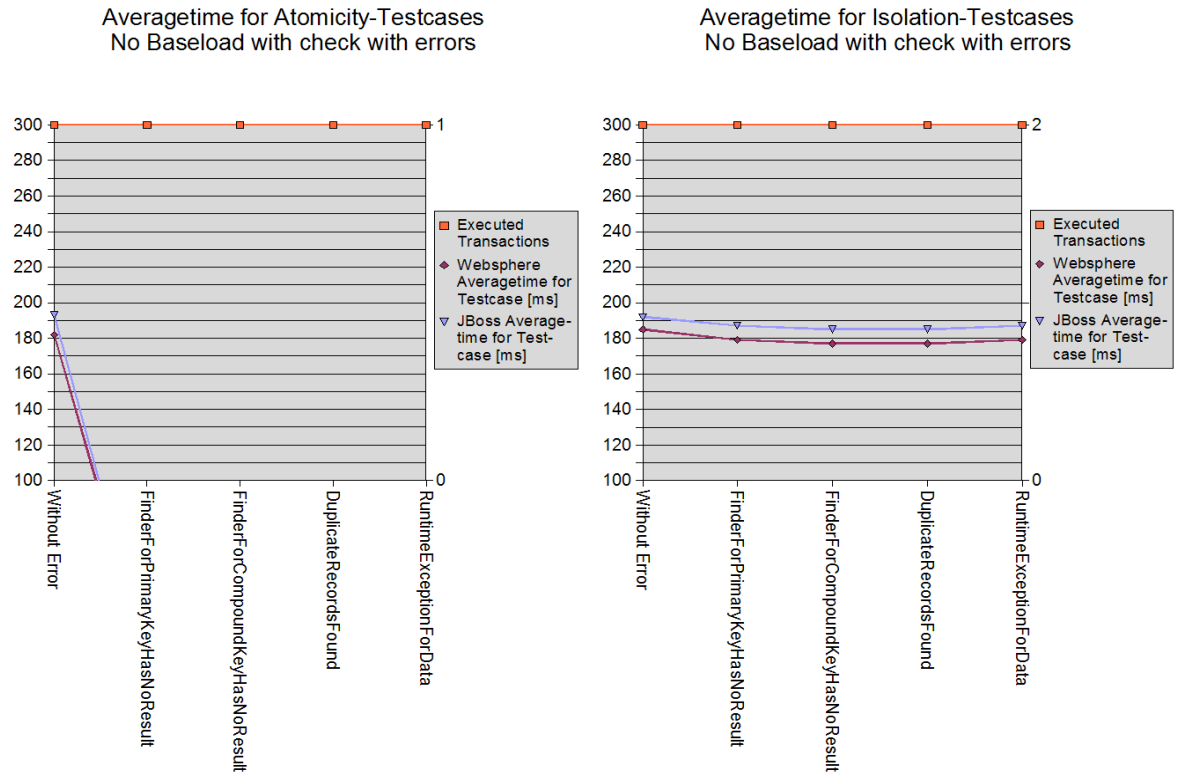

Abbildung 34 Performanz-Testergebnis für die Atomaritäts- und Isolationstestfälle der Testkategorie (1).

Dargestellt ist die durchschnittliche Ausführungsdauer (Averagetime) in Millisekunden (linke y-Achse) pro Testfall (x-Achse) sowie die Anzahl der pro Testfall durchgeführten Transaktionen (rechte y-Achse). Das linke Diagramm gibt die Ergebnisse für die Atomaritäts-Testfälle wieder, während das rechte Diagramm die Isolations-Testfälle beinhaltet. Die Testfälle pro Diagramm sind: Without Error = Ohne Fehlerfall, FinderForPrimaryKeyHasNoResult = Suche nach einem Primärschlüssel lieferte keine Entität zurück, FinderForCompoundKeyHasNoResult = Suche nach einem zusammengesetzten Fremdschlüssel lieferte keine Entität zurück, DuplicateRecordsFound = Suche nach einem eindeutigen Schlüssel lieferte mehrere Entitäten zurück, RuntimeExceptionForData = Bei der Auswertung der Daten ist ein Laufzeitfehler aufgetreten (z.B. NullPointerException).

Die Testergebnisse zur Performanz der J2EE-Infrastrukturen sind in Abbildung 34 für Testkategorie (1), Abbildung 35 für Testkategorie (3) und Abbildung 36 für Testkategorie (4) aufbereitet. Jede Abbildung beinhaltet zwei Diagramme. Das linke Diagramm gibt die Ergebnisse für die Atomaritäts-Testfälle wieder, während das rechte Diagramm die Isolations-Test-

fälle beinhaltet. Auf der x-Achse der Diagramme sind die Fehlerfälle (Kapitel 7.1.3) dieser Testkategorie aufgetragen. Die y-Achse links zeigt die durchschnittliche Zeitdauer (*Averagetime*) in Millisekunden, für die Ausführung der einzelnen Testfälle abzüglich der Zeit, welche als Koordinationsaufwand für die Ausführung von Testpunkten und die damit verbundene Verzögerung durch das TCS verbraucht wurde. Die Anzahl, der pro Testfall durchgeführten Transaktionen (Abbildung 34, Abbildung 35, Abbildung 36 - *Executed Transactions*), ist ebenfalls pro Diagramm aufgelistet und referenziert die alternative y-Achse auf der rechten Seite.

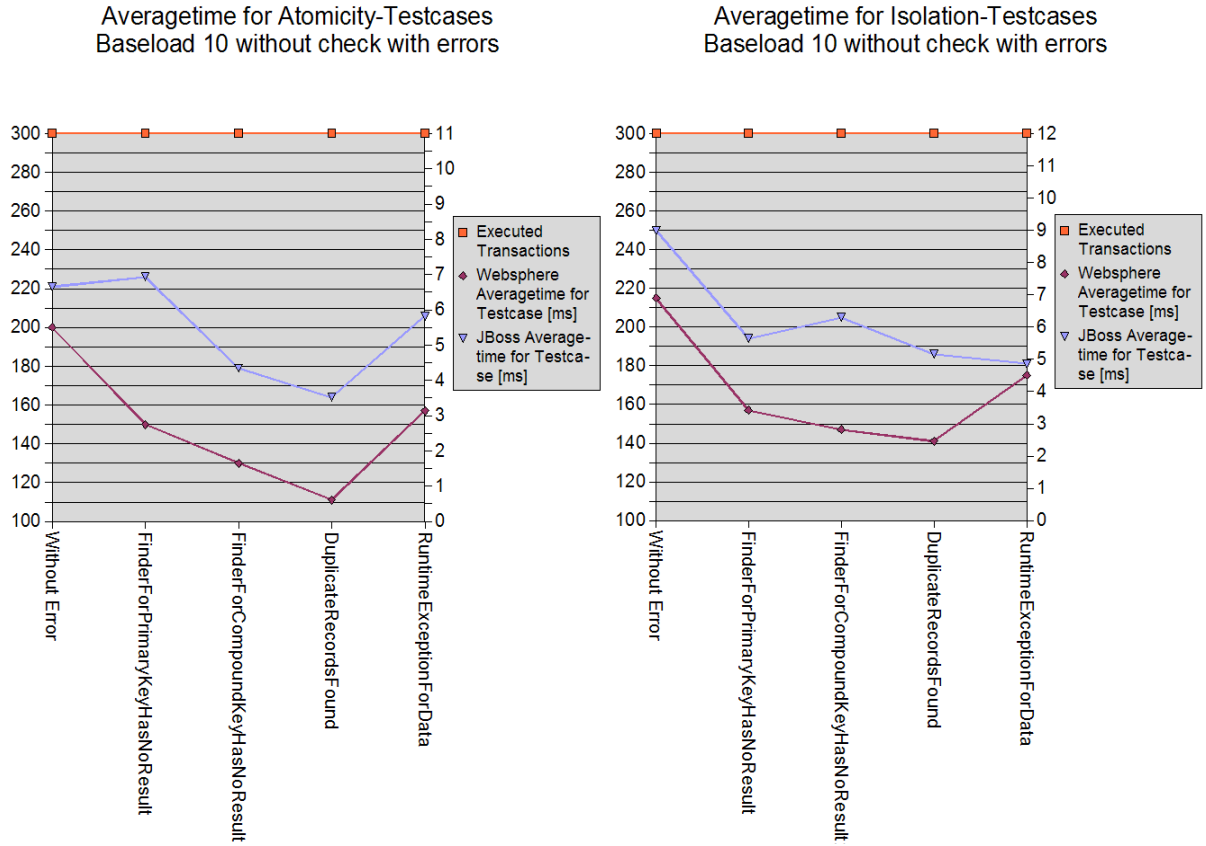

Abbildung 35 Performanz-Testergebnis für Atomaritäts- und Isolations-Testfälle der Testkategorie (3).

Dargestellt ist die durchschnittliche Ausführungsdauer (Averagetime) in Millisekunden (linke y-Achse) pro Testfall (x-Achse) sowie die Anzahl der pro Testfall durchgeführten Transaktionen (rechte y-Achse). Das linke Diagramm gibt die Ergebnisse für die Atomaritäts-Testfälle wieder, während das rechte Diagramm die Isolations-Testfälle beinhaltet. Die Testfälle pro Diagramm sind: Without Error = Ohne Fehlerfall, FinderForPrimaryKeyHasNoResult = Suche nach einem Primärschlüssel lieferte keine Entität zurück, FinderForCompoundKeyHasNoResult = Suche nach einem zusammengesetzten Fremdschlüssel lieferte keine Entität zurück, DuplicateRecordsFound = Suche nach einem eindeutigen Schlüssel lieferte mehrere Entitäten zurück, RuntimeExceptionForData = Bei der Auswertung der Daten ist ein Laufzeitfehler aufgetreten (z.B. NullPointerException).

Für das linke Diagramm aus Abbildung 34 existieren nur Performanz-Messdaten für den ersten Testfall, welcher ohne Fehlerfall ausgeführt wurde. Da diese Testkategorie keine Basislast beinhaltet, wurden für diese Atomaritäts-Testfälle nur jeweils eine Transaktion durchge-

führt, welche mit einer Ausnahmebedingung abgebrochen wurde. Hierfür wurden keine Performanz-Messdaten erhoben. In dem rechten Diagramm von Abbildung 34 existieren für die Testfälle wieder Werte, da die zweite Transaktion des Isolations-Testfalls auch in einem Fehlerfall erfolgreich beendet wird und somit Messdaten liefert. In den beiden Diagrammen von Abbildung 34 ist zu sehen, dass die Messergebnisse für JBoss und Websphere bei keiner bzw. einer parallelen Transaktion noch eng zusammen liegen, jedoch schon mit leichtem Vorteil für Websphere. Die Abbildung 35 beinhaltet Messdaten für alle Fehlerfälle, da diese Testkategorie mit einer Basislast von zehn parallelen Transaktionen ausgeführt wurde, welche Messdaten liefern. Es ist hier schon deutlich zu sehen, dass Websphere seinen Vorteil gegenüber JBoss ausbaut.

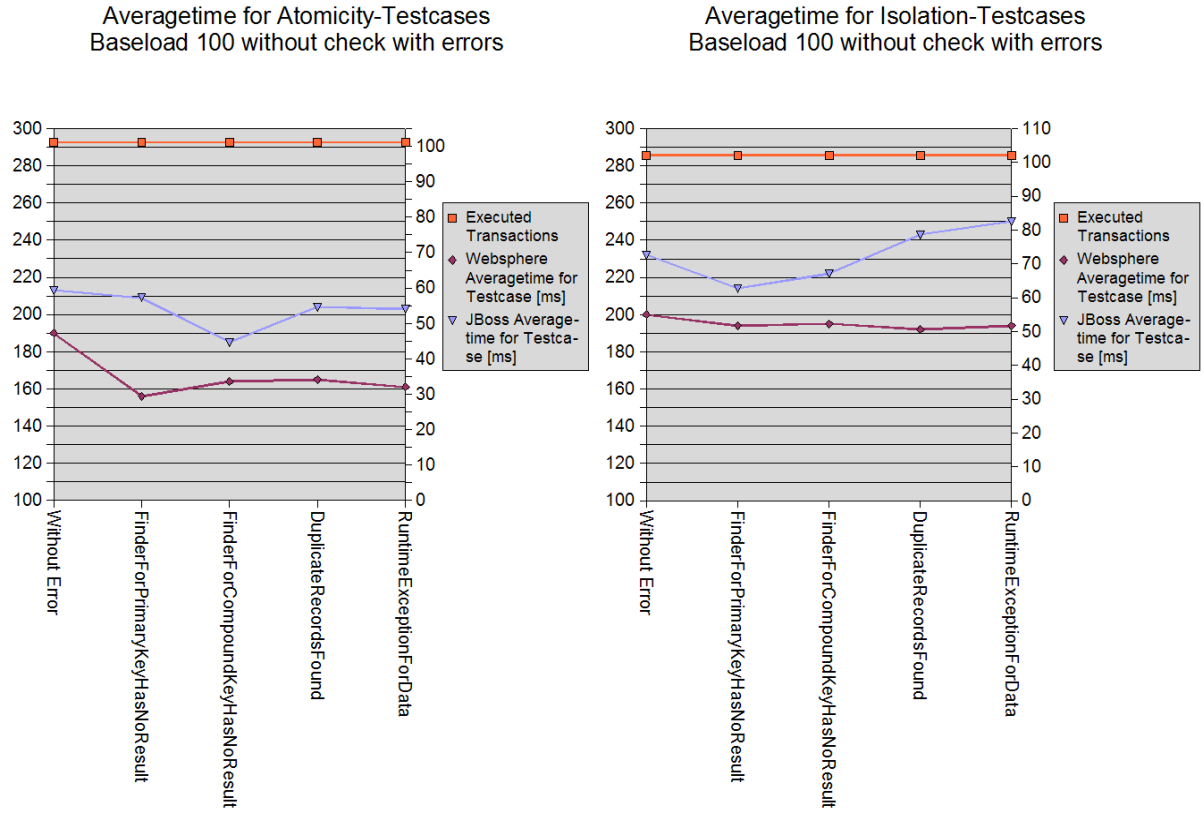

Abbildung 36 Performanz-Testergebnis für die Atomaritäts- und Isolations-Testfälle der Testkategorie (4).

Dargestellt ist die durchschnittliche Ausführungsdauer (Averagetime) in Millisekunden (linke y-Achse) pro Testfall (x-Achse) sowie die Anzahl der pro Testfall durchgeführten Transaktionen (rechte y-Achse).
Das linke Diagramm gibt die Ergebnisse für die Atomaritäts-Testfälle wieder, während das rechte Diagramm die Isolations-Testfälle beinhaltet. Die Testfälle pro Diagramm sind: Without Error = Ohne Fehlerfall, FinderForPrimaryKeyHasNoResult = Suche nach einem Primärschlüssel lieferte keine Entität zurück, FinderForCompoundKeyHasNoResult = Suche nach einem zusammengesetzten Fremdschlüssel lieferte keine Entität zurück, DuplicateRecordsFound = Suche nach einem eindeutigen Schlüssel lieferte mehrere Entitäten zurück, RuntimeExceptionForData = Bei der Auswertung der Daten ist ein Laufzeitfehler aufgetreten (z.B. NullPointerException).

Die Abbildung 36 beinhaltet Messdaten für alle Fehlerfälle, da diese Testkategorie mit einer Basislast von hundert parallelen Transaktionen ausgeführt wurde, welche Messdaten liefern. Es ist hier deutlich zu sehen, dass Websphere seinen Vorteil gegenüber JBoss behauptet.

7.2.1.3 Diskussion der Testergebnisse

Websphere, als marktführende Implementierung der J2EE-Spezifikation, hat zusammen mit DB2 die Erwartungen an Stabilität und Transaktionssicherheit, auch bei fatalen Fehlern, im vollen Umfang erfüllt. Störend haben sich während der Tests nur zwei Punkte gezeigt:

1. Der Aufruf eines Websphere-Dienstes von einer einfachen Java-Anwendung ist kompliziert. Es sind eine Vielzahl von Umgebungseinstellungen zu machen, sowie eine große Menge von JARs sowie Systembibliotheken der IBM JVM einzubinden. Erst damit ist ein erfolgreicher Aufruf der Websphere-Dienste und der in Websphere installierten Anwendungen möglich (Kapitel 4.2.3).
2. Bei dem Betrieb von mehreren JVM-Prozessen, welche über die IBM Orb Implementierung kommunizieren, war es nicht möglich, bei Ausfall eines Prozesses und nach einem anschließenden Neustart, ein *Reconnect* durchzuführen. Es kam auf der Ebene der Verbindungsaufnahme zwischen Client-Orb und Server-Orb zu einem Kommunikationsfehler (Kapitel 7.1.6).

Überraschender war die Feststellung, dass die Opensource Implementierung aus JBoss und der MaxDB, während der Tests selbst mit fatalen Fehlern zurecht kam und damit Stabilität und Transaktionssicherheit im selben Maße wie Websphere gewährleistete. Des weiteren stellte weder ein *Reconnect* nach einem fatalen JVM-Fehler noch der Aufruf von einer externen Java-Anwendung ein Problem dar. JBoss begnügt sich bei den Anforderungen an einen aufrufenden Client mit einem notwendigen Client-JAR sowie zwei zu setzenden Parametern für die Umgebung des `InitialContext`.

Auch die Performanz und der Durchsatz waren während der Tests durchaus vergleichbar, wenn auch Websphere in allen Kategorien etwas bessere Werte gezeigt hat (Abbildung 34, Abbildung 35 und Abbildung 36). Vor allem bei steigender Nebenläufigkeit und dem Setzen von Sperren auf denselben Ressourcen (jeweils rechtes Diagramm von Abbildung 35 und Abbildung 36), skaliert Websphere besser. Hier ist deutlich zu sehen, dass die Ressourcennutzung von Websphere ausgereifter ist und die Vermeidung des dynamischen Aspektes, im Gegensatz zu JBoss, sich positiv bemerkbar macht. Die Performanz-Betrachtungen waren jedoch, wie in der Einleitung beschrieben, nur ein untergeordneter Untersu-

chungspunkt, welche deshalb lediglich eine qualitative und keine quantitative Aussage über Performanz und Durchsatz zulassen.

Es stellt sich die Frage, wie sich begründen lässt, dass JBoss bei den Tests denselben Grad an Gewährleistung der Transaktionssicherheit aufwies wie Websphere. Bei genauerer Betrachtung und Berücksichtigung der in Kapitel 4 analysierten Implementierungsdetails, ist dies jedoch nicht verwunderlich.

Der Verdienst für die Gewährleistung der Transaktionssicherheit unter der Rahmenbedingung von einfachen Transaktionen (keine verteilte Transaktionen) liegt nicht bei JBoss, sondern bei der verwendeten Datenbank. Die Implementierung des Transaktionsmanagers ist

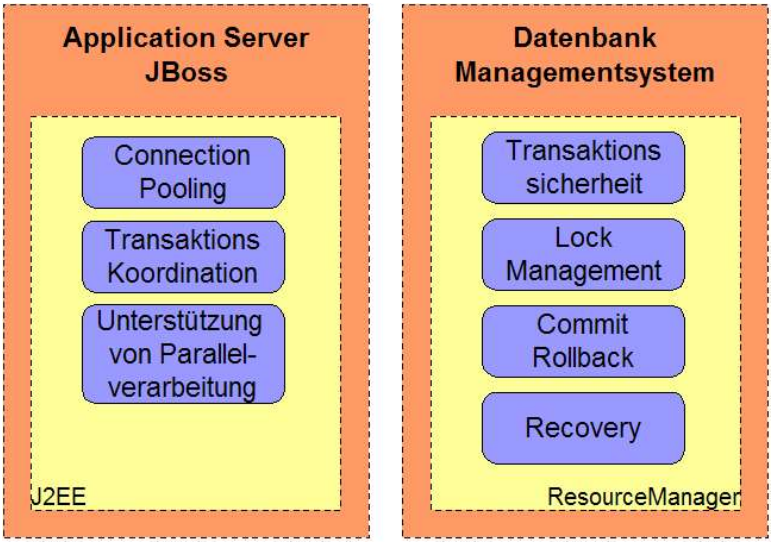

Abbildung 37 Aufteilung von Verantwortung zwischen JBoss und dem DBMS

bei JBoss recht einfach gehalten. Das Schreiben von *Recoverylogs* und die Möglichkeit zum *Recovery* fehlt beispielsweise völlig. JBoss übernimmt eigentlich nur die Rolle des Transaktionskoordinators und der damit verbundenen Bereitstellung des korrekten Transaktionskontextes für einen Aufruf. Die wirkliche Durchführung der Transaktion findet in der Implementierung der *XAResource* bzw. im damit assoziierten *Resourcemanager* statt. Die Verantwortung für die korrekte Durchführung von lokalen Transaktionen und deren Festschreibung (Commit) oder Zurücksetzung (Rollback) liegt demnach bei der MaxDB-Datenbank. Die Datenbank gewährleistet die Transaktionssicherheit und führt bei Bedarf *Recovery*-Funktionalitäten durch. JBoss ist im Gegenzug für die effiziente Ressourcenbenutzung, beispielsweise *Connection-Pooling* von Datenbank-Verbindungen, zuständig sowie die Koordination der Transaktion. Dies beinhaltet bei lokalen Transaktionen jedoch lediglich die korrekte Zuweisung des Transaktionskontextes bei Ausführung des Aufrufs im EJB-Container sowie die Registrierung und Deregistrierung von verwendeten `XAResourcen`. Der *Lockmanager* in JBoss ist hauptsächlich dafür zuständig, die Anforderung der EJB-Spezifikation zu gewähr-

leisten, eine Bean-Instanz nicht parallel von mehreren Threads durchlaufen zu lassen. Die tatsächliche Isolation durch das Setzen von Sperren auf Ressourcen findet bei JBoss in der Datenbank statt. Die Aufteilung dieser Verantwortlichkeiten ist in Abbildung 37 dargestellt.

Websphere, im Gegensatz, unterstützt in seiner Infrastruktur direkt das *Recovery* und verfügt über eine erweiterte Implementierung eines *Lockmanagers*. Websphere schreibt für jede Transaktion *Recoverylogs* und stellt den gewünschten Isolationsgrad über optimistische Synchronisationsverfahren oder pessimistische Sperrverfahren zusammen mit der Datenbank sicher. Der größte Teil des Quellcodes der Transaktionsverarbeitung ist für die Fehlerbehandlung zuständig und damit für die Gewährleistung der Transaktionssicherheit. Die Verantwortung für die letztendliche Festschreibung der Datenbankressourcen einer Transaktion trägt natürlich immer noch der *Resourcemanager*. Ebenso ist das Setzen von Sperren im DBMS weiterhin notwendig, da nicht alle Zugriffe über den Applicationserver getätigt werden (Kapitel 3.3.4). Die Aufteilung der Verantwortlichkeiten ist in Abbildung 38 dargestellt.

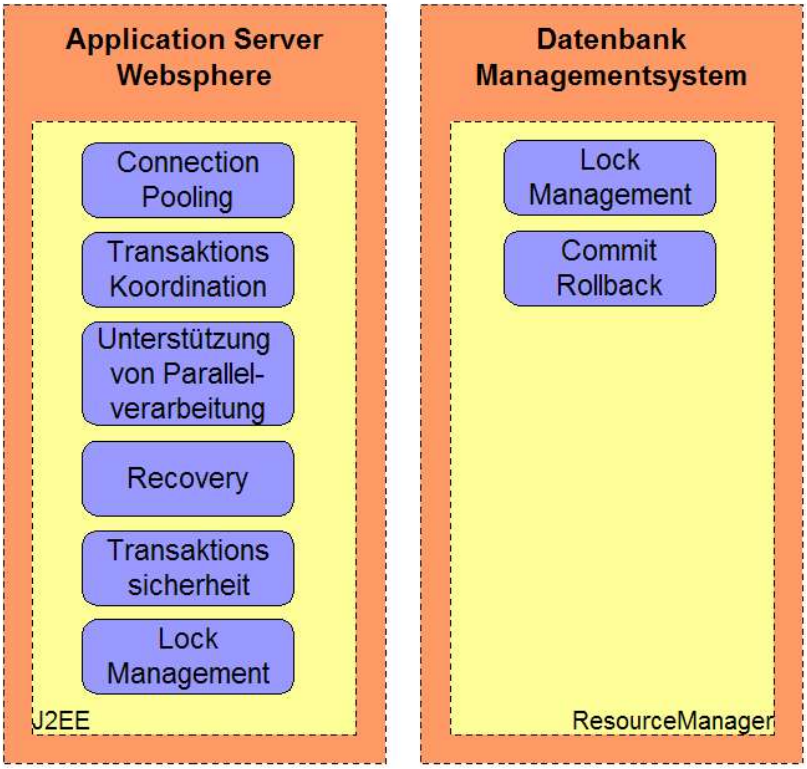

Abbildung 38 Aufteilung von Verantwortung zwischen Websphere und dem DBMS

Somit gelangen zwar die Tests zu demselben Ergebnis, jedoch stellen die beiden getesteten J2EE-Infrastrukturen die Transaktionssicherheit auf vollkommen unterschiedliche Weise sicher. Bei JBoss liegt die Verantwortung bei der Datenbank, somit sind die guten Testergebnisse der guten Implementierung der MaxDB zu verdanken. Würde ein anderes DBMS eingesetzt werden, welches Probleme bei der Gewährleistung der Transaktionssicherheit hat, bei-

spielsweise beim *Recovery* nach einem Systemfehler, hätten die Testergebnisse wahrscheinlich anders ausgesehen. Die guten Performanzdaten sind dagegen ein Verdienst von JBoss, welche jedoch durch die leichtgewichtige Realisierung des Transaktionsdienstes begünstigt werden.

Bei Websphere liegt die Verantwortung bei dem Transaktionsdienst von Websphere selbst. Dies bedeutet einerseits, dass mit Websphere dem DBMS viele Aufgaben abgenommen werden und damit theoretisch auch einfachere Systeme als DBMS einsetzbar sind. Zum anderen liegt in diesem Merkmal auch ein Grund für die bessere Performanz und auch Skalierung von Websphere unter zunehmender Last, da die Ressourcen innerhalb von Websphere kontrolliert werden und damit effektiver optimiert werden können als in einem externen DBMS-Prozess.

7.2.2 Anwendungstests

Um Aussagen über die Eignung des TTS für die Gewährleistung von stabilen und zuverlässigen J2EE-Anwendungen zu erhalten, wurden bewusst Fehler in die J2EE-Anwendung des SUT eingebaut. Das Ergebnis war, dass mit dem TTS sämtliche bewusst eingebauten Fehler sowie einige weitere Fehler aus den Bereichen Konfiguration, Nebenläufigkeit, Locking und fehlerhafte Ressourcenfreigabe gefunden wurden:

- Zu geringe Größe des *Heaps* der JVM für die Abarbeitung der erforderlichen Anzahl an parallelen Transaktionen.
- Zu geringe Anzahl von erlaubten Datenbankverbindungen, um die erforderliche Anzahl von parallelen Transaktionen zu verarbeiten.
- Fehler (`ConcurrentModificationException`) durch die gleichzeitige Manipulation von gemeinsam benutzten Ressourcen in unterschiedlichen Threads, auf die jedoch nur nacheinander zugegriffen werden darf.
- Verschiedene *Deadlock*-Situationen, ausgelöst beispielsweise durch Zugriff zweier Transaktionen auf dieselben Datensätze der Entität *Item*, jedoch in unterschiedlicher Reihenfolge.
- Skalierungsprobleme der Anwendung durch das Setzen von exklusiven Sperren für *Readonly*-Operationen (Kapitel 7.1.5, Testkategorie (2) und Abbildung 33), welche besser über eine dedizierte *Datasource*-Definition stattfinden sollte.
- Entwicklungsfehler, welche zu Endlosschleifen führten und damit zu einem *OutOfMemory*-Laufzeitfehler.
- Konfigurationsfehler der Isolation für die *Datasource*-Definition (Isolationsstufe *Read Commited* anstatt *Repeatable Read*). Der *NewOrderTransaction*-Prozess hat als Atomaritäts-Testfall funktioniert (nur eine Transaktion), jedoch als Isolationstestfall für Schreib-/

Schreibzugriffe nicht (zwei Transaktionen auf derselben *Customer*-Entität und damit auch auf demselben *District* indem die Auftragsnummer verwaltet wird). Hier hat die Konsistenzprüfung angemerkt, dass die Konsistenzbedingung (1) aus Kapitel 6.4 verletzt wurde.

- Entwicklungsfehler beim Speichern des *NewOrder*-Datensatzes wurde entdeckt durch Verstoß gegen Konsistenzkriterium (2) aus Kapitel 6.4.
- Fehler bei Konfiguration des Benutzers und der Zugriffsrechte in der *Datasource*-Definition und der Datenbank selbst.
- Fehler bzw. Inkonsistenzen in den JNDI-Namen der EJBs im EAR, welche zu `NameNotFound`-Fehlern beim Aufruf von Testfällen führte.
- Fehlerhaftes oder fehlendes Freigeben von Ressourcen (z.B. *Filehandle* oder *Streams*), welche sich erst bei massiver Nebenläufigkeit zu Ressourcen-Engpässen und damit zu Fehlern steigerten.

Darüber hinaus konnte das TTS auch zur Optimierung der Konfiguration hinsichtlich des Durchsatz erfolgreich eingesetzt werden. Bei JBoss wurde die Konfiguration `insert-after-ejbPostCreate` aktiviert, sodass nur ein einzelnes Datenbankstatement abgesetzt wurde anstatt zwei. Generell wurde sowohl bei Websphere als auch bei JBoss die Commit-Option A (Kapitel 3.3.4) gewählt, da die Anwendung als einzige auf die Datenbank manipulierend zugreift.

Das TTS erscheint demnach für das Aufspüren von Anwendungs-, Integrations- und Konfigurationsfehlern geeignet zu sein. Besonders für die verlässliche reproduzierbare Teststellung von Nebenläufigkeitsszenarien im Normalbetrieb und unter Fehlereinwirkung spielt das TTS seine Stärke gegenüber herkömmlichen Modultest bzw. Lasttest aus. Der Modultest ist nur auf den Test einer funktionalen Anforderung ausgelegt und berücksichtigt nicht die Problematiken der Nebenläufigkeit sowie der Integration des Moduls in das Gesamtsystem. Der Lasttest erzeugt eine Eingangslast für eine Komponente, ein Teilsystem oder ein Gesamtsystem. Es erfolgt eine Analyse auf Durchlaufzeit und Ressourcennutzung des betrachteten Systems. Ziel ist die Optimierung des Gesamtsystems und die Analyse und die Tests von Konfigurationen (Hardware sowie Software), welche mit der erwarteten Produktionslast an Nebenläufigkeit zurechtkommen sollen. Das TTS simuliert Unternehmensprozesse und komplexe Fehlersituationen bei Nebenläufigkeit. Es ist in der Lage verschiedene Transaktionszusammenstellungen, mit frei zu definierender Basislast unter Fehlereinwirkung, bis hin zu fatalen Systemfehlern reproduzierbar zu machen und kontrolliert durchzuführen und auszuwerten. Ebenso scheint das TTS geeignet zu sein, Quellcode sowie die Konfiguration des Systems zu optimieren und bei Versionswechseln von Infrastruktur-Komponenten die Kompatibilität sicherzustellen.

Bei der Entwicklung des TTS und der Standard-J2EE-Anwendung für das SUT hat sich sehr schnell ein bestimmter Entwicklungszyklus eingestellt, welcher im folgenden auf einen Einsatz in einem beliebigen Unternehmen verallgemeinert wird. In Anlehnung an andere Vorgehensmodelle, wie beispielsweise das V-Modell, wird es als T-Modell referenziert (Abbildung 39).

Das Vorgehensmodell startet mit der Entscheidung des Managements, eine Anwendung auf Basis der J2EE-Plattform zu entwickeln (Abbildung 39, 1). Es werden möglicherweise schon einige Basisentscheidungen bzgl. Budget, Zeit und auch zu Produkten getroffen. Für den hier betrachteten Abstraktionsgrad des Vorgehensmodells sei dahingestellt, ob eine bereits existierende Anwendung portiert werden oder eine Anwendung vollkommen neu entwickelt werden soll. Die Infrastrukturbetreuung, welche die zukünftige J2EE-Infrastruktur installiert, konfiguriert und betreut, stellt eine möglichst produktionsnahe J2EE-Infrastruktur auf einem dedizierten Testsystem bereit (Abbildung 39, 2). Die Anwendungsentwicklung entwickelt nach Maßgabe von fachlichen Anforderungen die J2EE-Anwendung, welche später in der Produktion eingesetzt werden soll und stellt die Anwendung für das Testsystem bereit (Abbildung 39, 3).

Das Setup des TTS beinhaltet die Konfiguration der J2EE-Infrastruktur für die Bedürfnisse der J2EE-Anwendung (z.B. *Datasource*-Definitionen, Sicherheitseinstellungen) ebenso, wie das Schreiben und Zusammenstellen der Test-, Validierungs- und Fehlerfälle (Testkonfiguration) sowie das anschließende *Deployment* der Anwendung auf die J2EE-Infrastruktur. Die Setupphase stellt ein vollständig konfiguriertes Testsystem zur Verfügung, welches bereit ist für die Durchführung von Testfällen (Abbildung 39, 4). Die anschließende Simulationsphase führt die Simulation der Unternehmensprozesse, auf Basis der in der Setupphase definierten Testkonfigurationen, durch. Es werden die vom Test-Client gewünschten Rahmenparameter bezüglich Basislast und Fehlerkategorien in der Ausführung berücksichtigt sowie die Ablaufsteuerung der einzelnen Testprozesse und die Synchronisierung der Testprozesse bezüglich bestimmter Testfälle durchgeführt. Der letzte Schritt der Simulationsphase ist die Durchführung der Validierungslogik für die Testprozesse und die Konsistenzprüfung der Daten, mit anschließender Speicherung in der TCS-Datenbank. Diese Daten werden der darauf folgenden Auswertungsphase zur Verfügung gestellt (Abbildung 39, 5). In der Auswertungsphase werden die während der Simulationsphase gesammelten Daten ausgewertet und Detailmetrikdaten (vgl. Kapitel 6.5) daraus abgeleitet und gespeichert. Anschließend werden die wichtigsten Daten auf der Ebene von Vergleichsmetriken kumuliert, die einen schnellen Überblick über den Testzyklus und die Eignung unterschiedlicher Produkte bzw. Produktkonfigurationen erlauben.

Die Detailmetriken sind vor allem für die Anwendungsentwicklung wichtig, um Problemstellen innerhalb der J2EE-Anwendung zu entdecken und durch geeignete Modifikationen zu lösen (Abbildung 39, 6). Die Infrastrukturbetreuung zieht aus den Vergleichsmetriken für un-

terschiedliche Produkte bzw. Produktkonfigurationen den meisten Nutzen, obwohl für ein Detailverständnis die Betrachtung der Detailmetriken durchaus auch von Vorteil ist. Der Infrastruktursupport kann Fehler in der Konfiguration erkennen bzw. Produktkonfigurationen anhand der gelieferten Daten optimieren (Abbildung 39, 7).

Das Management schließlich dürfte ausschließlich Interesse an den Vergleichsmetriken und den darin dargestellten Zahlen für erfolgreich durchgeführte Tests pro Produktkonfiguration sowie der relativen Performanz haben. Zusammen mit den Informationen über Kosten der Infrastruktur und Kosten für Spezifikas des Entwicklungsprozesses lassen sich sehr viel effektiver gesicherte Entscheidungen für oder gegen Produkte treffen, aber auch steuernd in den Entwicklungsprozess einer J2EE-Anwendung eingreifen. Die Folge ist eine Reduzierung des Risikos bei der Durchführung eines solchen Unterfangens sowie eine schnellere und qualitativ hochwertigere Entwicklung eines stabilen und zuverlässigen J2EE-Systems.

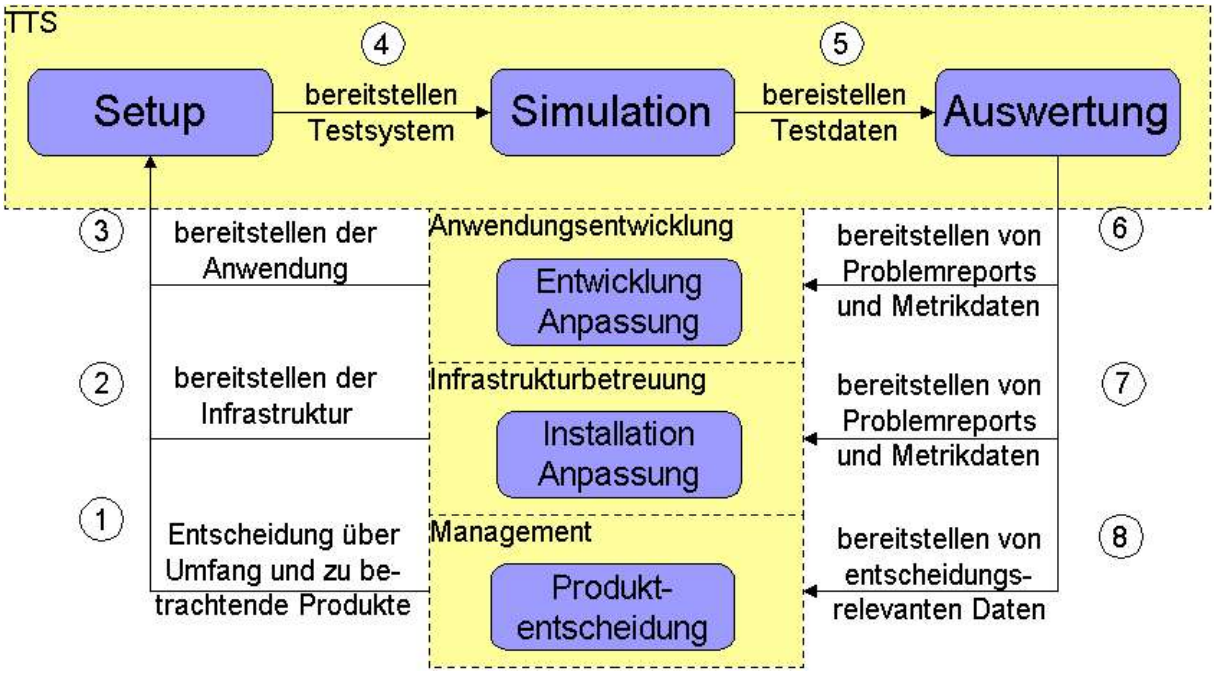

Abbildung 39 Entwicklungszyklus mit TTS (T-Modell)

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

8.1.1 Grundlagen der Transaktionsverarbeitung und J2EE

Angesichts des zunehmenden Einsatzes von J2EE-Applicationservern für unternehmenskritische Anwendungen stellt sich die Frage, wie es um den Entwicklungsstand der zentralen Komponenten, wie der Transaktionsverarbeitung, bestellt ist. Die bisher für diesen Zweck eingesetzten Transaktionsmonitore haben eine lange Entwicklungsgeschichte hinter sich, und Produkte wie IBM CICS, Bea Tuxedo oder SAP R/3 gelten heute als stabil und ausgereift.

Obwohl die J2EE-Plattform eine offengelegte, standardisierte Spezifikation ist, definiert sie nur ein minimales Rahmenwerk und Interaktionsschnittstellen. Es ist weitgehend den J2EE-Produkten überlassen, wie sie Spezifikationslücken handhaben, Funktionalitäten intern umsetzen und Stabilität sicherstellen. Das J2EE-Produkt stellt sich damit weitgehend als *Blackbox* dar. Die Frage nach der Qualität und der Funktionsweise der Transaktionsverarbeitung in J2EE-Systemen führt zu der Frage, was Transaktionsverarbeitung und das Konstrukt der Transaktion selbst ist und welche Mittel zur Gewährleistung von korrektem Verhalten von Transaktionen eingesetzt werden.

Eine Transaktion kann als eine Interaktion mit einem System betrachtet werden, welche den Zustand des Systems ändert. Diese Interaktion ist nach [OTS03] eine logische Einheit (*Unit of Work*), welche die Eigenschaften Atomarität, Konsistenz, Isolation und Dauerhaftigkeit (ACID) unterstützt. Die Atomarität einer Transaktion bedeutet, dass eine Transaktion als logische Einheit gesehen wird, die entweder ganz oder gar nicht ausgeführt werden darf ([TPCC03], [OTS03]). Bei Abbruch der Transaktion müssen bereits ausgeführte Teiloperationen zurückgesetzt werden. Das Konsistenzkriterium bedingt, dass eine Transaktion ein System von einem konsistenten Zustand in einen weiteren konsistenten Zustand überführt [TPCC03]. Die Isolation von nebenläufigen Transaktionen untereinander ist essentiell für ein korrektes Ergebnis der Transaktion. Die ANSI SQL-Spezifikation unterscheidet die vier Isolationsstufen *Read Uncommited*, *Read Commited*, *Repeatable Read* und *Serializable*, welche eine zunehmende Isolation zwischen nebenläufigen Transaktionen bewirken. Diese werden jedoch durch eine steigenden Anzahl von gesetzten Sperren (*Locks*) realisiert, welche zu einem sinkenden Grad an Parallelverarbeitung führen. Das Dauerhaftigkeits-Kriterium einer Transaktion bedingt, dass das Ergebnis einer Transaktion solange Bestand hat, bis es von einer anderen Transaktion geändert wird.

Isolation wird durch Synchronisierungsverfahren realisiert. Es wird zwischen optimistischen Verfahren, verzögernden Verfahren und pessimistischen Sperrverfahren unterschieden. Optimistische Verfahren analysieren kontinuierlich die ablaufenden Transaktionen auf ihre Serialisierbarkeit und setzen bei Verletzung der Serialisierbarkeitsbedingung die betroffenen Transaktionen zurück. Pessimistische Verfahren verhindern Verletzungen der Serialisierbarkeitsbedingung dadurch, dass Datenobjekte für eine Transaktion explizit gesperrt werden, sodass die ablaufende Transaktion, für eine bestimmte Zeit, einen exklusiven Zugriff auf das gesperrte Datenobjekt hat.

Es wird zwischen den Transaktionstypen lokale Transaktion, verteilte Transaktion und *Business Transaction* unterschieden. Lokale Transaktionen müssen die ACID-Kriterien auf einer einzelnen Datenquelle sicherstellen, verteilte Transaktionen auf mehreren Datenquellen. Um in einem solchen Szenario ein konsistentes Commit durchzuführen, müssen Transaktionsmanager und Ressourcen ein mehrphasiges Festschreibungsprotokoll unterstützen. Das weit verbreitete *Two Phase Commit*-Protokoll (2PC) realisiert dies über eine *Prepare*-Phase und eine separate *Commit*-Phase.

J2EE ist eine Dachspezifikation, bestehend aus einer Vielzahl von Einzelspezifikationen, welche zusammen ein Serverframework für verteilte Systeme sowie mit der EJB-Spezifikation ein Programmiermodell für die Erstellung von stabilen, transaktionalen und skalierbaren Anwendungskomponenten darstellt. J2EE-Produkt-Hersteller müssen gemäß der aktuellen Version 1.4 ausschließlich das *Flat Transaction*-Modell unterstützen, welche sich jedoch über mehrere Komponenten und transaktionale Ressourcen erstrecken kann. Als transaktionale Ressourcen werden Verbindungen zu relationalen Datenbanksystemen über JDBC, *JMS Sessions* und Verbindungen zu einem *Resource Adapter* verstanden [JTA99].

EJBs bestehen aus einer Bean-Klasse sowie einer `EJBHome`- und einer `EJBObject`-Schnittstelle, über welche ein Client innerhalb derselben JVM oder auch einer entfernten JVM mit der Bean-Klasse interagieren kann. Die Bean-Klasse ist nicht im besonderen dafür ausgelegt, Remote-Kommunikation mit dem Client zu unterstützen; dies gewährleistet die Infrastruktur der J2EE-Plattform. Die Transaktionsgrenzen können über zwei Arten bei EJBs gezogen werden. Die *Bean Managed Transaction (BMT)* erlaubt dem Entwickler über die `UserTransaction`-Schnittstelle der *Java Transaction API (JTA)* die Transaktionsgrenzen explizit zu steuern, wohingegen bei der *Container Managed Transaction (CMT)* die Begrenzung im *Deployment Descriptor* durch die Transaktionsattribute *Required* (Aufruf mit Transaktionskontext), *RequiresNew* (Aufruf mit neuem Transaktionskontext), *NotSupported* (Aufruf ohne Transaktionskontext), *Supports* (Aufruf mit vorhandenem Transaktionskontext falls existent), *Mandatory* (ein Transaktionskontext muss für den Aufruf vorhanden sein) und *Never* (kein Transaktionskontext darf für den Aufruf vorhanden sein) deklariert werden. Über die Commit-Optionen kann die Nutzung von Ressourcen durch den Container effizienter

gestaltet werden. Bei Option A geht der Container, anders als bei Option B und C, von der Annahme aus allein auf der Datenquelle zu arbeiten, sodass ein Datenabgleich der Bean-Instanz am Anfang einer neuen Transaktion nicht erfolgen muss.

Für die Transaktionsverarbeitung definiert die J2EE-Spezifikation die Rollen Transaktionsmanager, EJBContainer, *Resourcemanager*, Anwendungskomponente (EJB) und *Communication Resourcemanager (CRM)*. Die JTA-Spezifikation enthält eine Java-Umsetzung des X/Open Standards für verteilte Transaktionen und definiert den Transaktionsmanager als Komponente, welcher Dienst- und Verwaltungsfunktionalitäten zur Verfügung stellt, um Transaktionsgrenzen zu steuern, transaktionale Ressourcen zu verwalten, Synchronisation von EJBs zu erlauben und die Propagation des Transaktionskontextes zu ermöglichen. Die EJB-Spezifikation definiert den EJBContainer des Applicationservers als Laufzeitumgebung für Anwendungskomponenten und als Bindeglied zwischen Anwendungskomponenten, *Resourcemanager* und Transaktionsmanager. Der *Resourcemanager* (z.B. ein RDBMS per JDBC-Treiber) ist Teil der *Java Connector Architecture (JCA)* und stellt den Anwendungskomponenten Zugriff auf Ressourcen über einen *Resource Adapter* zur Verfügung. Er wird vom Transaktionsmanager benutzt, um die Festschreibung (Commit oder Rollback) von Transaktionen durchzuführen und Wiederherstellungsarbeiten (*Recovery*) durchzuführen. Eine Anwendungskomponente ist ein EJB innerhalb eines J2EE-Applicationservers und nutzt Dienste wie das Transaktionsmanagement über BMT oder CMT. Der CRM ist für den Import und Export des Transaktionskontextes von und nach extern zuständig und wird durch die *Java Transaction Service*-Spezifikation (JTS) genauer definiert.

8.1.2 Untersuchung von JBoss und Websphere

Es wurden in dieser Arbeit die J2EE-Implementierungen JBoss Applicationserver und IBM Websphere untersucht. JBoss ist eine Opensource-Implementierung und hat vor kurzem in der Version 4.0 den Kompatibilitätstest für die J2EE 1.4 – Zertifizierung erfolgreich bestanden [IW04]. Die Architektur des JBoss Applicationservers basiert auf der *Java Management Extension (JMX)* und unterstützt *Hot-Deployment* für EARs und Konfigurationsänderungen.

Jedes Modul in JBoss ist ein eigenständiges JMX-Modul, welches separat verwaltet werden kann und damit JBoss sehr flexibel macht. Die Kommunikation zwischen Client und Server sowie die Abbildung von `EJBHome` und `EJBObject` auf konkrete EJBs funktioniert bei JBoss über die *Reflection API*. Die Servlet-Laufzeitumgebung Tomcat ist als Webcontainer standardmäßig in JBoss integriert.

JBoss spricht seinen Transaktionsmanager (bis auf den Import und Export des Transaktionskontextes) allein über die JTA-Schnittstellen an, sodass sich verhältnismäßig einfach auch andere JTA-konforme Transaktionsmanager integrieren lassen. Der Standard-Trans-

aktionsmanager von JBoss unterstützt *Interposing*. Hierbei werden, für eine effizientere Kommunikation zwischen verteilten Systemen, die an einer Transaktion beteiligten Ressourcen durch den jeweils lokalen Transaktionsmanager verwaltet und koordiniert, welcher als einziger an der globalen Transaktion registriert ist. Jedoch unterstützt der Standard-Transaktionsmanager kein *Recovery* und auch keine Propagation des Transaktionskontextes über JVM-Grenzen hinaus. Der ebenfalls im Lieferumfang von JBoss enthaltene OTS-spezifikationskonforme Tyrex Transaktionsmanager hat diese Einschränkungen nicht und kann wahlweise als Transaktionsmanager in JBoss benutzt werden.

Der Applicationserver Websphere von IBM ist eine kommerzielle Implementierung der J2EE-Spezifikation. Der IBM Websphere Applicationserver basiert auf der CORBA Implementierung von IBM, welche IIOP zur Kommunikation zwischen JVM-Prozessen benutzt. Der Transaktionsmanager, welcher in Websphere benutzt wird, basiert nicht auf CORBA, sondern ist eine Java-Implementierung. CORBA-Ressourcen können über die JTS-Implementierung von Websphere an Transaktionen teilnehmen. Im Lieferumfang von Websphere sind zwei Transaktionsmanager-Implementierungen verfügbar. Der Standard-Transaktionsmanager unterstützt das *Recovery* von Transaktionen und transaktionalen Ressourcen, die andere Implementierung ist leichtgewichtiger und unterstützt dies nicht.

Anders als JBoss minimiert Websphere den dynamischen Aspekt durch die Erzeugung von zusätzlichen Klassen aus den EJBs und den Informationen des *Deployment Descriptors* beim *Deployment*, welche die Integration der EJBs in die Websphere-Laufzeitumgebung ermöglichen. Der Persistenzzugriff ist in Websphere nach dem Konzept eines *Resource Adapter* (JCA) realisiert und wird für die EntityBeans, einschließlich Schnittstellen und JDBC-Statements, ebenfalls beim *Deployment* generiert.

JBoss als Opensource-Implementierung der J2EE-Spezifikation trennen nicht nur lizenzspezifische Aspekte von der kommerziellen Implementierung Websphere von IBM. JBoss besteht aus knapp 4.300 Klassen und belegt installiert 50 MB, Websphere besteht aus über 20.000 Klassen und belegt installiert 460 MB. Die Stärken von JBoss liegen klar in seiner Flexibilität und dem modularen Aufbau, welche einen einfachen Zugang zu Verständnis und Funktionalität ermöglicht. Die einfache Erweiterbarkeit durch eigene Komponenten und die Austauschbarkeit selbst von Kernkomponenten wie dem Transaktionsmanager, ermöglichen die fallweise Anpassung auf konkrete Bedürfnisse des Anwenders (z.B. einklinken von eigenen Interzeptoren in einen EJB-Aufruf). In Websphere wird sehr viel Aufwand für die Fehlerbehandlung betrieben, v.a. in den zentralen Komponenten der Transaktionsverarbeitung (ca. 80% bei Websphere gegenüber ca. 40% bei JBoss).

8.1.3 Problematik der Isolation und existierende Lösungsansätze

Um auf die Frage eine Antwort zu finden, wie es um die Gewährleistung der Transaktionssicherheit bei J2EE-Systemen steht, muss zunächst geklärt werden, was unter Transaktionssicherheit zu verstehen ist und wie die Transaktionsverarbeitung gefährdet werden kann.

Unter Transaktionssicherheit wird die qualitative Eigenschaft eines transaktionsverarbeitenden Systems verstanden, welches das gesicherte und isolierte Ausführen von Transaktionen ermöglicht. Dies bedeutet insbesondere, dass vorangegangene, parallel durchgeführte oder fehlgeschlagene Transaktionen sich nicht außerhalb der transaktional- veränderten Datenbasis der betreffenden Datenquellen beeinflussen dürfen. [Borm01] führt zu dieser Isolationsproblematik durch das System die Grade Beeinflussung und Zuverlässigkeit an. Beeinflussung geschieht durch Manipulation des globalen Systemzustands durch eine erste Transaktion (z.B. statische Variable) und Ausführung einer zweiten Transaktion innerhalb dieses geänderten Systemzustands mit Auswirkung auf das Ergebnis. Die Zuverlässigkeit adressiert die Fortpflanzung von Fehlern einer Transaktion auf alle momentan parallel ausgeführten Transaktionen (kritische JNI-Fehler in der JVM oder Speicherknappheit).

Es existieren mittlerweile einige Lösungsansätze für diese Problematik. Der Lösungsansatz, die Isolation mittels Betriebssystemprozessen sicherzustellen, wird in der *Persistent Reusable JVM* [Borm01] angewendet. Die Vorteile dieses Ansatzes sind vor allem durch den hohen (jedoch nicht absoluten) Grad der Isolation durch das Betriebssystem und durch die *Reset*-Fähigkeit der JVM gegeben, welche damit sowohl die Isolationsproblematik Beeinflussung als auch Zuverlässigkeit adressiert. Nachteil ist der, trotz verschiedener Optimierungen, immer noch existierende höhere Ressourcenverbrauch bei der Verwendung von Betriebssystemprozessen gegenüber Threads und dem Kommunikations-Overhead bei Kollaboration von verteilten Komponenten.

Die Isolation mittels Classloader wird in fast allen J2EE-Implementierungen eingesetzt, zusammen mit einem Multi-Threading-Ansatz. Die Isolation wird mittels Classloader-Hierarchien, Restriktionen in der EJB-Spezifikation und der Verlagerung von kritischen Funktionalitäten aus der Anwendung heraus in die „vertrauenswürdige“ J2EE-Infrastruktur adressiert. Die Vorteile dieses Ansatzes sind in der hohen Performanz und dem verhältnismäßig geringen Ressourcenverbrauch zu sehen. Nachteil ist die vergleichsweise hohe Verantwortung der Anwendungsentwicklung für die Isolation und die durch die höhere Anzahl der LOCs gestiegene Fehlerwahrscheinlichkeit der Isolations-Implementierung.

Die Isolation durch Modifikation der JVM sicherzustellen, wird vor allem von [Cza00] präferiert. Auf Ebene der JVM werden statische Klassenvariablen pro Thread vervielfältigt und lokal gehalten, sodass eine Beeinflussung von Transaktionen durch dieses Mittel nicht mehr möglich ist. Die Zuverlässigkeit wird wie im J2EE-Fall innerhalb der JVM realisiert. Vorteile gegenüber dem Classloader-Ansatz sind, neben höherer Performanz und dem optima-

leren Ressourcenverbrauch durch das nur einmalige Laden der Klassen, auch die automatische Verfügbarkeit der Isolation (Reduzierung der Verantwortung der Anwendungsentwicklung). Nachteile sind, neben der schon erwähnten Realisierung der Isolation innerhalb der JVM, auch, dass die Isolation bzgl. Beeinflussung von nachfolgenden Transaktionen nicht direkt gegeben ist.

Die kürzlich fertiggestellte Isolate-API definiert eine Basis API für die Isolation von Komponenten. Java-Anwendungen werden unabhängig voneinander innerhalb von *Isolates* ausgeführt, welche keine Objektinstanzen mit anderen *Isolates* teilen können und eine eigene Kopie der statischen Klassenvariablen hält. Wie Threads, ermöglichen *Isolates* die nebenläufige Ausführung von Anwendungen und lassen sich erzeugen, starten, beenden und managen. Darüber hinaus verfügen sie wie eine JVM über einen eigenen *System-Level*-Kontext.

Die Güte und Art der Isolation kommt dabei stark auf die Implementierung selbst an. Die Technologie der PRJVM ist gut dafür geeignet, eine Abbildung von *Isolates* auf Betriebssystemprozesse zu ermöglichen (*One-to-One*). Die in [Cza00] vorgestellte Modifikation der JVM und die Separation von statischen Klassenvariablen ist die Basis für die Abbildung aller *Isolates* einer JVM auf einen Betriebssystemprozess (*All-in-One*). Diese Variante wurde von Sun bereits für die kommende J2SE Version 1.5 angekündigt. Dies ist mit ein Indiz dafür, welche hohe Priorität diesem Thema eingeräumt wird.

Mit der Verfügbarkeit von J2EE-Implementierungen auf Basis der Isolate-API sollte nichts mehr gegen den Einsatz von Java und J2EE in transaktionsverarbeitenden Systemen jeglicher Art sprechen.

8.1.4 Entwurf des Transaction Testsystems

Die Verfügbarkeit von ausgereiften J2EE-Produkten auf Basis der Isolate-API wird aber wohl noch etwas auf sich warten lassen und - obwohl der Ansatz sehr vielversprechend ist - bleibt vieles eine Implementierungsangelegenheit. Dies wirft die Frage auf, mit welcher Basis ein Entscheidungsträger die Faktoren Kosten und Leistung eines J2EE-Produktes abschätzen und somit zu einer substanziellen Entscheidung kommen kann, welches J2EE-Produkt für sein Anwendungsszenario am besten geeignet ist. Des weiteren muss auch sichergestellt werden, dass die Anwendungskomponenten stabil und zuverlässig funktionieren, was durch den relativ komplexen, entkoppelten und verteilten Entwicklungsprozess der J2EE-Anwendung recht schwierig ist.

Ein J2EE-Produkt ist für ein Anwendungsszenario genau dann einsetzbar, wenn es für alle Anwendungsfälle des Szenarios unter den definierten Rahmenparametern (Menge der parallelen Transaktionen, Gesamtmenge von Transaktionen, maximale Durchlaufzeit) die Inputparameter auf das erwartete Ergebnis abbildet. Um diesen Sachverhalt sicherzustellen, lassen

sich die formale Verifikation des Quellcodes, Expertenempfehlungen oder die Evaluierung durch Tests benutzen. Die formale Verifikation ist in der Praxis wegen Zeit- und Kostengründen kaum anwendbar. Der Ansatz der Experten-Empfehlung wird trotz subjektiver Sichtweise und Fehlerträchtigkeit am häufigsten benutzt. Der dritte Ansatz des Tests von J2EE-Infrastrukturen und Anwendungen kommt aus Gründen von fehlendem Wissen, mangelnder Zeit oder knappem Budget meistens zu kurz, würde aber - abgesehen von der formalen Verifikation - die besten Ergebnisse liefern. Die Loslösung der Entwicklung eines solchen Testsystems von dem tatsächlichen Bedarf der Entscheidungsfindung innerhalb eines Projektes, erscheint der Schlüssel für eine Lösung der Problematik dieses Ansatzes zu sein. Bei Sammlung der Testergebnisse und deren Veröffentlichung müsste nur einmal in den Test einer J2EE-Infrastruktur investiert werden, während die Anwendungsfälle wiederverwendet werden könnten.

Aus den Anforderungen an das Testsystem wurde im Rahmen dieser Arbeit das *Transaction Testsystem (TTS)*, bestehend aus dem *TestController-System (TCS)*, dem *System under Test (SUT)* und einer Reihe von Test- und Fehlerfällen, entwickelt. Das TCS beinhaltet den `TestClient`, welcher ein Tester benutzt um eine Testsuite, bestehend aus mehreren Testfällen, zu erstellen und dem `TestController` zu übergeben. Der `TestController` ist ein Dienst innerhalb eines J2EE-Containers und stellt Schnittstellen zum Ausführen und Interagieren mit einer Testsuite bereit. Der `TestController` verwaltet die Testprozesse zu einem Testfall und übergibt diese zur Ausführung auf dem SUT dem `ProcessManager`. Ebenfalls steuert er die Basislast gegen das SUT während der Ausführung eines Testprozesses.

Der `ProcessManager` ist die Integrationsschicht zwischen dem TTS und einer beliebigen zu testenden J2EE-Anwendung. Während der Ausführung des Testprozesses kommuniziert der `ProcessManager` an bestimmten Synchronisationspunkten mit dem `TestController`, um sich mit den parallel ausgeführten Testprozessen desselben Testfalls zu synchronisieren. Die Synchronisation wird vom `TestController` durchgeführt. Wenn ein Testprozess den für ihn vorgesehenen Synchronisationspunkt erreicht hat, wird der Methodenaufruf vom `TestController` intern blockiert, bis die restlichen Testprozesse ebenfalls ihre Synchronisationspunkte erreicht haben. Wenn alle Testprozesse ihren Synchronisationspunkt erreicht haben, kehren die Testprozesse in einer definierten Reihenfolge zurück. Bei der Rückkehr der Testprozesse können Fehlerfälle im SUT ausgelöst werden. Bei Fehlern der Fehlerkategorie ERROR (z.B. `NullPointerException`), darf nur der fehlerauslösende Testprozess fehlschlagen und ein Rollback durchführen, alle anderen Testprozesse müssen erfolgreich sein. Bei der Fehlerkategorie FATAL (z.B. *Systemcrash*) laufen alle aktiven Testprozesse auf einen Fehler. Das System muss evtl. neu gestartet und auf dem *Resourcemanager* ein *Recovery* durchgeführt werden.

Nach der Ausführung eines Testprozesses startet das TCS die Überprüfung der Korrektheit der Ergebnisse des Testprozesses sowie die Konsistenzprüfung der Anwendungsdatenbank. Wenn das SUT durch die Ausführung eines FATAL-Fehlers zeitweise nicht verfügbar ist, wird die Validierung solange verzögert. Zuletzt werden die Daten für die Testmetrik und die Vergleichsmetrik ausgewertet und in die Metrik-Datenbank des TCS geschrieben.

8.1.5 Testdurchführung und Ergebnisse

Um die Eignung des TTS sowohl für die Auswahl einer für das Anwendungsszenario geeigneten J2EE-Infrastruktur als auch für die Sicherstellung von stabilen und verlässlichen Anwendungskomponenten zu untersuchen, wurde das TTS im Rahmen dieser Arbeit mit zwei Varianten einer J2EE-Infrastruktur (Testsetups) auf dem SUT getestet. Testsetup 1, bestehend aus dem JBoss Applicationserver und der MaxDB als Datenbank und Testsetup 2, bestehend aus IBM Websphere und DB2 als Datenbank. Der TestController des TCS wurde für die Tests auf einer JBoss-Infrastruktur, mit Nutzung von optimistischen Synchronisationsverfahren und mit der Persistierung über die MaxDB, betrieben. Auf dem SUT wurde ein pessimistisches Sperrverfahren eingesetzt.

Um das Verhalten der J2EE-Infrastrukturen mit unterschiedlichen Konfigurationen und unter verschiedener Last zu untersuchen, wurden alle Testszenarien wiederholt und mit unterschiedlichen Rahmenparametern durchgeführt: ohne Basislast mit Überprüfung aller Transaktionen und nur Fehler der Kategorie ERROR (Testkategorie 1), mit Basislast von zehn parallelen Transaktionen mit Überprüfung aller Transaktionen und nur Fehler der Kategorie ERROR (Testkategorie 2), mit Basislast von zehn parallelen Transaktionen ohne Überprüfung der Basislast-Transaktionen und nur Fehler der Kategorie ERROR (Testkategorie 3), mit Basislast von hundert parallelen Transaktionen ohne Überprüfung der Basislast-Transaktionen und nur Fehler der Kategorie ERROR (Testkategorie 4) und mit Basislast von zehn parallelen Transaktionen ohne Überprüfung der Basislast-Transaktionen und nur Fehler der Kategorie FATAL (Testkategorie 5).

Websphere als marktführende Implementierung der J2EE-Spezifikation sowie DB2 haben die Erwartungen an Stabilität und Transaktionssicherheit auch bei fatalen Fehlern im vollen Umfang erfüllt. Störend haben sich während der Tests nur das relativ komplizierte Verfahren zum Aufruf eines Websphere-Dienstes von einer separaten Java-Anwendung erwiesen und dass es nicht ohne Client-Neustart möglich war, die Verbindung zu Websphere nach einem SUT-Neustart wiederherzustellen (*Reconnect*). Überraschender war die Feststellung, dass die Opensource-Variante aus JBoss und MaxDB in den Tests ebenfalls selbst mit fatalen Fehlern zurecht kam und die Stabilität und Transaktionssicherheit im selben Maße wie die kommerzielle Websphere-Implementierung gewährleistete. Des weiteren stellte weder ein *Reconnect*

noch der Aufruf von einer externen Java-Anwendung ein Problem dar. Auch die Performanz und der Durchsatz waren während der Tests durchaus vergleichbar, wenn auch Websphere in allen Kategorien bessere Werte gezeigt hat. Vor allem bei steigender Nebenläufigkeit und dem Setzen von Sperren auf denselben Ressourcen skaliert Websphere besser, was wohl hauptsächlich an der Vermeidung der geschilderten dynamischen Aspekte von JBoss und der effektiveren Ressourcennutzung von Websphere liegen dürfte.

Obwohl die Tests damit zu demselben positiven Ergebnis für die Gewährleistung von Transaktionssicherheit auf beiden J2EE-Infrastrukturen gelangen, wird diese bei genauerer Betrachtung auf vollkommen unterschiedliche Weise realisiert. Bei JBoss liegt die Verantwortung bei der Datenbank, somit sind die guten Testergebnisse der guten Implementierung der MaxDB zu verdanken. Würde ein anderes DBMS eingesetzt werden, welches Probleme bei der Gewährleistung der Transaktionssicherheit hat, beispielsweise beim *Recovery* nach einem Systemfehler, hätten die Testergebnisse wahrscheinlich anders ausgesehen. Die akzeptablen Performanzdaten sind dagegen ein Verdienst von JBoss, welche jedoch durch die leichtgewichtige Realisierung des Transaktionsdienstes begünstigt werden. Bei Websphere liegt die Verantwortung bei dem eigenen Transaktionsdienst, da Websphere selbst *Recovery*, erweitertes *Locking* und ausgefeilte Fehlerbehandlungen realisiert und damit dem DBMS viele Aufgaben abnimmt.

Um Aussagen über die Eignung des TTS für die Gewährleistung von stabilen und zuverlässigen J2EE-Anwendungen zu erhalten, wurden bewusst Fehler in die J2EE-Anwendung des SUT eingebaut. Das Ergebnis war, dass mit dem TTS sämtliche bewusst eingebauten Fehler sowie einige weitere Fehler aus den Bereichen Konfiguration, Nebenläufigkeit, *Locking* und fehlerhafte Ressourcenfreigabe gefunden wurden. Darüber hinaus konnte das TTS auch erfolgreich zur Optimierung der Konfiguration hinsichtlich des Durchsatz eingesetzt werden und zur Sicherstellung der Kompatibilität bei Versionswechsel der J2EE-Infrastruktur-Komponenten. Das TTS erscheint demnach für das Aufspüren von Anwendungs-, Integrations- und Konfigurationsfehlern geeignet zu sein. Besonders für die verlässliche reproduzierbare Teststellung von Nebenläufigkeitsszenarien im Normalbetrieb und unter Fehlereinwirkung, spielt das TTS seine Stärke gegenüber herkömmlichen Modul- oder Lasttest aus.

Mit Sicherheit hat jedes Unternehmen und jede Behörde unterschiedliche Anforderungen an ein J2EE-Produkt sowohl aus fachlicher, technischer als auch wirtschaftlicher Sicht. Mit der Bereitstellung von Bewertungsmetriken auf Basis der bekannten Anwendungsfälle und der im TTS gesammelten Daten, kann die Entscheidung für die Eignung eines J2EE-Produktes mit mehr Qualität und Substanz getroffen werden. Eine Detailmetrik gibt dabei jeden Testfall auf das genaueste wieder (anhand dreißig Werten aus den Bereichen technische Fehler, fachliche Fehler, Ausführungsdauer, Konsistenz und Anzahl modifizierter Datensätzen). Eine konsolidierte Vergleichsmetrik aus nur fünf Werten eignet sich für den Vergleich zwischen J2EE-Produkten oder unterschiedlichen Produktkonfigurationen.

Durch die positiven Testergebnisse belegt, kann das TTS zur Reduzierung des Entwicklungsrisikos bei der Durchführung eines J2EE-Projektes eingesetzt werden. Mit dem T-Modell als Entwicklungsprozess fördert es die schnellere und qualitativ hochwertigere Entwicklung von stabilen und verlässlichen J2EE-Systemen. Das T-Modell startet mit der Entscheidung des Managements, eine J2EE-Anwendung zu entwickeln. Die Infrastrukturbetreuung, welche die zukünftige J2EE-Infrastruktur installiert, konfiguriert und betreut, stellt eine möglichst produktionsnahe J2EE-Infrastruktur auf einem dedizierten Testsystem bereit.Die Anwendungsentwicklung entwickelt nach Maßgabe von fachlichen Anforderungen die J2EE-Anwendung, welche später in der Produktion eingesetzt werden soll und stellt die Anwendung für das Testsystem bereit. Für das Testsetup wird in der Setupphase die Konfiguration der J2EE-Infrastruktur, für die Bedürfnisse der J2EE-Anwendung (z.B. *Datasource*-Definitionen, Sicherheitseinstellungen) ebenso wie das Zusammenstellen der Test-, Validierungs- und Fehlerfälle, durchgeführt. Die anschließende Simulationsphase führt die Simulation der Unternehmensprozesse auf Basis der in der Setupphase definierten Testkonfigurationen durch. Die während der Testdurchführung gesammelten Daten werden der darauf folgenden Auswertungsphase zur Verfügung gestellt, welche die Detail- und Vergleichsmetriken erstellt.

Die detaillierten Testmetrikdaten sind vor allem für die Anwendungsentwicklung wichtig, um Problemstellen innerhalb der J2EE-Anwendung zu entdecken und durch geeignete Modifikationen zu lösen. Die Infrastrukturbetreuung zieht aus den Vergleichsmetriken für unterschiedliche Produkte bzw. Produktkonfigurationen den meisten Nutzen, obwohl für ein Detailverständnis die Betrachtung der detaillierten Testmetriken durchaus auch von Vorteil ist. Der Infrastruktursupport kann somit Fehler in der Konfiguration erkennen bzw. Produktkonfigurationen anhand der gelieferten Daten optimieren. Das Management schließlich, dürfte ausschließlich Interesse an den Vergleichsmetriken und den darin dargestellten Zahlen für erfolgreich durchgeführte Tests pro Produktkonfiguration sowie der relativen Performanz haben. Zusammen mit den Informationen über Kosten der Beschaffung und des Betriebs der J2EE-Infrastruktur, lässt sich eine Kosten-/Leistungs-Relation für eine optimale Entscheidung herleiten.

8.2 Schlussfolgerung

Verlässliche Transaktionsverarbeitung ist notwendig, aber komplex, schwierig und fehleranfällig zu realisieren. Trotz der momentan noch existierenden Isolationsproblematiken der Beeinflussung und Zuverlässigkeit von Transaktionen in J2EE-Systemen, lassen sich durch Tests und Qualitätssicherungsmaßnahmen stabile und verlässliche Anwendungen erreichen.

Die Qualität von J2EE-Anwendungen lässt sich durch den Einsatz des TTS deutlich steigern, bei gleichzeitiger Senkung der Testkosten durch Benutzung einer vorhandenen Infrastruktur.

Die Isolationsproblematik der Beeinflussung und Zuverlässigkeit ist dabei in keiner Weise ein Problem von Java oder J2EE. CICS, als etablierter und weit verbreiteter Transaktionsmonitor, wird gemäß [Herr03] und [Gray93] mit Nucleus, Diensten und Anwendungsprogrammen in einem Betriebssystemprozess ausgeführt, welcher auf einem Adressraum arbeitet. Fehlerhafte Anwendungsprogramme können somit Auswirkungen auf den gesamten CICS-Prozess und alle aktuell ausgeführten Transaktionen haben. Dabei sind die Anwendungen meist in den Sprachen Cobol, C oder C++ geschrieben, bei denen der Anwendungsentwickler die Verantwortung für Allokierung und Deallokierung von Speicher trägt und Fehler zu Instabilitäten oder ungewollter Modifikation von Speicherinhalten führen können. Dies ist eine Fehlerquelle, welche in Java nicht mehr gegeben ist, denn hier kann nur noch über die Verwendung von statischen Klassenvariablen die Isolationsproblematik Beeinflussung vorkommen. Diese Vorkommen lassen sich recht einfach mit Qualitätssicherungsmaßnahmen finden und auf Gefahrenpotential analysieren. Die J2EE-Plattform leistet ihr übriges, um Transaktionsverarbeitung mit derselben Sicherheit zu betreiben wie auf einem CICS-System. Ganz nach dem Vorbild von CICS lagert sie die kritische Funktionalität für Dienste, wie beispielsweise die Transaktionsverarbeitung und den Zugriff auf externe Ressourcen, in die „vertrauenswürdige“ Infrastruktur aus und damit in die Verantwortung des Produktherstellers.

Mit der Verfügbarkeit von J2EE-Produkten auf Basis der Isolate-API ist eine weitere Reduzierung der Isolationsproblematik Beeinflussung und Zuverlässigkeit gegeben. Die Tests auf den J2EE-Infrastrukturen JBoss/MaxDB und Websphere/DB2 haben ebenfalls gezeigt, dass schon heute existierende J2EE-Systeme die Transaktionssicherheit, selbst bei fatalen Fehlern, gewährleisten. Die Erstellung von stabilen und zuverlässigen Anwendungen kann durch das TTS sichergestellt werden. Mit diesen Maßnahmen lassen sich J2EE-Anwendungen mit einem ähnlich hohen Grad an Sicherheit ausführen wie herkömmliche CICS Anwendungen.

Ein Performanzvergleich von CICS und J2EE-Systemen könnte sich noch zugunsten von CICS darstellen, da CICS unbestreitbar einen längeren Entwicklungszyklus absolviert hat und damit umfangreicher optimiert werden konnte. Doch dies dürfte nicht für jede Anwendung ausschlaggebend sein und auch nicht von Dauer.

Generell haben die Tests gezeigt, dass auch die kostengünstige Opensource-Variante aus JBoss und MaxDB dieselbe Transaktionssicherheit wie Websphere leistet, bei der Durchführung von lokalen Transaktionen, wenn auch mit völlig anderer Verteilung der Verantwortung. JBoss lagert die Verantwortung auf das DBMS aus, während Websphere die Transaktionsverarbeitung hauptsächlich selbst realisiert und zusammen mit dem DBMS lediglich das Setzen von Sperren und die Festschreibung der Daten realisiert. Die daraus resultierende, bessere Ressourcennutzung von Websphere bringt zum einen Performanzvorteile und zum anderen ermöglicht sie es Websphere, unter Last besser zu skalieren. Ob diese Ergebnisse auf andere

Teststellungen (z.B. verteilte Transaktionen) übertragen werden können, müssen weitere Untersuchungen zeigen.

8.3 Ausblick

Diese Arbeit hat mit der detaillierten Analyse der J2EE-Spezifikation und der beiden J2EE-Implementierungen JBoss und Websphere eine Basis für weitere Untersuchungen gelegt. Das im Rahmen dieser Arbeit entwickelte Testsystem für J2EE-Systeme (TTS) kann im weiteren dazu benutzt werden, um die Tests auf weitere J2EE-Produkte auszudehnen und erweiterte Testszenarien hinzuzufügen. Die folgenden Themen sollten noch untersucht werden:

- In dieser Arbeit wurden die Tests auf die Durchführung von lokalen Transaktionen beschränkt. Interessant wären Ergebnisse einer Untersuchung derselben J2EE-Infrastrukturen auf ihr Verhalten bei verteilten Transaktionen.
- Ebenfalls wurden die Performanz-Messungen in dieser Arbeit lediglich zum Zwecke qualitativer Aussagen durchgeführt. Interessant wäre eine Konzentration auf die Erzielung von quantitativen Performanzaussagen bzgl. J2EE-Infrastrukturen untereinander oder auch im Vergleich zu Transaktionsmonitoren.
- In dieser Arbeit wurden lediglich die beiden J2EE-Implementierungen JBoss und IBM Websphere untersucht. Ein Vergleich mit anderen am Markt existierenden Applicationservern (Bea, Oracle, Orion und Jonas) und Zusammenstellung der Vergleichsmetriken würde eine bessere Orientierung liefern.
- Um den Einfluss der Infrastruktur auf die Testergebnisse zu untersuchen, sollten die Tests mit weiteren unterschiedlichen Testsetups wiederholt werden, beispielsweise auf unterschiedlichen Betriebssystemen (Windows XP, Windows NT, Linux, AIX, HP-UX, z/OS) oder unterschiedlichen JVM-Implementierungen (IBM, Sun, Microsoft).
- Weiterhin ist der Einfluss von Transaktionsmonitoren auf die Entwicklung der J2EE-Produkte zu klären. Beispielsweise ob es zutrifft, dass bewährte Algorithmen aus CICS in Websphere eingesetzt werden (gleiches gilt für Tuxedo und Weblogic von Bea).

- Die J2EE-Plattform wurde entwickelt, um Anwendungskomponenten einfach auf unterschiedlichen J2EE-Infrastrukturen zu installieren, ohne dabei Klassen anpassen oder neu übersetzen zu müssen. Es stellt sich jedoch die Frage, ob standardisierte Dienste der J2EE-Infrastruktur austauschbar sind. JBoss spricht seinen Transaktionsmanager beispielsweise nur über die JTA-Schnittstellen an. Lässt sich auf einfache Art und Weise dieser Transaktionsmanager gegen einen ebenfalls JTA-konformen Transaktionsmanager austauschen? Von Interesse wäre auch eine Untersuchung, welche einen Transaktionsmonitor mittels einer JTA-konformen Hülle integriert und die Auswirkung auf die Testergebnisse beschreibt.
- In dieser Arbeit wurde lediglich der *Resource Adapter* für ein RDBMS untersucht und die Ausführung von lokalen Transaktionen getestet. Es stellt sich die Frage, wie sich weitere transaktionale *Resource Adapter* (*CICS Connector*, *JMS Provider*, SAP R/3 *Gateway*) bzgl. Transaktionssicherheit verhalten.

9 Abkürzungsverzeichnis

2PC	Two Phase Commit
ACID	Atomicity, Consistency, Isolation, Durability
AG	Aktiengesellschaft
AIX	Advanced Interactive Executive (IBM Unix Betriebssystem)
ANSI	American National Standards Institute
API	Application Programming Interface
APP	Application
AWT	Abstract Window Toolkit
BMP	Bean Managed Persistence
BMT	Bean Managed Transaction
BO	Business Object
BOT	Begin of Transaction
BTP	Business Transaction Protocol
BTTC	Business Transaction Technical Commitee
CICS	Customer Information Control System
CMP	Container Managed Persistence
CMR	Container Managed Relation
CMT	Container Managed Transaction
COBOL	Common Business Oriented Language
CORBA	Common Object Request Broker Architecture
CRM	Communication Resourcemanager
DB2	Database 2 (IBM Datenbank)
DBMS	Datenbank Management System
DDL	Data Definition Language (SQL)
DLL	Dynamic Link Library
EAI	Enterprise Application Integration
EAR	Enterprise Archive
EJB	Enterprise JavaBean
EOT	End of Transaction
HP-UX	HP Unix Betriebssystem
HTTP	Hypertext Transfer Protocol

I/O	Input / Output
IBM	Industrial Business Machines
IIOP	Internet Inter ORB Protocol
Inc	Incorporated
IS-Lock	Shared Intention Lock
IX-Lock	Exclusive Intention Lock
J2EE	Java 2 Enterprise Edition
J2SE	Java 2 Standard Edition
JAAS	Java Authentication and Authorisation Services
JAR	Java Archive
JAWS	Just another Web Storage (JBoss OR-Mapper)
JCA	Java Connector Architecture
JCP	Java Community Process
JDBC	Java Database Connectivity
JDK	Java Development Kit
JMS	Java Messaging Service
JMX	Java Management Extension
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRMP	Java Remote Method Protocol
JSP	Java Server Pages
JTA	Java Transaction API
JTS	Java Transaction Service
JVM	Java Virtual Machine
KW	Kalenderwoche
LOC	Lines of Code
MaxDB	MaxDB (früher SapDB). Ursprünglich von der Software AG entwickelt (Adabas D) wurde die Datenbank von SAP aufgekauft und als Opensource-Projekt freigegeben. Seit kurzem ist nun die Firma MySql AB für die Weiterentwicklung zuständig.
MB	Megabyte
MDB	Message-Driven Bean
MS	Microsoft
OIL	Optimized Invocation Layer

OMG	Object Management Group
OR	Object Relational
ORM	Object Relational Mapping
OTS	Object Transaction Service (CORBA)
PRJVM	Persistent Reusable Java Virtual Machine
RDB	Relational Database
RDBMS	Relationales Datenbank Management System
RMI	Remote Method Invocation
RUN	Runtime
S-Lock	Shared Lock
SIX-Lock	Exclusive and Shared Lock
SQL	Structured Query Language
SUT	System under Test
TCP/IP	Transmisson Control Protocol / Internet Protocol
TCS	Test Controller System
TERPH	Terrific ERP for Housewares (Beispiel für transaktionsverarbeitendes System)
TP	Transaction Processing
TPC	Transaction Processing Performance Council
TPC-C	Benchmark zur Simulation von Transaktionsverarbeitung (TPC)
TS	Testsystem
TTS	Transaction Testsystem
TX	Transaction
U-Lock	Update Lock
UOW	Unit of Work
WAD	Websphere Studio for Application Developer
WAR	Web Archive
X-Lock	Exclusive Lock
X/Open XA	Industriestandard für verteilte Transaktionen
XA	vgl. X/Open XA
XML	Extensible Markup Language

10 Literaturverzeichnis

[Arno98] K. Arnold, J. Gosling: *The Java Programming Language*. Addison-Wesley, 1998, ISBN 0201704331.

[Back98] G. Back, P. Tullmann, L. Stoller, W. Hsieh, J. Lepreau: *Java Operation Systems: Design and Implementation*. TR UUCS-98-015, University of Utah, 1998.

[Beck02] K. Beck: *Test Driven Development: By Example*. Addison-Wesley, 2002, ISBN 0321146530.

[Bern97] P. Bernstein, E. Newcomer: *Principles of Transaction Processing*. Morgan Kaufmann, 1997, ISBN 1-55860-415-4.

[Bey04] M. Beyerle: *Architekturanalyse der Java VirtualMachine unter z/OS und Linux*. Universität Tübingen, 2004.

[Borm01] S. Borman, S. Paice, M. Webster, M. Trotter, R. McGuire, A. Stevens,B. Hutchison, R. Berry: *A Serially Reusable Java(tm) Virtual Machine Implementation forHigh Volume, Highly Reliable, Transaction Processing*. TR 29.3406, IBM, 2001.

[CICS02] IBM: *Enterprise JavaBeans for z/OS and OS/390: CICS Transaction Server V2.2*. SG24-6284-01, IBM, 2002.

[Cza00] G. Czajkowski: *Application Isolation in the Java™ Virtual Machine.* http://java.sun.com/j2se/jcp/AppIsolationAPI/oopsla00-czajkowski-final.pdf, Sun, 2000.

[Dar01] I. Darwin: *Java Cookbook*. O'Reilly, 2001, ISBN 0-596-00170-3.

[DeMa97] T. DeMarco: *The Deadline: A Novel about Project Management*. Dorset House Publishing Company, 1997, ISBN 0932633390.

[Dill00] D. Dillenberger, R. Bordawekar, C. Clark, D. Durand, D. Emmes, O. Gohda: *Building a Java virtual machine for server applications: The Jvm on OS/390*. Volume 39, Number 1, IBM Systems Journal, 2000.

[Ecl04] Eclipse: *Hyades - Automated Software Quality Evaluation framework.* http://www.eclipse.org/hyades.

[Eic03] D. Eickstädt, T. Reuhl: *Java mit Open Source-Tools*. Markt+Technik, 2003, ISBN 3-8272-6462-6.

[EJB03] L. DeMichiel et al.: *Enterprise JavaBeans Specification Version 2.1*. Final Release, Sun, 2003.

[Gam03] E. Gamma, K. Beck: *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003, ISBN 0321205758.

[Gam96] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns*. Addison Wesley, 1996, ISBN 0-201-63361-2.

[Gorr04] L. Gorrie: *Echidna - a free Multitask System in Java.* http://www.javagroup.org/echidna.

[Gray93] J. Gray, A. Reuter: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993, ISBN 1-55860-190-2.

[Haw98] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, T. von Eicken: *Implementing Multiple Protection Domains in Java*. USENIX Annual Conference, New Orleans, 1998.

[Herr03] P. Herrmann, U. Kebschull, W. Spruth: *Einführung in z/OS und OS/390*.

Oldenbourg, 2003, ISBN 3-486-27214-4.

[Hor02] E. Horn, T. Reinke: *Softwarearchitektur und Softwarebauelemente*. Hanser, 2002, ISBN 3-446-21300-7.

[Hors00] J. Horswill: *Designing and Programming CICS Applications*. O'Reilly, 2000, ISBN 1-56592-676-5.

[IW04] Evers: *JBoss Application Server gets J2EE-certified*. http://www.infoworld.com/article/04/07/16/HNjbosscert_1.html.

[J2EE03] B. Shannon et al.: *Java 2 PlatformEnterprise Edition Specification, v1.4.* , Sun, 2003.

[Janos04] J. Lepreau, W. Hsieh, J. Carter, M. Flatt, J. Zachary: *The Janos Project*. http://www.cs.utah.edu/flux/janos.

[JBoss04] S. Stark et al.: *JBoss Administration and Development*. Second Edition, JBoss, 2004.

[JSR121] Lea, Soper, Sabin et al.: *Java Application Isolation API*. http://jcp.org/aboutJava/communityprocess/review/jsr121/index.html, JCP, 2002.

[JTA99] S. Cheung, V. Matena: *Java Transaction API (JTA)*. Version 1.0.1, Sun, 1999.

[JTS99] S. Cheung: *Java Transaction Service (JTS)*. Version 1.0, Sun, 1999.

[Jur01] M. Juric, S. Basha, R. Leander, R. Nagappan: *Professional J2EE EAI*. Wrox Press, 2001, ISBN 1-861005-44-X.

[Kla03] H. Klaeren: *Skriptum Softwaretechnik*. Universität Tübingen, 2003.

[Mar02] F. Marinescu: *EJB Design Patterns*. John Wiley & Sons, 2002, ISBN 0-471-20831-0.

[Mas03] V. Massol, T. Husted: *JUnit in Action*. Manning Publications, 2003, ISBN 1930110995.

[McG03] J. McGovern, S. Tyagi, M. Stevens, S. Mathew: *Java Web Services Architecture*. Morgan Kaufmann, 2003, ISBN 1558609008.

[McL02] B. McLaughlin: *Building Java Enterprise Applications, Volume I, Architecture*. O'Reilly, 2002, ISBN 0-596-00123-1.

[Oak99] S. Oaks, H. Wong: *Java Threads, Second Edition*. O'Reilly, 1999, ISBN 1-56592-418-5.

[Ope92] Open Group: *Distributed Transaction Processing: The XA Specification*. X/OpenCompany Ltd, 1992, ISBN 1-85912-057-1.

[Orf98] R. Orfali, D. Harkey: *Client/Server Programming with Java and CORBA, 2nd Edition*. John Wiley & Sons, 1998, ISBN 0-471-24578-X.

[OS04] Open Source Initiative: *Open Source Initiative*. http://www.opensource.org.

[OTS03] OMG: *Transaction Service Specification*. Version 1.4, OMS, 2003.

[Rom01] E. Roman, S. Ambler, T. Jewell, F. Marinescu: *Mastering Enterprise JavaBeans (2nd Edition)*. John Wiley & Sons, 2001, ISBN 0471417114.

[Sha03] S. Shavor, J. D'Anjou, S. Fairbrother, D.Kehn, J. Kellerman, P. McCarthy: *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2003, ISBN 0321159640.

[Shar01] R. Sharma, B. Stearns, T. Ng: *J2EE Connector architecture and enterprise application integration*. Addison Wesley, 2001, ISBN 0-201-77580-8.

[Shi00] J. Shirazi: *Java Performance Tuning*. O'Reilly, 2000, ISBN 0-596-00015-4.

[Sta02] G. Starke: *Effektive Software-Architekturen*. Hanser, 2002, ISBN 3-446-

21998-6.

[Til02] J. Tilly, E. Burke: *Ant: The Definitive Guide*. O'Reilly, 2002, ISBN 0-596-00184-3.

[TPCC03] TPC: *TPC Benchmark™ C - Standard Specification*. Revision 5.2, TPC, 2003.

[Tyr04] Tyrex: *Tyrex Transaktionsmanager*. http://tyrex.exolab.org.

[Wal03] C. Walls, N. Richards, R. Oberg: *XDoclet in Action*. Manning Publications, 2003, ISBN 1932394052.

[Whi00] J. Whitaker: *What is Software Testing? And Why Is It So Hard?*.IEEE Software, 17(1):70, 79, 2000.

[WS04] IBM: *IBM WebSphere Application Server, Version 5*. ftp://ftp.software.ibm.com/software/webserver/appserv/v5/g325-2047-003.pdf, 2004.

Anhang

Anhang A : Produkt-Konfiguration

Dieser Anhang enthält Auszüge von Konfigurationsdateien für den JBoss Applicationserver und IBM Websphere.

A.1 JBoss

A.1.1 Aufruf eines Clients zu einem Stateless Session Bean

```
01    <invoker-proxy-binding>
02       <name>stateless-rmi-invoker</name>
03       <invoker-mbean>jboss:service=invoker,type=jrmp</invoker-mbean>
04       <proxy-factory>org.jboss.proxy.ejb.ProxyFactory</proxy-factory>
05       <proxy-factory-config>
06          <client-interceptors>
07             <home>
08                <interceptor>org.jboss.proxy.ejb.HomeInterceptor</interceptor>
09                <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
10                <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
11                <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
12             </home>
13             <bean>
14                <interceptor>org.jboss.proxy.ejb.StatelessSessionInterceptor</interceptor>
15                <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
16                <interceptor>org.jboss.proxy.TransactionInterceptor</interceptor>
17                <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
18             </bean>
19          </client-interceptors>
20       </proxy-factory-config>
21    </invoker-proxy-binding>
```

Listing 1 Konfiguration für einen Aufruf eines Clients zu einem Stateless SessionBean

A.1.2 Standard Stateless SessionBean-Container

```
01 <container-configuration>
02    <container-name>Standard Stateless SessionBean</container-name>
03    <call-logging>false</call-logging>
04    <invoker-proxy-binding-name>stateless-rmi-invoker</invoker-proxy-binding-name>
05
06    <container-interceptors>
07       <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>
08       <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
09       <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
10       <!-- CMT -->
11       <interceptor transaction="Container">
12          org.jboss.ejb.plugins.TxInterceptorCMT
13       </interceptor>
14       <interceptor transaction="Container" metricsEnabled="true">
15          org.jboss.ejb.plugins.MetricsInterceptor
16       </interceptor>
17       <interceptor transaction="Container">
18          org.jboss.ejb.plugins.StatelessSessionInstanceInterceptor
19       </interceptor>
20       <!-- BMT -->
21       <interceptor transaction="Bean">
22          org.jboss.ejb.plugins.StatelessSessionInstanceInterceptor
23       </interceptor>
```

```
24          <interceptor transaction="Bean">
25          org.jboss.ejb.plugins.TxInterceptorBMT
26       </interceptor>
27       <interceptor transaction="Bean" metricsEnabled="true">
28          org.jboss.ejb.plugins.MetricsInterceptor
29       </interceptor>
30       <interceptor>
31          org.jboss.resource.connectionmanager.CachedConnectionInterceptor
32       </interceptor>
33    </container-interceptors>
34
35    <instance-pool>org.jboss.ejb.plugins.StatelessSessionInstancePool</instance-pool>
36    <instance-cache></instance-cache>
37    <persistence-manager></persistence-manager>
38
39    <container-pool-conf>
40       <MaximumSize>100</MaximumSize>
41    </container-pool-conf>
42 </container-configuration>
```

Listing 2 Konfiguration für den Standard-Stateless-SessionBean-Container

A.1.3 Standard CMP-EntityBean-2.0-Container

```
01 <container-configuration>
02    <container-name>Standard CMP 2.x EntityBean</container-name>
03    <call-logging>false</call-logging>
04    <invoker-proxy-binding-name>entity-rmi-invoker</invoker-proxy-binding-name>
05    <sync-on-commit-only>false</sync-on-commit-only>
06    <insert-after-ejb-post-create>false</insert-after-ejb-post-create>
07
08    <container-interceptors>
09       <interceptor>org.jboss.ejb.plugins.ProxyFactoryFinderInterceptor</interceptor>
10       <interceptor>org.jboss.ejb.plugins.LogInterceptor</interceptor>
11       <interceptor>org.jboss.ejb.plugins.SecurityInterceptor</interceptor>
12       <interceptor>org.jboss.ejb.plugins.TxInterceptorCMT</interceptor>
13       <interceptor metricsEnabled="true">
14          org.jboss.ejb.plugins.MetricsInterceptor
15       </interceptor>
16       <interceptor>org.jboss.ejb.plugins.EntityCreationInterceptor</interceptor>
17       <interceptor>org.jboss.ejb.plugins.EntityLockInterceptor</interceptor>
18       <interceptor>org.jboss.ejb.plugins.EntityInstanceInterceptor</interceptor>
19       <interceptor>org.jboss.ejb.plugins.EntityReentranceInterceptor</interceptor>
20       <interceptor>
21          org.jboss.resource.connectionmanager.CachedConnectionInterceptor
22       </interceptor>
23       <interceptor>org.jboss.ejb.plugins.EntitySynchronizationInterceptor</interceptor>
24       <interceptor>org.jboss.ejb.plugins.cmp.jdbc.JDBCRelationInterceptor</interceptor>
25    </container-interceptors>
26
27    <instance-pool>org.jboss.ejb.plugins.EntityInstancePool</instance-pool>
28    <instance-cache>org.jboss.ejb.plugins.InvalidableEntityInstanceCache</instance-cache>
29    <persistence-manager>
30       org.jboss.ejb.plugins.cmp.jdbc.JDBCStoreManager
31    </persistence-manager>
32    <locking-policy>org.jboss.ejb.plugins.lock.QueuedPessimisticEJBLock</locking-policy>
33
34    <container-cache-conf>
35       <cache-policy>org.jboss.ejb.plugins.LRUEnterpriseContextCachePolicy</cache-policy>
36       <cache-policy-conf>
37          <min-capacity>50</min-capacity>
38          <max-capacity>1000000</max-capacity>
39          <overager-period>300</overager-period>
40          <max-bean-age>600</max-bean-age>
41          <resizer-period>400</resizer-period>
42          <max-cache-miss-period>60</max-cache-miss-period>
43          <min-cache-miss-period>1</min-cache-miss-period>
44          <cache-load-factor>0.75</cache-load-factor>
```

```
45       </cache-policy-conf>
46    </container-cache-conf>
47
48    <container-pool-conf>
49       <MaximumSize>100</MaximumSize>
50    </container-pool-conf>
51
52    <commit-option>B</commit-option>
53 </container-configuration>
```

Listing 3 Konfiguration für den Standard-CMP-EntityBean-2.0-Container

A.2 Websphere

Die Datei `implfactory.properties` beinhaltet voll qualifizierte Klassennamen, welche von Websphere benutzt werden um Dienst-Implementierungen per Konfiguration auszutauschen. Die wichtigsten transaktionsrelevanten Deklaration und ihre Standardwerte sind:

```
#JTA UserTransaction implementation class
com.ibm.ws.transaction.UtxImpl=
          com.ibm.ws.Transaction.JTA.UserTransactionImpl

#JTA TransactionManager implementation class
com.ibm.ws.transaction.WSTxManager=
          com.ibm.ws.Transaction.JTA.TranManagerSet

#JTA Non Recoverable TransactionManager implementation class
com.ibm.ws.transaction.NonRecovWSTxManager=
          com.ibm.ws.Transaction.client.NonRecoverableTranManagerSet

#JTS Current
com.ibm.ws.transaction.TXCurrent=
          com.ibm.ws.Transaction.JTS.CurrentImpl

#Local Transaction Current
com.ibm.ws.transaction.LocalTranCurrent=
          com.ibm.ws.LocalTransaction.LocalTranCurrentSet

#Extended JTA Transaction implementation class
com.ibm.ws.transaction.ExtendedJTATransaction=
          com.ibm.ws.jtaextensions.ExtendedJTATransactionImpl

#Transaction Workload Collaborator implementation class
com.ibm.ws.transaction.WorkloadCollaborator=
          com.ibm.ws.Transaction.JTS.TransactionWorkloadCollaborator
```

Anhang B : Ergebnisse der Anforderungsanalyse

Dieser Anhang beinhaltet die detaillierten Ergebnisse der Anforderungsanalyse für den Entwurf des *Transaction Testsystem (TTS)* in Kapitel 6.

B.1 Datenmodell und Mengengerüst

Als Basis dient das in Abbildung 40 dargestellte TERPH-Datenmodell der *Terrific Housewares AG*, da es von der TPC auch speziell dazu herangezogen wurde, ein praxisrelevantes auftragsverarbeitendes System zu simulieren. Prinzipiell sind zwei Ansätze denkbar für die Realisierung der einzelnen *Warehouses*:

- Möglichkeit 1: die Daten aller Warehouses sind in einer zentralen Datenbasis abgelegt. Es finden nur Transaktionen über einen *Resourcemanager* statt.
- Möglichkeit 2: die Daten sind pro Warehouse in einer separaten Datenbasis abgelegt. Es finden also Transaktionen sowohl über einen *Resourcemanager* als auch über Mehrere statt.

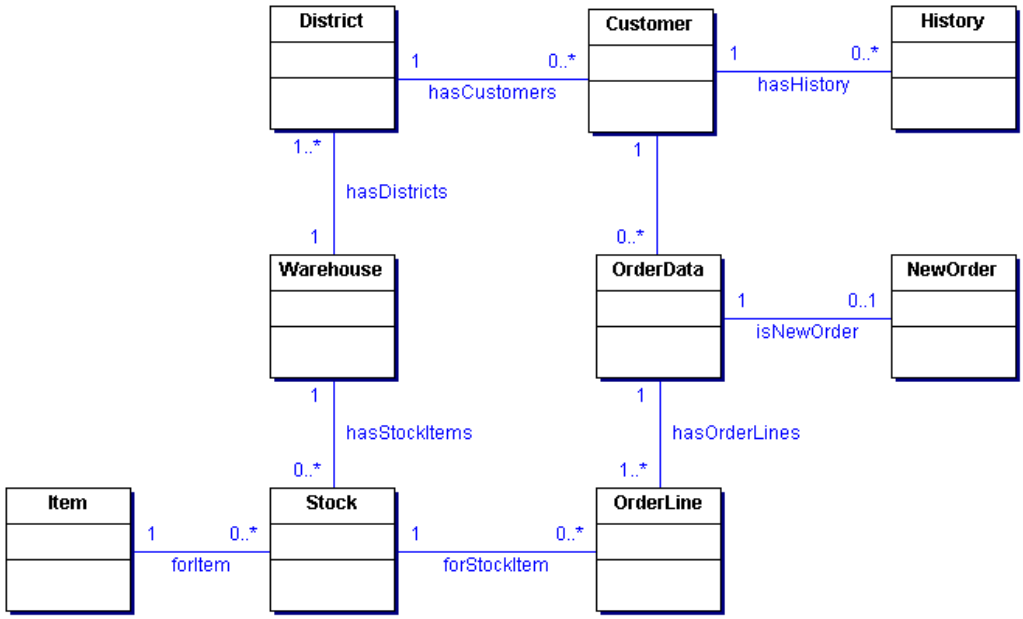

Abbildung 40 Datenmodell von TERPH

Für diese Arbeit genügt es wenn das TTS die Möglichkeit 2 vorsieht. Die konkreten Tests werden mit der Möglichkeit 1 durchgeführt. Die mengenmäßige Skalierung der Datenbasis erfolgt über die Anzahl der *Warehouses*. Da das hauptsächliche Interesse an der Transaktionssicherheit liegt, genügt eine Skalierung von zwei *Warehouses*. (vgl. Tabelle 20).

Tabelle 20 Mengengerüst für die Anwendungsdaten

Entität	*Menge pro Entität*	*Gesamtmenge*
Warehouse	2	2
District	10 pro Warehouse	20
Customer	3000 pro District	60000
Items	100000	100000
Stock	100000 pro Warehouse	200000
Order	3000 pro District	60000
OrderLine	Random(5,15)[2] pro Order	600000

B.2 Konsistenzkriterien

Das vorgestellte Datenmodell muss die folgenden Konsistenzkriterien erfüllen:

1. Die Datensätze von *District*, *OrderData* und *NewOrder* müssen der Bedingung genügen:

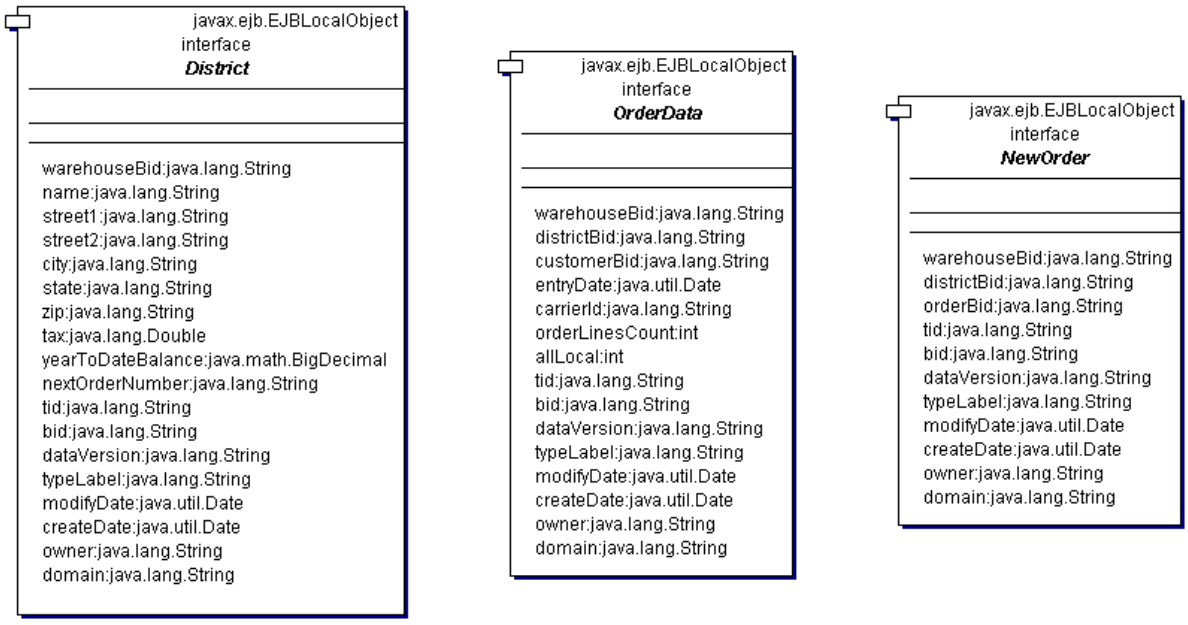

Abbildung 41 Entitäten zu District, OrderData und NewOrder

```
District::nextOrderNumber - 1 =
                max(OrderData::bid) = max(NewOrder::bid)
```

2 Random(x,y) ist eine Funktion auf Basis der `Random`-Klasse von Java. Die Funktion wird benutzt um eine zufallsbedingte Auswahl einer Zahl zwischen x und y zu simulieren.

mit den Beziehungen:

```
District::warehouseBid =
OrderData::warehouseBid =
NewOrder::warehouseBid

District::bid =
OrderData::districtBid =
NewOrder::districtBid
```

Die Attribute `nextOrderNumber` und `bid` sind zwar im Datenmodell (Abbildung 41) als `String` modelliert, beinhalten jedoch nur numerische Werte.

2. Die *NewOrder*-Datensätze müssen der Bedingung genügen:

```
max(NewOrder::bid) – min(NewOrder::bid) + 1 =
     [Anzahl Datensätze von NewOrder für diesen District]
```

mit den Beziehungen:

```
NewOrder::warehouseBid =
NewOrder::warehouseBid

NewOrder::districtBid =
NewOrder::districtBid
```

3. Die Datensätze von *OrderData* und *OrderLine* (Abbildung 42) müssen der Bedingung genügen:

```
sum(OrderData::orderLinesCount)
     = [Anzahl der Datensätze von OrderLine pro District]
```

mit den Beziehungen:

```
OrderData::warehouseBid =
OrderLine::warehouseBid

OrderData::districtBid =
OrderLine::districtBid
```

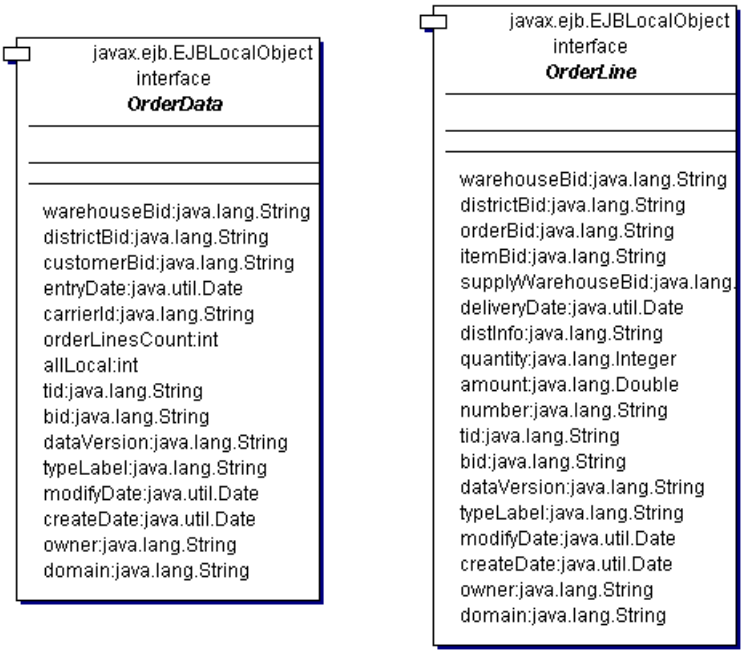

Abbildung 42 Entitäten für OrderData und OrderLine

4. Für jeden Datensatz in *OrderData* gilt:

   `OrderData::carrierId` ist auf den Wert „null“ gesetzt, genau dann, wenn der einen korrespondierenden *NewOrder* Datensatz hat mit den Beziehungen:

   ```
   OrderData::warehouseBid =
   NewOrder::warehouseBid

   OrderData::districtBid =
   NewOrder::districtBid

   OrderData::bid =
   NewOrder::orderBid
   ```

5. Für jeden Datensatz in *OrderData* gilt:

   das Feld `OrderData::orderLinesCount` muss gleich der Anzahl der Datensätze von *OrderLine* sein, welche die Beziehung haben:

   ```
   OrderData::warehouseBid =
   OrderLine::warehouseBid
   ```

```
OrderData::districtBid =
OrderLine::districtBid

OrderData::bid =
OrderLine::orderBid
```

6. Für jeden *OrderLine* Datensatz gilt:

   das Feld `OrderLine::deliveryDate` muss auf „null“ gesetzt werden, genau dann, wenn ein korrespondierender *OrderData* Datensatz existiert mit dem Wert „null“ für `OrderData::carrierId`. Für *OrderData* und *OrderLine* müssen die folgenden Beziehungen erfüllen:

```
OrderData::warehouseBid =
OrderLine::warehouseBid

OrderData::districtBid =
OrderLine::districtBid

OrderData::bid =
OrderLine::orderBid
```

B.3 Testkategorien

Das Testsystem muss es ermöglichen, Testfälle mit und ohne Basislast durchführen zu können. Die Testkategorien sind an der Einhaltung der ACID-Kriterien ausgelegt. Für die Tests muss das Testsystem es ermöglichen an definierten Stellen Fehlersituationen auszulösen und die Ergebnisse anschließend auszuwerten. Folgende Testkategorien sollen möglich sein:

- Sicherstellung von Atomarität im Normalfall und bei Fehlerfall.
- Korrekte Isolation von Lese- / Schreibkonflikten im Normalfall und bei Fehlerfall.
- Korrekte Isolation von Schreib- / Schreibkonflikten im Normalfall und bei Fehlerfall.
- Korrekte Isolation von Repeatable Read Konflikten im Normalfall und bei Fehlerfall
- Korrekte Isolation von Phantomkonflikten

Die Konsistenz der Daten muss dabei nach jedem Testfall geprüft werden und über die gesamte Testausführung gesichert sein. Das Kriterium der Dauerhaftigkeit soll mit entsprechenden Fehlerfällen überprüft werden.

B.4 Geschäftsvorfälle

Für die Ausführung der Testfälle sind Anwendungs-Funktionen notwendig welche auf dem definierten Datenmodell arbeiten. Bei der folgenden Untersuchung wird versucht, möglichst viele Testfälle auf dieselben Funktionen zurückzuführen. Damit kann der Aufwand für die Integration von bestehenden Anwendungen in das Testsystem gering gehalten werden.

Für die Tests auf Atomarität wird eine komplexere Transaktion benötigt, an der Lese und Schreibzugriffe auf unterschiedliche Entitäten stattfinden. Hierfür ist die *NewOrder*-Funktionalität, welche schon in den vorherigen Kapiteln des öfteren angeführt wurde, geeignet. Über die *NewOrder*-Transaktion wird ein kompletter neuer Auftrag innerhalb einer Transaktion auf die Datenbank geschrieben. Innerhalb der Transaktion werden einzelne Daten aus den Stammdaten (*Item*, *Stock*, *Warehouse*, *District*) gelesen und im Rahmen des Auftrags (*OrderData, NewOrder*) und der Auftragszeilen (*OrderLine*) gespeichert. Die *NewOrder*-Transaktion ist eine mittelgewichtige Schreib/Lese-Operation mit einer hohen Transaktionsrate in der Praxis. Es ist eine Online-Transaktion welche eine Antwort an den Benutzer zurückliefern muss. Eine kurze Antwortzeit ist notwendig.

Für die Tests auf Isolation von Schreib/Lese-Konflikten kann ebenfalls die *NewOrder*-Transaktion für die schreibende Transaktion herangezogen werden. Jedoch benötigt man noch eine rein lesende Transaktion welche auf denselben Entitäten operieren kann. Hierfür ist die *OrderStatus*-Funktionalität gut geeignet. Die *OrderStatus*-Transaktion fragt den Status des letzten Auftrags des Kunden ab. Es ist eine mittelgewichtige, nur lesende Operation mit in der Praxis nur niedriger Transaktionsrate. Es ist ebenfalls eine Online-Transaktion und benötigt kurze Antwortzeiten. Für die Tests auf Isolation bei Schreib-Schreib-Konflikten kann die *NewOrder*-Transaktion für beide in Konflikt tretende Transaktionen benutzt werden.

Für die Tests auf Isolation von *Repeatable Read*-Konflikten kann wieder als Basis die *NewOrder*-Transaktion herangezogen werden. Es wird jedoch noch eine weitere Funktionalität benötigt, welche Teile der Stammdaten eines Auftrages abfragen und verändern kann. Hierfür ist eine *Pricing*- sowie eine *PriceAcquisition*-Funktionalität geeignet. Die *Pricing*-Funktionalität ermittelt zu einem gegebenen *Item* den Preis. Es ist eine leichtgewichtige nur lesende Operation mit in der Praxis hoher Transaktionsrate, da es für alle *NewOrder*-Transaktionen durchlaufen werden muss. Die *PriceAcquisition*-Funktionalität modifiziert den Preis eines *Items*. Es ist eine leichtgewichtige lesende und schreibende Operation mit in der Praxis nur niedriger Transaktionsrate.

Für die Tests auf Isolation von Phantom-Konflikten kann wieder die beiden auftragsbezogenen Funktionalitäten *NewOrder* und *OrderStatus* benutzt werden.

B.5 Isolationskriterien

Die Geschäftsvorfälle *NewOrder* und *OrderStatus* operieren auf denselben Daten. Die Transaktionen sollen stark isoliert sein voneinander und sollen deshalb zueinander die Isolationsstufe *Repeatable Read* haben. Die Stammdaten-Funktionalitäten *Pricing* und *Price-Acquisition* sollen mit *Read Commited* von den auftragsverarbeitenden Transaktionen separiert werden. Dieser Sachverhalt wird in Tabelle 21 zusammengefasst.

Tabelle 21 Isolationsmatrix für das Testsystem

	NewOrder	***Order Status***	***Pricing***	***Price Acquisation***	***Andere***
NewOrder	Repeatable Read	Repeatable Read	Read Commited	Read Commited	Read Commited
OrderStatus	Repeatable Read	Repeatable Read	Read Commited	Read Commited	Read Commited
Pricing	Read Commited	Read Commited	Read Commited	Read Commited	Read Commited
Price Acquisation	Read Commited	Read Commited	Read Commited	Read Commited	Read Commited
Andere	Read Commited	Read Commited	Read Commited	Read Commited	Read Commited

B.6 Fehlerszenarien

In Tabelle 22 werden die Fehlersituationen zusammengefasst, welche durch das Testsystem ausgelöst werden müssen. Es werden dabei Kategorien gebildet, welche sich auf den Einfluss des Fehlers auf das System beziehen:

- APP / ERROR: dies ist ein Fehler der Anwendung selbst. Dieser Fehler muss von der Anwendungskomponente gehandhabt werden. Es darf kein expliziter Rollback durch den Container erfolgen.
- RUN / ERROR: ein Laufzeitfehler welcher sich nur auf die aktuelle Transaktion auswirken darf. Hierfür muss der Container ein Rollback durchführen.
- RUN / FATAL: ein Laufzeitfehler welcher sich auf alle aktiven Transaktionen auswirkt. In diesem Fall muss das System neu gestartet werden und evtl. ein *Recovery* auf den betroffenen *Resourcemanagers* durchgeführt werden.

Tabelle 22 Zusammenfassung der Fehlersituationen

ID	***Kategorie***	***Fehler***	***Beschreibung***
1	APP / ERROR	FinderForPrimaryKey HasNoResult	Suche nach einem Primärschlüssel lieferte keine Entität zurück.
2	APP / ERROR	FinderForCompound-KeyHasNoResult	Suche nach einem zusammengesetzten Fremd-schlüssel lieferte keine Entität zurück.
3	APP / ERROR	DuplicateRecords-Found	Suche nach einem eindeutigen Schlüssel (z.B. Primärschlüssel oder eindeutiger Fremdschlüssel) lieferte mehrere Entitäten zurück.
4	RUN / FATAL	DatabaseNotAvailable	Die Verbindung zur Datenbank wurde beendet.
5	RUN / ERROR	DatabaseInternalErro-rInCall	Während der Ausführung eines SQL-Statements geschah ein interner Fehler in der Datenbank, wel-cher ein Shutdown und Restart zur Folge hat.
6	RUN / ERROR	RuntimeException-ForData	Bei der Auswertung der Daten ist ein Laufzeitfehler aufgetreten (z.B. NullPointer-Exception).
7	RUN / ERROR	RuntimeException-ForMemory	Eine Transaktion beansprucht (z.B. durch eine re-kursive Endlosschleife) den gesamten Heap für sich. Es kommt zu einem OutOfMemory-Fehler.
8	RUN / ERROR	RuntimeException-ForJNI	Innerhalb eines JNI-Aufrufs kommt es zu einem Fehler.
9	RUN / FATAL	RuntimeException ForContainer	Während der Ausführung einer Transaktion wird der Container abrupt beendet.
10	RUN / FATAL	ShutdownOfContainer	Während der Ausführung einer Transaktion wird der Container durch ein „Shutdown“-Kommando kontrolliert beendet.
11	RUN / FA-TAL	RuntimeException ForMachine	Während der Ausführung einer Transaktion wird die Maschine anormal beendet (Stromausfall, Hard-waredefekt)
12	RUN / FA-TAL	RuntimeException ForAllMemory	Während der Ausführung einer Transaktion wird Hauptspeicher und Festplatte korrumpiert. Daten-bank muss neu aufgesetzt werden und über Recove-ry-Logs wiederhergestellt werden.

B.7 Testszenarien

In diesem Kapitel werden die relevanten Testszenarien aufgelistet welche das Testsystem durchführen und auswerten können muss. Es wird in der folgenden Tabelle 23 neben dem Namen des Testfalls, der detaillierte Ablauf beschrieben sowie die Stelle an der ein Fehler ausgelöst werden soll. Dieser Fehler soll stellvertretend für die Fehlerszenarien aus dem Anhang B.6 sein. Für jeden Testfall wird auch noch das erwartete Ergebnis beschrieben.

Der Begriff *Systemreset* bezieht sich auf eine Aktion in Folge eines Fehlers aus der Kategorie FATAL der auf alle aktiven Transaktionen eine Auswirkung hat. Bei einem *Systemreset* werden die geschädigten Ressourcen neu erzeugt und initialisiert, beispielsweise ein EJB-Container neu gestartet oder die Datenbank wieder verfügbar gemacht.

Tabelle 23 Testszenarien der Anforderungsanalyse

ID	***Testfall***	***Details***	***Fehlerfall***	***Ergebnis***
1	Atomarität	Ausführen einer NewOrder-Transaktion T1 für einen Kunden K1.		1. Datensatz für ORDER, ORDERLINE, NEWORDER muss entsprechend geändert worden sein. 2. Konsistenzbedingungen müssen erfüllt sein.
2	Atomarität bei Fehler	1. Ausführen einer NewOrder-Transaktion T1 für einen Kunden K1. 2. T1 unmittelbar vor dem Commit stoppen. 3. In T1 tritt Fehler auf (Rollback).	Alle	1. Datensatz für ORDER, ORDERLINE, NEWORDER muss ursprüngliche Werte enthalten. 2. Konsistenzbedingungen müssen erfüllt sein.

ID	*Testfall*	*Details*	*Fehlerfall*	*Ergebnis*
3	Isolation von Lese/Schreib Konflikten	1. NewOrder-Transaktion T1 starten. 2. T1 unmittelbar vor dem Commit stoppen. 3. OrderStatus-Transaktion T2 für den Kunden aus T1 starten. 4. Beim Versuch auf den Kunden zuzugreifen muss T2 warten. 5. Auf T1 Commit durchführen. 6. T2 sollte nun weiterlaufen und auch den Commit durchführen.		1. T2 muss als letzten Auftrag für den Kunden den Auftrag aus T1 zurückgeben. 2. T2 muss gewartet haben (implizit gegeben, da sonst letzter Auftrag nicht gleich T1-Auftrag wäre). 3. Konsistenzbedingungen müssen erfüllt sein.
4	Isolation von Lese/Schreib Konflikten bei Fehlern.	1. OrderStatus-Transaktion T0 für Kunde K1 durchführen. 2. NewOrder-Transaktion T1 für K1 starten. 3. T1 unmittelbar vor dem Commit stoppen. 4. OrderStatus-Transaktion T2 für den K1 starten. 5. Beim Versuch auf den Kunden zuzugreifen muss T2 warten. 6. In T1 tritt Fehler auf (-> Rollback von T1). 7. T2 sollte nun weiterlaufen und auch den Commit durchführen.	APP / ERROR, RUN / ERROR	1. Ergebnis von T2 muss mit T0 übereinstimmen. 2. T2 muss gewartet haben. 3. Konsistenzbedingungen müssen erfüllt sein.

ID	*Testfall*	*Details*	*Fehlerfall*	*Ergebnis*
5	Isolation von Lese/Schreib Konflikten bei Fehlern.	1. OrderStatus-Transaktion T0 für Kunde K1 durchführen. 2. NewOrder-Transaktion T1 für K1 starten. 3. T1 unmittelbar vor dem Commit stoppen. 4. OrderStatus-Transaktion T2 für den K1 starten. 5. Beim Versuch auf den Kunden zuzugreifen muss T2 warten. 6. In T1 tritt Fehler auf . 7. T2 läuft ebenfalls auf Fehler. 8. Systemreset 9. OrderStatus-Transaktion T3 für Kunde K1 durchführen.	RUN / FATAL	1. Ergebnis von T3 muss mit T0 übereinstimmen. 2. T2 muss gewartet haben. 3. Konsistenzbedingungen müssen erfüllt sein. 4. Alle Transaktionen welche erfolgreich beendet wurden müssen in Datenbank abfragbar sein.
6	Isolation von Schreib / Schreib-Konflikten	1. NewOrder-Transaktion T1 für Kunde K1 starten. 2. T1 vor Commit stoppen. 3. Weitere NewOrder-Transaktion T2 für Kunde K1 starten. 4. T2 muss warten. 5. T1 darf Commit durchführen. 6. T2 sollte nun auch Commit durchführen.		1. Auftragsnummer (OrderNumber) von T2 muss größer sein als T1 (wenn nur zwei Transaktion gleichzeitig gelaufen sind, dann um genau 1 größer) und muss größer sein als vor T1 (bei nur zwei durchgeführten Transaktionen genau um 1 größer). 2. T2 muss gewartet haben. 3. Konsistenzbedingungen müssen erfüllt sein.

ID	*Testfall*	*Details*	*Fehlerfall*	*Ergebnis*
7	Isolation von Schreib / Schreib-Konflikten bei Fehler	1. NewOrder-Transaktion T1 für Kunde K1 starten. 2. T1 vor Commit stoppen. 3. Weitere NewOrder-Transaktion T2 für Kunde K1 starten. 4. T2 muss warten. 5. In T1 tritt Fehler auf (Rollback). 6. T2 sollte nun auch Commit durchführen.	APP / ERROR, RUN / ERROR	1. Auftragsnummer (Order-Number) von T2 muss größer sein als vor T1 (wenn nur zwei Transaktion gleichzeitig gelaufen sind, dann um genau 1 größer). 2. T2 muss gewartet haben. 3. Konsistenzbedingungen müssen erfüllt sein.
8	Isolation von Schreib / Schreib-Konflikten bei Fehler	1. NewOrder-Transaktion T1 für Kunde K1 starten. 2. T1 vor Commit stoppen. 3. Weitere NewOrder-Transaktion T2 für Kunde K1 starten. 4. T2 muss warten. 5. In T1 tritt Fehler auf 6. T2 läuft ebenfalls auf Fehler. 7. Systemreset. 8. NewOrder-Transaktion T3 für Kunde K1 durchführen.	RUN / FATAL	1. Auftragsnummer (Order-Number) von T3 muss größer sein als vor T1 (wenn nur zwei Transaktion gleichzeitig gelaufen sind, dann um genau 1 größer). 2. T2 muss gewartet haben. 3. Konsistenzbedingungen müssen erfüllt sein. 4. Alle Transaktionen welche erfolgreich beendet wurden müssen in Datenbank abfragbar sein.

ID	*Testfall*	*Details*	*Fehlerfall*	*Ergebnis*
9	Isolation von Repeatable-Read Konflikten	1. Pricing-Transaktion T1 für Item x1 und x2 durchführen. 2. NewOrder-Transaktion T2 starten. Order soll Item x1 zweimal und x2 einmal enthalten. 3. T2 nach Abfrage des Preises für erstes x1 stoppen und unmittelbar vor Abfrage des Preises für zweites x1 und x2. 4. PriceAcquisition-Transaktion T3 durchführen welche die vorhandenen Preise um 10% erhöht für Item x1 und x2. 5. T3 blockiert. 6. T2 weiterlaufen lassen und Commit durchführen. 7. T3 sollte danach auch weiterlaufen und Commit durchführen. 8. Pricing-Transaktion T4 durchführen.		1. Preise für beide x1 und x2 in T2 müssen gleich den Werten von T1 sein. 2. Preise für x1 und x2 in T4 müssen gleich den modifizierten Werten von T3 sein. 3. T3 muss gewartet haben. 4. Konsistenzbedingungen müssen erfüllt sein.

ID	*Testfall*	*Details*	*Fehlerfall*	*Ergebnis*
10	Isolation von Repeatable-Read Konflikten bei Fehlern	1. Pricing-Transaktion T1 für Item x1 und x2 durchführen. 2. NewOrder-Transaktion T2 starten. Order soll Item x1 zweimal und x2 einmal enthalten. 3. T2 nach Abfrage des Preises für erstes x1 stoppen und unmittelbar vor Abfrage des Preises für zweites x1 und x2. 4. PriceAcquisition-Transaktion T3 durchführen welche die vorhandenen Preise um 10% erhöht für Item x1 und x2. 5. T3 blockiert. 6. T2 weiterlaufen lassen und Commit durchführen. 7. In T3 tritt Fehler auf (Rollback). 8. Pricing-Transaktion T4 durchführen.	APP / ERROR, RUN / ERROR	1. Preise für beide x1 und x2 in T2 müssen gleich den Werten von T1 sein. 2. Preise für x1 und x2 in T4 müssen gleich den Werten von T1 sein. 3. T3 muss gewartet haben. 4. Konsistenzbedingungen müssen erfüllt sein.

ID	*Testfall*	*Details*	*Fehlerfall*	*Ergebnis*
11	Isolation von Repeatable-Read Konflikten bei Fehlern	1. Pricing-Transaktion T1 für Item x1 und x2 durchführen. 2. NewOrder-Transaktion T2 starten. Order soll Item x1 zweimal und x2 einmal enthalten. 3. T2 nach Abfrage des Preises für erstes x1 stoppen und unmittelbar vor Abfrage des Preises für zweites x1 und x2. 4. PriceAcquisition-Transaktion T3 durchführen welche die vorhandenen Preise um 10% erhöht für Item x1 und x2. 5. T3 blockiert. 6. In T2 tritt Fehler auf. 7. T3 läuft ebenfalls auf Fehler. 8. Systemreset. 9. Pricing-Transaktion T4 durchführen.	RUN / FATAL	1. Preise für beide x1 und x2 in T2 müssen gleich den Werten von T1 sein. 2. Preise für x1 und x2 in T4 müssen gleich den Werten von T1 sein. 3. T3 muss gewartet haben. 4. Konsistenzbedingungen müssen erfüllt sein. 5. Alle Transaktionen welche erfolgreich beendet wurden müssen in Datenbank abfragbar sein.

ID	*Testfall*	*Details*	*Fehlerfall*	*Ergebnis*
12	Isolation von Phantom Konflikten	1. OrderStatus-Transaktion T1 für einen Kunden K1 starten. 2. T1 nach dem lesen der ORDER-Tabelle stoppen (letzte Auftragsnummer ist bekannt). 3. NewOrder-Transaktion T2 für Kunde K1 starten. 4. T2 blockiert (oder nicht, dann wird Commit durchgeführt). 5. T1 weiterlaufen lassen, indem nochmals die ORDER-Tabelle ausgewertet wird und die letzte Auftragsnummer bestimmt wird. 6. T1 Commit durchführen. 7. Falls T2 blockiert hat, sollte T2 nun weiterlaufen und auch Commit durchführen.		1. Letzter Auftrag bzw. Auftragsnummer muss für beide Leseversuche von T1 derselbe sein. 2. (T2 muss gewartet haben.) 3. Konsistenzbedingungen müssen erfüllt sein.

Als Rahmenbedingung soll noch gelten, dass es möglich sein muss eine Basislast von *NewOrder*-Transaktionen auf dem Testsystem vor und während der Tests auszuführen. So ist sichergestellt, dass das System schon *dirty* ist. Damit lässt sich bei einem fatalen Fehler überprüfen, ob irgendeine Form von Caching in den beteiligten Komponenten die Transaktionssicherheit gefährdet.

Anhang C : Entwurf des Transaction Testsystems (TTS)

Aus den in der Analyse gewonnenen Anforderungen (Anhang B) lassen sich nun die architekturrelevanten Anforderungen ableiten. Diese dienen dazu, den architekturellen Rahmen des Transaction Testsystems (TTS) abzustecken, Komponenten abzugrenzen und die Kommunikations- und Datenflüsse festzulegen. Die weiteren Anforderungen haben Einfluss auf das Design der Funktionalitäten bzw. des Datenmodells der einzelnen Komponenten.

C.1 Architektur des TTS

Die System-Architektur zerlegt ein komplexes System in einzelne grobgranulare Komponenten, welche jeweils eine spezifische Rolle im Gesamtsystem erfüllen. Es wird die Abhängigkeit der Komponenten aufgezeigt sowie der Kommunikations- und Datenfluss festgelegt. Aus der in der Analyse gesammelten Anforderungen lassen sich folgende architekturrelevanten Aspekte gewinnen:

- Das Testsystem besteht zumindest aus den Komponenten *TestController* und *System under Test (SUT)*. Der *TestController* steuert die Ausführung der Tests, welche auf dem *System under Test (SUT)* ausgeführt werden.
- Die Komponenten *TestController* und *System under Test* müssen auf physikalisch getrennten Maschinen angesiedelt werden, da für einzelne Fehlerszenarien die Maschine mit der SUT-Komponente komplett beendet und neu gestartet werden muss.
- Das gesamte TTS v.a. jedoch das SUT muss gemäß J2EE-Spezifikation entwickelt werden, um auf möglichst vielen J2EE Produkten ablauffähig zu sein.
- Um Anwendungsszenarien flexibel hinzuzufügen und auch ihre Transaktionseigenschaften und Einbettung in eine Transaktion einfach verändern zu können, muss das Testszenario in einer Klasse abgebildet werden mit einer einheitlichen Schnittstelle. Die veränderlichen Informationen (z.B. Transaktionseigenschaften) müssen in einer Konfiguration abgelegt werden. Damit ist es möglich ein technisches Framework bereitzustellen, welches nichts von der Fachlichkeit des Tests kennen muss, jedoch trotzdem die Test-Klassen ausführen und auswerten kann.
- Das TTS muss eine Implementierung einer J2EE-Anwendung im SUT bereitstellen, welches für die Tests der J2EE-Infrastruktur herangezogen werden kann. Eine J2EE-Anwendung besteht aus dem Datenmodell sowie der Funktionalität welche die Geschäftsprozesse eines Unternehmens abbildet (z.B. *NewOrder*-Transaktion).
- Das TTS muss eine Integrationskomponente vorsehen, damit nicht nur die eigene J2EE-Anwendung ausgeführt, sondern auch eine separat entwickelte evtl. schon produktive Anwendung als zu testendes System eingeklinkt werden kann. Dies bedingt, dass keine

oder zumindest möglichst wenige Änderungen an der produktiven Anwendung gemacht werden dürfen, um sie testbar zu machen.

- Das TTS muss während der gesamten Testausführung Daten über die Transaktion sammeln und speichern, da eine Validierung des Testergebnisses erst nach Abschluss des gesamten Testfalls durchgeführt werden kann.
- Das TTS muss pro Testfall mehrere parallele Transaktion starten und kontrollieren können. Zusätzlich muss das TTS eine Basislast auf dem SUT über die gesamte Testausführung erzeugen.
- Das TTS muss die einzelnen parallelen Transaktionen zu einem Testfall an bestimmten (pro Anwendungsfall verschiedenen) Testpunkten miteinander synchronisieren, da ein Fehlerfall in einer exakt vorgesehenen Konstellation der Transaktionen ausgeführt werden muss. Nur so ist es nach der Testausführung möglich, ein Erwartetes mit einem angenommenen Ergebnis erfolgreich vergleichen zu können.
- Viele Testfälle gehen davon aus, dass eine der Transaktionen während der Testfalldurchführung blockiert werden muss aufgrund einer Sperre des Transaktionsmanagers beim Zugriff auf gemeinsam genutzte Ressourcen. Allein auf die implizite Annahme zu bauen, dass wenn das erwartete Ergebnis mit dem tatsächlichen Ergebnis übereinstimmt auch eine Blockierung in Folge einer Sperre aufgetreten ist, erscheint zu unsicher. Das TTS muss eine Möglichkeit vorsehen, solche Blockierungssituationen zu erkennen.

Zusätzlich wäre wünschenswert:

- Das Benutzen von existierenden Standards und Anwendungen, wenn möglich (z.B. JUnit, Apache Cactus u. a. [Mas03])
- Das Vorsehen von Integrationsmöglichkeiten in bestehende Test-Ansätze und Test-Plattformen, z.B. in das Hyades-Projekt auf Basis der Entwicklungsplattform Eclipse ([Ecl04]).

Mit diesen architekturrelevanten Aspekten lässt sich nun eine Architektur des Systems bestimmen welche die gestellten Anforderungen abdecken kann. Das *Transaction Testsystem (TTS)* besteht aus dem *TestController-System (TCS)* und dem *System under Test (SUT)*. Das TCS beinhaltet den `TestClient` welcher ein Tester benutzt um eine Testsuite, bestehend aus mehreren Testfällen, zu erstellen und dem `TestController` zu übergeben (Abbildung 43, 1). Der `TestController` ist ein Dienst innerhalb eines J2EE-Containers und stellt Schnittstellen bereit zum Ausführen und Interagieren mit einer Testsuite. Der `TestController` verwaltet die Testprozesse zu einem Testfall und übergibt diese zur Ausführung auf dem SUT dem `ProcessManager` (Abbildung 43, 2). Ebenfalls steuert er die Basislast gegen das SUT während der Ausführung eines Testprozesses. Der `ProcessManager` ist die Integrationsschicht zwischen dem TTS und einer beliebigen zu testenden J2EE-Anwendung (Abbildung 43, 3). Während der Ausführung des Testprozesses kommuniziert der `Process-`

`Manager` an bestimmten Synchronisationspunkten mit dem `TestController` (Abbildung 43, 4), um sich mit den parallel ausgeführten Testprozessen desselben Testfalls zu synchronisieren. Die Synchronisation wird vom `TestController` durchgeführt. Wenn ein Testprozess den für ihn vorgesehenen Synchronisationspunkt erreicht hat wird der Methodenaufruf vom `TestController` intern blockiert, bis die restlichen Testprozesse ebenfalls ihre Synchronisationspunkte erreicht haben. Wenn alle Testprozesse ihren Synchronisationspunkt erreicht haben, kehren die Testprozesse in einer definierten Reihenfolge zurück. Bei der Rückkehr der Testprozesse können Fehlerfälle im SUT ausgelöst werden. Bei Fehlern der Fehlerkategorie ERROR (z.B. `NullPointerException`), darf nur der fehlerauslösende Testprozess fehlschlagen und ein Rollback durchführen, alle anderen Testprozesse müssen erfolgreich sein. Bei der Fehlerkategorie FATAL (z.B. *Systemcrash*) laufen alle aktiven Testprozesse auf einen Fehler. Das System muss evtl. neu gestartet und auf dem *Resourcemanager* ein *Recovery* durchgeführt werden. Nach der Ausführung eines Testprozesses startet das TCS die Überprüfung der Korrektheit der Ergebnisse des Testprozesses, sowie die Konsistenzprüfung der Tabelleninhalte der Anwendungsdatenbank. Wenn das SUT durch die Ausführung eines FATAL-Fehlers zeitweise nicht verfügbar ist, wird die Validierung solange verzögert. Zuletzt werden die Daten für die Testmetrik und die Vergleichsmetrik ausgewertet und in die Metrik-Datenbank des TCS geschrieben. Der `TestClient` kann jederzeit beim TCS den Status und die Ergebnisse der übergebenen Testsuite abfragen (Abbildung 43, 5).

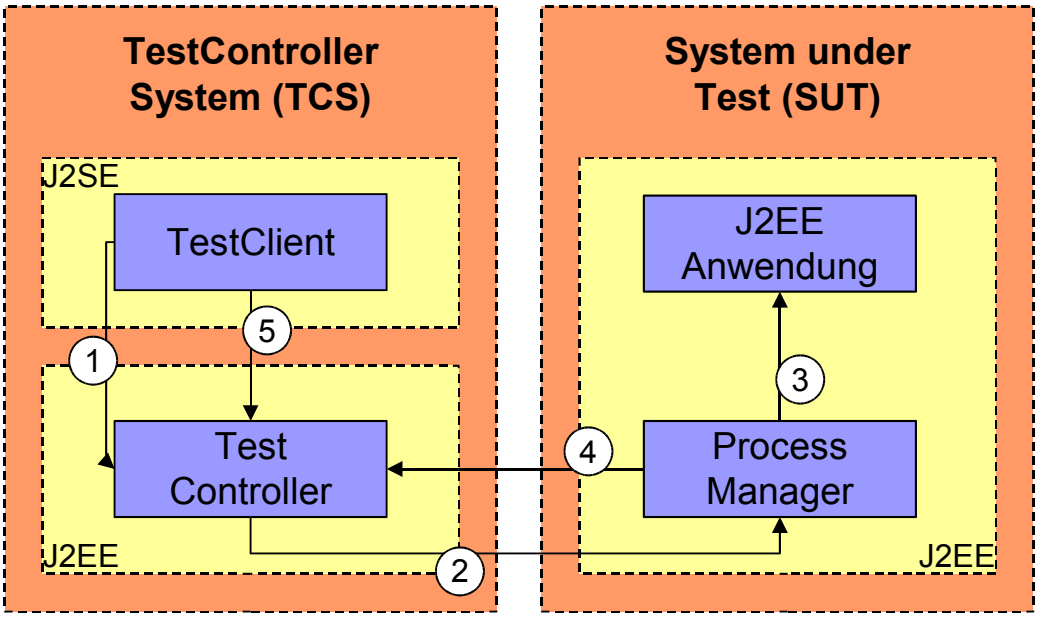

Abbildung 43 Architektur des Transaction Testsystem (TTS)

C.2 Das TestController-System (TCS)

Der TestController ist ein Dienst innerhalb eines J2EE-Containers. Er stellt Schnittstellen bereit zum Ausführen einer Test-Konfiguration, der Interaktion mit der ablaufenden Test-Konfiguration sowie um ihren Status abzufragen.

C.2.1 Die Test-Konfiguration

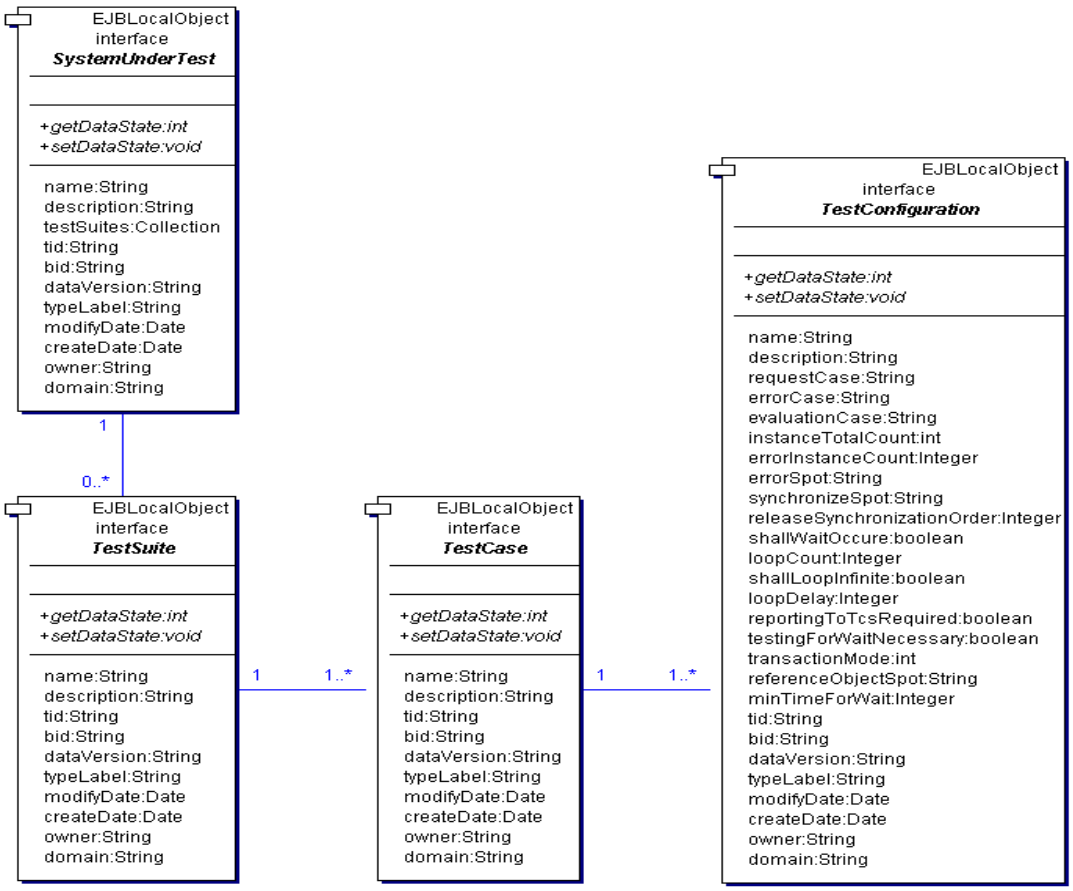

Abbildung 44 Datenmodell der Test-Konfiguration

Die Basis für die gesamte Testdurchführung ist die Test-Konfiguration (Abbildung 44). Die Test-Konfiguration besteht aus den EntityBeans `TestSuite`, `TestCase` und `TestConfiguration`. Eine `TestSuite` stellt die Gesamtheit der Testfälle dar, welche auf dem System in einem Testdurchgang ausgeführt werden. Eine `TestSuite` beinhaltet einen oder mehrere `TestCases`. Der `TestController` führt die einzelnen `TestCases` sequentiell nacheinander aus. Ein `TestCase` beinhaltet einen Testfall sowie die dafür definierte Basislast. Ein `TestCase` besteht aus einer oder mehreren `TestConfigura-`

tions. Der TestController führt die einzelnen TestConfigurations für einen TestCase parallel aus, da diese innerhalb des Testfalls miteinander agieren. Beispielsweise wartet eine Transaktion T1, welche durch TestConfiguration TC1 gestartet wurde, bis Transaktion T2, welche aus TestConfiguration TC2 gestartet wurde, seinen Synchronisationspunkt erreicht hat.

Die EntityBean SystemUnderTest stellt das zu testende System dar für welches die Testfälle erfasst wurden. Es kann hier beschrieben werden welche Ausprägung und welche Leistungsdaten das System hat. So bleibt erhalten in welchem Kontext und auf welcher Maschine die TestSuite ausgeführt wurde. Tabelle 24 beschreibt die TestConfiguration etwas ausführlicher.

Tabelle 24 Beschreibung der Attribute von TestConfiguration

Attribut	***Beschreibung***
Name	Name der TestConfiguration.
Description	Beschreibung der TestConfiguration.
RequestCase	Voll qualifizierter Klassenname eines JUnit-TestCase, welcher den Testprozess erzeugt und mit Daten initialisiert.
ErrorCase	Voll qualifizierter Klassenname eines JUnit-TestCase, welcher benutzt wird um einen Fehlerfall zu erzeugen.
EvaluationCase	Voll qualifizierter Klassenname eines JUnit-TestCase, welcher benutzt wird um das Ergebnis des Testfalls zu bestimmen und mit einem erwarteten Ergebnis zu vergleichen.
InstanceTotal-Count	Eine TestConfiguration kann einmal erfasst werden, aber mehrfach mit denselben Parametern ausgeführt werden (z.B. für Konfiguration der Basislast). Dieses Attribut gibt an wie häufig die TestConfiguration parallel ausgeführt werden soll.
ErrorInstance-Count	Wenn eine TestConfiguration mehrfach ausgeführt wird (siehe InstanceTotalCount) kann innerhalb dieser parallelen Instanzen ebenfalls eine Instanz einen Fehler ausführen. Über dieses Mittel lässt sich recht einfach die Beeinflussung von parallelen Transaktionen im Fehlerfall testen. Dieses Attribut gibt die Instanz an, welche den Fehlerfall ausführen soll. Es gilt: ErrorInstanceCount <= InstanceTotalCount
Synchronize-Spot	Gibt die symbolische Bezeichnung des Synchronisationspunktes innerhalb des Testprozesses an, bei dem diese Transaktion mit den anderen Transaktionen synchronisiert werden soll.

Attribut	*Beschreibung*
Release Synchroniz-ationOrder	Nachdem alle Testprozesse eines `TestCases` ihren Synchronisationspunkt (`SynchronizeSpot`) erreicht haben werden diese nacheinander wieder freigegeben. Dieses Attribut bestimmt die Reihenfolge in welcher die Testprozesse wieder freigegeben werden. Werte: 0,1,2.... Das setzen von allen Testprozessen auf 0 hat zur Folge das alle gleichzeitig freigegeben werden.
ReleaseDelay	Dieser Wert gibt an welcher zeitliche Versatz zwischen den einzelnen Schritten in `ReleaseSynchronizationOrder` liegen. Werte werden in Millisekunden angegeben. Beispielsweise bedeutet ein Wert von 2000, bei einem Wert in `ReleaseSynchronizationOrder` von 1, dass der Testprozess 2000 ms nach dem Freigegeben des Testprozesses mit dem Wert 0 freigegeben wird.
ShallWait-Occure	Mit diesem Wert wird angezeigt ob angenommen wird, dass diese Trans-aktion während der Testausführung blockiert, aufgrund pessimistischem Sperrverfahren auf gemeinsam genutzten Daten.
LoopCount	Eine `TestConfiguration` kann angewiesen werden, dass sie nicht nur einmal ausgeführt wird, sondern so oft wie in diesem Wert angegeben.
ShallLoop-Infinite	Eine `TestConfiguration` kann angewiesen werden, dass sie nicht nur einmal ausgeführt wird, sondern solange ausgeführt wird bis alle anderen `TestConfiguration` (mit `ShallLoopInfinite` == false) durchge-führt worden sind. Dieser Parameter ist speziell für `TestConfigurations` geeignet, wel-che eine Basislast für einen `TestCase` erzeugen.
LoopDelay	Dieser Wert gibt für eine mehrmalige Ausführung derselben `TestConfi-guration` (siehe `LoopCount` oder `ShallLoopInfinite`) an, mit welchem zeitlichen Versatz die nächste Ausführung der `TestConfigu-ration` anlaufen soll.
ReportingTo-TcsRequired	Dieser Wert gibt für die `TestConfiguration` an ob der `ProcessMa-nager` mit dem `TestController` kommunizieren soll, wenn Test- bzw. Synchronisationspunkte durchlaufen werden. Eignet sich für `TestConfigurations` welche reine Basislast erzeugen sollen oder bei Performance-Messungen.
TestingFor-WaitNecessary	Hiermit wird der Klasse des `EvaluationCase` übergeben, dass sie prü-fen soll ob die Ausführung des Testprozesses blockiert wurde.

Attribut	*Beschreibung*
Transaction-Mode	Dieser Wert gibt das Transaktionsattribut an, welches für die Ausführung des Testprozesses auf dem `ProcessManager` benutzt werden soll.
Reference-ObjectSpot	Dieses Attribut gibt die symbolische Bezeichnung des Synchronisations-punktes innerhalb des Testprozesses an, wo der Zustand von Objekten gespeichert werden sollen, welche für eine spätere Evaluierung benötigt werden. Die Daten der Objekte können später mit einem Endzustand der Daten in der Datenbank verglichen werden.
MinTimeFor-Wait	Dieser Wert gibt die Dauer in Millisekunden an, ab welcher eine gemessene Aufrufverzögerung innerhalb aufeinander folgender Messpunkte als Blo-ckierung gewertet wird.

C.2.2 Der TestContext

Um während der Ausführung des `TestCases` relevante Daten zwischen den Systemen zu übergeben sowie Daten über den Test zu sammeln, wird jedem Testprozess ein `TestCon-text` übergeben. Tabelle 25 beschreibt den `TestContext` genauer.

Tabelle 25 Beschreibung der Methoden von TestContext

Methode	*Funktionalität*
Handle getHandle() void setHandle(Handle handle)	Über diese Methoden wird dem `TestContext` ein `Handle` auf die Laufzeitinstanz des Testpro-zesses übergeben oder abgefragt. Mit diesem `Handle` ist der Testprozess, welcher den `TestContext` enthält am `TestCon-troller` registriert. Bei der Kommunikation zwischen `ProcessManager` und `TestCon-troller` wird immer dieses `Handle` übergeben, damit der `TestController` den ankommenden Aufruf zuordnen kann.

Methode	***Funktionalität***
void startTest() void stopTest() long getUsedTimeForTest() long getStartTime() long getEndTime()	Beim Start des Testprozesses im `TestController` wird die Methode *`startTest`* auf dem `TestContext` aufgerufen. Der `TestContext` wird dadurch initialisiert und es wird die Startzeit bestimmt. Nach Durchführung des Testprozesses wird *`stopTest`* auf dem `TestContext` aufgerufen, damit die Endezeit bestimmt wird. Mit den Methoden *`getStartTime`* und *`getEndTime`* lassen sich die entsprechenden Werte abfragen. Mit *`getUsedTimeForTest`* kann die Dauer des Tests in Millisekunden abgefragt werden.
void startProcess() void stopProcess() long getProcessEndTime() long getProcessStartTime() long getUsedTimeForProcess()	Beim Start des Testprozesses im `ProcessManager` wird *`startProcess`* auf dem `TestContext` aufgerufen, damit die Startzeit der Ausführung des Prozesses bestimmt wird. Nach Durchführung des Testprozesses im `ProcessManager` wird *`stopProcess`* auf dem `TestContext` aufgerufen, damit die Endezeit bestimmt wird. Mit den Methoden *`getProcessStartTime`* und *`getProcessEndTime`* lassen sich die entsprechenden Werte abfragen. Mit *`getUsedTimeForProcess`* kann die Dauer der Ausführung des Testprozesses in Millisekunden abgefragt werden.

Methode	***Funktionalität***
void interruptExecutionBusinessLogic() void resumeExecutionBusinessLogic() long getUsedTimeInBusinessLogic() int getInterruptsCount() boolean isReportingToTcsRequired()	Wenn *isReportingToTcsRequired* nicht auf false gesetzt ist, wird die Ausführung eines Testprozesses im `ProcessManager` an jedem Synchronisationspunkt durch die Kommunikation mit dem `TestController` unterbrochen. Damit trotz dieser zeitaufwendigen Kommunikation zwischen zwei physikalisch separaten Einheiten eine sinnvolle Performanz-Aussage für den Testprozess getroffen werden kann wird bei Beginn der Ausführung des Synchronisationspunktes die Methode *interruptExecution-BusinessLogic* aufgerufen, welche den Beginn der Unterbrechung der Testprozess-Ausführung bezeichnet. Intern wird ein Zeitstempel (*lastSuspendTime*) gesetzt und der *Interrupt-Counter* inkrementiert. Nach der Ausführung der internen Logik des Synchronisationspunktes wird die Methode *resumeExecutionBusiness-Logic* aufgerufen, welche einen erneuten Zeitstempel (*lastResumeTime*) nimmt. Die Differenz zwischen lastResumeTime und lastSuspend-Time zum Zeitpunkt eines erneuten *inter-ruptExecutionBusinessLogik* Aufrufs wird zum Wert von *usedTimeInBusiness-Logic* hinzugezählt. Über die entsprechende Methode *getUsedTimeInBusinessLogic* erhält man die Dauer der reinen Ausführung des Testprozesses ohne Overhead der Ausführung der Synchronisationspunkte.

Methode	***Funktionalität***
void startPossibleWait() void stopPossibleWait() boolean isTestingForWaitNecessary() boolean isWaitOccured()	Um herauszufinden ob eine Transaktion durch ein Sperrverfahren blockiert wird muss mit einem Trick gearbeitet werden, da für einen derartigen Fall kein Listener gemäß EJB-Spezifikation vorgesehen ist. Deshalb kann nur mittels der Messung von Zeit zwischen zwei Aufrufen gearbeitet werden, in welcher eine potentielle Blockierungssituation (*Wait*) passieren könnte. Da die Zeitdauer (zumindest wenn man nicht eine unangemessen hohe Zeit annimmt) von `TestConfiguration` zu `TestConfiguration` unterschiedlich ist kann über das Attribut *`minTimeForWait`* der `TestConfiguration` angegeben werden, ab welcher Zeitdauer eine Transaktion als „blockiert“ gewertet wird. Vor einem Aufruf einer potentiell blockierenden Funktionalität wird *`startPossibleWait`* aufgerufen, welche eine Zeitmarke setzt. Nach dem Aufruf wird *`stopPossibleWait`* aufgerufen, welche ebenfalls wieder eine Zeitmarke nimmt. Bei der anschließenden Auswertung (`EvaluationCase`) des Testfalls wird über *`isTestingForWaitNecessary`* überprüft ob die Transaktion überhaupt blockieren sollte und dann mit *`isWaitOccured`* ermittelt ob dem so war.
String getErrorCase() void setErrorCase(String errorCase) String getReferenceSpotId() void setReferenceSpotId(String string)	Wenn der `TestController` nach Erreichung der Synchronisationspunkte aller beteiligten Transaktionen die Ausführung eines Fehlerfalls anordnet, liefert er den Klassenname im `TestContext` zurück. Der `ProcessManager` holt sich den Klassenname über *`getErrorCase`*, instanziert die Klasse und führt sie aus. Über die *`getReferenceSpotId`* Methode überprüft der Testprozess bei erreichen eines Testpunktes ob der Objektzustand gesichert werden muss.

C.2.3 Der TestController

Die Komponente `TestController` besteht aus zwei Einheiten (Abbildung 45). Das SessionBean `TestControllerBean` welches den eigentlichen Dienst bereitstellt und das *Message-Driven Bean* `TestProcessorMDB`. Das `TestProcessorMDB` wird vom `TestControllerBean` intern benutzt, um zum einen die Übergabe der `TestSuite` vom `TestClient` an den `TestController` von der Ausführung der `TestSuite` zu entkoppeln, sowie die parallele Ausführung der einzelnen Testprozesse eines `TestCases` zu gewährleisten. Die Übergabe erfolgt über eine JMS-Queue mit Namen `TestProcessorQ`.

Abbildung 45 Design von TestControllerBean und TestProcessorMDB

Der Client übergibt die `TestSuite` über die Methode *`runTestSuite`*. Die `TestSuite` wird in ein XML-Format umgewandelt und als Text-Message an die `TestProcessorQ` versendet. Für die TestSuite wird ein `Handle` generiert und an den `TestClient` zurückgegeben. Der Client kann damit stets den aktuellen Status der `TestSuite` abfragen. Das `TestProcessorMDB` konsumiert die Message über *`onMessage`*, wandelt sie von der XML-Repräsentation in ein Objekt um. Jeder `TestCase` der `TestSuite` wird sequentiell an *`runTestCase`* übergeben. Jede `TestConfiguration` des `TestCases` wird vom `TestController` ausgewertet und die entsprechenden Testprozesse erzeugt. Diese werden wiederum in XML-Format umgewandelt und als Text-Message an die `TestProcessorQ` gesendet. Die `TestProcessorMDB` konsumiert diese wiederum und wandelt die Message in ein Objekt zurück. Der Testprozess wird über *`runTest`* des `TestControllers` ausgeführt. Pro Testprozess wird eine `TestRun`-Laufzeit-Instanz in die Datenbank gespeichert und ein `TestRun-Handle` erzeugt. Dieses

`Handle` wird als identifizierendes Merkmal bei allen Methoden-Aufrufen zusammen mit dem `TestContext` übergeben. Die *`getTestContextForHandle`* und *`storeTestContextForHandle`* sind für die Interaktion mit dem zu einem `TestRun-Handle` gehörenden `TestContext` zuständig welcher in der Datenbank gespeichert wird. Die *`getTestRunForTestSuiteHandle`* wird hauptsächlich vom `TestClient` benötigt, um zu einem `Handle` auf die `TestSuite` die korrespondierende `TestRun`-Instanzen zu erhalten. Über die `TestRun`-Instanzen erhält der `TestClient` zu jedem Testprozess detaillierte Informationen über Status und Ergebnis.

Der `ProcessManager` ruft bei der Ausführung eines Synchronisationspunktes die *`reportToExecutionMonitor`*-Methode auf. Er übergibt dabei das `TestRun-Handle` und den `TestContext`. Innerhalb dieser Methode überprüft der `TestController` ob sich der Testprozess schon am richtigen Synchronisationspunkt befindet, um den Testfall durchzuführen. Wenn nicht darf die Methode sofort zurückkehren, wenn ja dann wird die Rückkehr blockiert. Es wird dann überprüft ob sich alle anderen Testprozesse für diesen `TestCase` am richtigen Synchronisationspunkt befinden. Trifft dies noch nicht zu bleiben die Methoden der Testprozesse, welche schon im richtigen Synchronisationspunkt sind, blockiert bis die verbleibenden Testprozesse ebenfalls den richtigen Synchronisationspunkt erreicht haben. Der letzte Testprozess welcher an seinem vorgesehenen Synchronisationspunkt eintrifft wird als solcher erkannt. Es werden nun die einzelnen Testprozesse gemäß ihrer in `ReleaseSynchronizationOrder` festgelegten Reihenfolge und in `ReleaseDelay` definierten Verzögerung wieder freigegeben. Der ausgeführte Testprozess im `ProcessManager` hat noch eine weitere Möglichkeit, Informationen an den `TestController` zu übertragen. Wenn er keinen Synchronisationspunkt durchläuft, sondern nur Informationen (z.B. ein *Log-Statement*) an den `TestController` übergeben möchte, damit dies zur `TestRun`-Instanz in der Datenbank protokolliert wird, kann er dies über die Methode *`logTestEvent`* erreichen. Auch hier wird die Ausführung der Businesslogik unterbrochen und damit *`interruptExecutionBusinesslogic`* und *`resumeExcecutionBusinesslogic`* aufgerufen.

Mit der Methode *`createSnapshotStatistic`* werden Informationen über den aktuellen Zustand der Datenbank gesammelt und in eine `SnapShotStatistic`-Entität zur `TestRun`-Instanz gespeichert. Hauptsächlich wird damit ermittelt wieviele Records eine Tabelle des Datenmodells der J2EE-Anwendung insgesamt hat, wieviele von dieser `TestRun`-Instanz erzeugt wurden und wieviele von dieser `TestRun`-Instanz modifiziert wurden. Um dies technisch zu realisieren, definiert jede Entität der J2EE-Anwendung drei zusätzliche Attribute (`createDate`, `lastModifyDate` und `owner`). In dem Feld `owner` trägt die ausführende `TestRun`-Instanz ihr `Handle` ein, so kann für jeden Testprozess bzw.

`TestCase` aufgelistet werden wieviele Records erzeugt und modifiziert wurden und ob dies mit den erwarteten Zahlen übereinstimmt.

Die Methoden-Gruppe aus *`storeReportedDataToTestRun`*, *`findModifiedObjects`* und *`getObjectDataForModifiedObjectDataBO`* ermöglicht es, dass der Testprozess den Inhalt von innerhalb seiner Transaktion veränderten Objekten an den `TestController` zur Persistierung übergibt, zusammen mit dem `TestRun-Handle` und der symbolischen Bezeichnung des Testpunktes. Diese Informationen werden vom `TestController` mit Bezug zur `TestRun`-Instanz persistiert und lassen sich für die Evaluierung von Objektzuständen am Ende einer Transaktion nutzen. Die übrigen Methoden dienen dazu, die interne Abarbeitung der Testprozesse zu gewährleisten und ihren Status zu setzen oder abzufragen.

C.3 Der ProcessManager

Der `ProcessManager` ist die Integrationsschicht zwischen dem TTS und einer beliebigen J2EE-Anwendung welche getestet werden soll. Der `ProcessManager` führt den ihm übergebenen Testprozess aus. Ein Testprozess besteht aus einer konkreten Klasse, welche die Klasse `BaseProcess` erweitert. `BaseProcess` ist die Basisklasse für alle Testprozesse und implementiert die Schnittstelle `Process`, welche Methoden zur Steuerung des *Lifecycle* durch den `ProcessManager` definiert (Abbildung 46). Die Hauptfunktionalität ist in der Basisklasse `Task` definiert, die von `BaseProcess` erweitert wird. Dort wird die Verbindung zur `EJBObject`-Schnittstelle des `TestController` gehalten (*`createTestController`* / *`getTestController`* / *`clearTestController`*) und zur `EJBLocalObject`-Schnittstelle des `ProcessManager` (*`createProcessManager`* / *`getProcessManager`* / *`clearProcessManager`*). Ebenso ist in der `Task`-Basisklasse die technische Funktionalität der Ausführung eines Testpunktes (*TestSpot*) beim erreichen eines Synchronisationspunktes angesiedelt sowie das übergeben eines beliebigen `TestEvents` an den `TestController` über *`logTestEvent`*.

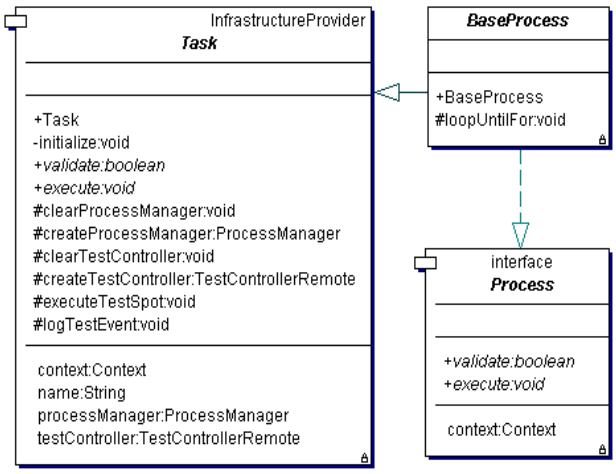

Abbildung 46 Design des Testprozesses

Der `ProcessManager` bietet über seine `EJBObject`-Schnittstelle dem `TestController` Funktionalitäten an, um Testprozesse mit unterschiedlichen Transaktionsattributen auszuführen (Abbildung 47). Da die EJB-Spezifikation für CMT eine deklarative Methodik durch den *Deployment Descriptor* eines EJBs vorschreibt sind die Transaktionseigenschaften von Methoden nicht sofort änderbar. Es müsste der *Deployment Descriptor* jeweils verändert werden, eine erneute Paketierung in ein EAR stattfinden und ein anschließendes *Deployment* in den EJB-Container. Da jedoch für einen Testprozess die jeweilige `TestConfiguration` festlegen soll in welchen Transaktionsgrenzen dieser ausgeführt werden muss, wurde der Ansatz gewählt, für jedes verwendete Transaktionsattribut eine entsprechende Methode bereitzustellen und mittels einer Abwandlung des *Command-Pattern* [Gam96] bzw. *EJB-Command-Pattern* [Mar02]einen kompletten Testprozess an die jeweilige Methode des `ProcessManagers` zu übergeben.

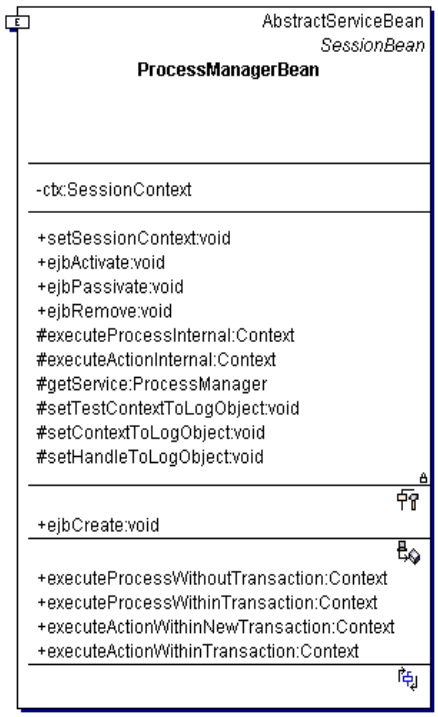

Abbildung 47 Design ProcessManagerBean

Der `ProcessManager` führt über *`executeProcessWithinTransaction`* den Testprozess innerhalb einer Transaktion aus (Transaktionsattribut *Required*), während er denselben Testprozess wenn er über *`executeProcessWithoutTransaction`* übergeben würde ohne Transaktion ausführen würde (Transaktionsattribut *NotSupported*). Dieser Modus ist sehr wichtig für langlaufende Prozesse, wie z.B. initiale Ladeprozesse, da diese sonst in einen Transaktionstimeout laufen würden. Natürlich muss die Funktionalität welche intern EntityBean-Instanzen erzeugt und auf die Datenbank speichert eine entsprechende Transaktion erzeugen (z.B. durch SessionBeans mit Transaktionsattribut *RequiresNew*).

Innerhalb der *`execute`*-Methoden des `ProcessManagers` geschieht immer dasselbe. Auf dem übergebenen Testprozess wird *`validate`* aufgerufen. Hierbei prüft der konkrete Testprozess, ob er über alle Daten verfügt um ausgeführt zu werden. Liefert der Testprozess den Wert „true“ zurück, dann wird die *`execute`*-Methode auf dem Testprozess aufgerufen. Damit wird die Businesslogik des Testprozesses ausgeführt.

C.4 Integration von beliebigen J2EE-Anwendungen

Für die Strukturierung von Anwendungen und insbesondere J2EE-Anwendungen gibt es die unterschiedlichsten Ansätze. Im folgenden soll ein Ansatz vorgestellt werden, der den Gedanken von Geschäftsprozessen und einzelner Aktionen innerhalb dieser Geschäftsprozesse aufnimmt und diese auf ein abgewandeltes *Command-Pattern* [Gam96] abbildet. Der Vorteil dieses Ansatzes im allgemeinen ist die Strukturierung des Quellcodes entlang der tatsächlichen Geschäftsprozesse (*Process*) bzw. der einzelnen Aktionen (*Action*) eines Geschäftsprozesses. Damit vereinfacht sich die Kommunikation zwischen den Designern bzw. Anwendungsentwicklern und den Experten des Geschäftsfeldes, welche das Wissen über die fachlichen Funktionalitäten haben. Des weiteren ist eine derartig gekapselte *Action* sehr viel einfacher innerhalb eines Prozesses zu verschieben oder gar in anderen Prozessen wiederzuverwenden. Im speziellen Fall des TTS wird dadurch zusätzlich die Flexibilität gewonnen, die Geschäftsprozesse, welche den Testfällen zu Grunde liegen, leicht anpassen zu können. Beispielsweise werden Funktionalitäten derartig in konkrete `Action`-Klassen zusammengruppiert, dass die Testpunkte (*Testspot*) immer zwischen den Aufrufen der `Actions` liegen, also im umgebenen Testprozess. Dadurch werden die `Action`-Klassen nicht durch Testcode verunreinigt und können so – ohne Modifikation – in Produktionsumgebungen laufen. Dieses Merkmal ist auch die Basis für die Integration von anderen J2EE-Anwendungen in das TTS, sodass diese getestet werden können. Eine tiefgreifende Modifikation einer J2EE-Anwendung sollte nicht erfolgen müssen damit sie innerhalb des TTS ablauffähig ist. Wenn getesteter Code und in der Produktion ausgeführter Code sich unterscheiden sind die Tests der Anwendung wenig aussagekräftig. Im folgenden wird dieser Ansatz mittels des Geschäftsprozesses für das Anlegen eines neuen Auftrages (*NewOrder*) erläutert.

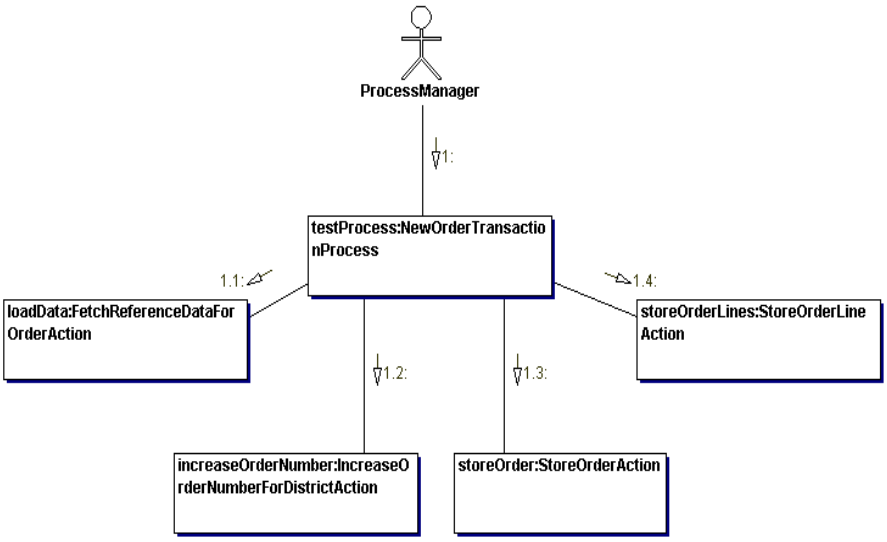

Abbildung 48 Kollaborationsdiagramm für den NewOrder-Prozess

Der `ProcessManager` führt dabei eine Instanz von `NewOrderTransactionProcess` aus (Abbildung 48). Diese ruft zunächst `FetchReferenceDataForOrderAction` auf, welche die relevanten Stammdaten für die Transaktion liest. Als nächstes wird über `IncreaseOrderNumberForDistrictAction` die letzte vergebene Auftragsnummer für den *District* ermittelt und inkrementiert. Danach wird über `StoreOrderAction` zunächst der Auftragskopf auf die entsprechenden EntityBeans (`OrderDataBean` und `NewOrderBean`) abgebildet und für jede Auftragszeile `StoreOrderLineAction` aufgerufen. Dann ist der `NewOrderTransactionProcess` beendet und kehrt zum Aufrufer zurück. Zwischen jeder der ausgeführten `Actions` können nun Testpunkte angesiedelt werden, seien es Synchronisationspunkte, einfache *Logging-Events* oder potentielle Wartepunkte beim Einsatz von pessimistischen Sperrverfahren. Ein Blick in den Quellcode soll dies veranschaulichen:

```
public class NewOrderTransactionProcess extends BaseProcess {
```

Der `NewOrderTransactionProcess` erweitert den `BaseProcess`. In `BaseProcess` und der Basisklasse `Task` werden die eigentlichen *Callback*-Methoden zur Kommunikation mit `ProcessManager` und `TestController` bereitgestellt.

```
public static final String TEST_SPOT_INIT = "01-Init";
public static final String TEST_SPOT_EXIT = "02-Exit";

public static final String TEST_SPOT_FETCH_REFERENCE_DATA_BEFORE = "10";
public static final String TEST_SPOT_FETCH_REFERENCE_DATA_AFTER = "11";

public static final String TEST_SPOT_INCREASE_ORDER_NUMBER_BEFORE = "12";
public static final String TEST_SPOT_INCREASE_ORDER_NUMBER_AFTER = "13";

public static final String TEST_SPOT_STORE_ORDER_BEFORE = "14";
public static final String TEST_SPOT_STORE_ORDER_AFTER = "15";

public static final String TEST_SPOT_STORE_ORDER_LINE_BEFORE = "16";
public static final String TEST_SPOT_STORE_ORDER_LINE_AFTER = "17";
```

Hier werden die symbolischen Bezeichnungen für die einzelnen Testpunkte dieses Prozesses definiert. Es gibt jeweils ein Paar vor dem Aufruf und nach dem Aufruf einer `Action`, sowie eine für den Beginn des Prozesses und das Ende.

```
public void execute() throws CommunicationException, UnknownApplicationException {
        initalize();

        [...]

        //InitSpot (before modifing sth at all)
        executeTestSpot(TEST_SPOT_INIT, this.getClass().getName(), null);
```

Der Prozess wird über den Aufruf von *`initalize`* initialisiert (z.B. wird hier die *Logging*-Infrastruktur initialisiert). Danach wird über *`executeTestSpot`* der erste Testpunkt

TEST_SPOT_INIT durchlaufen. Intern wird jetzt ein Aufruf am TestController gemacht und ihm signalisiert, dass dieser Testpunkt nun durchlaufen wird.

```
Action action = null;
action = new FetchReferenceDataForOrderAction();
action.setContext(this.getContext());
```

Die Action welche ausgeführt werden soll wird erzeugt und mit dem aktuellen Context befüllt. Dieser beinhaltet Werte welche vom RequestCase erzeugt wurden (z.B. OrderDataBO, OrderLineBO und gewählter Kunde) und wird benutzt um Daten zwischen den einzelnen Actions auszutauschen.

```
if (action.validate()) {

        executeTestSpot(
                TEST_SPOT_FETCH_REFERENCE_DATA_BEFORE,
                action.getClass().getName(),
                null);
```

Es wird überprüft, ob die Action alle Daten beinhaltet um mit der Ausführung beginnen zu können. Trifft dies zu, wird wiederum *executeTestSpot* aufgerufen um dem TestController über das erreichen des Testpunktes in Kenntnis zu setzen. Ist in der TestConfiguration eingetragen worden, dass dies der Synchronisationspunkt für diesen Testprozess ist, dann blockiert der TestController die Rückkehr solange bis alle Testprozesse des TestCase ihre Synchronisationspunkte erreicht haben.

```
        [...]

        context.getTestContext().startPossibleWait();
        logTestEvent(
                TEST_SPOT_FETCH_REFERENCE_DATA_BEFORE,
                action.getClass().getName(),
                getLogObject());

        action.execute();
        setContext(action.getContext());

        context.getTestContext().stopPossibleWait();
        logTestEvent(
                TEST_SPOT_FETCH_REFERENCE_DATA_AFTER,
                action.getClass().getName(),
                getLogObject());

        [...]
```

Beim Ausführen dieser Action kann eine Blockierungssituation mit einer anderen Transaktion auftreten, welche auf dieselben Ressourcen zugreift (in Folge einer gesetzten Sperre). Um später auswerten zu können, ob eine solche Blockierungssituation aufgetreten ist, wird der TestContext zunächst über *startPossibleWait* aufgefordert eine Zeitmarke zu nehmen. Danach wird ein TestEvent an den TestController kommuniziert der ein LogObject beinhaltet, welches innerhalb des ProcessManagers und des Testprozesses

zum schreiben von *Logging*-Einträgen benutzt wird. Das `LogObject` verfügt deshalb über interessante Informationen über den aktuellen Zustand des Prozesses. Diese Informationen werden vom `TestController` zu der jeweiligen `TestRun`-Instanz in die Datenbank abgelegt. Danach wird die `Action` ausgeführt (evtl. tritt hier die Blockierungssituation ein). Nach der Ausführung wird der modifizierte `Context` von der `Action` geholt und an den `Context` des aufrufenden Testprozesses (`NewOrderTransactionProcess`) überführt. Danach wird über *`stopPossibleWait`* der `TestContext` dazu aufgefordert eine weitere Zeitmarke zu nehmen. Ist die Differenz aus End- und Startmarke größer als der Schwellwert, welcher in der `TestConfiguration` für diesen Testprozess eingetragen wurde, wird der Aufruf als „blockiert“ gewertet. Der `TestController` wird wieder mit der Übertragung eines `TestEvents` von dem Durchlaufen eines Testpunktes informiert.

```
[...]

action = new IteratorAction();
action.setContext(this.getContext());

//Prepare Context
action.
        getContext().
        getLocalContext().
        setCurrentAction(new StoreOrderLineAction());

action.
        getContext().
        getLocalContext().
        setItemList(context.getCurrentOrderLines());

if (action.validate()) {

        logTestEvent(
                TEST_SPOT_STORE_ORDER_LINE_BEFORE,
                action.getClass().getName(),
                getLogObject());

        action.execute();
        setContext(action.getContext());

        logTestEvent(
                TEST_SPOT_STORE_ORDER_LINE_AFTER,
                action.getClass().getName(),
                getLogObject());

}

[...]
```

Der Aufruf der `StoreOrderLineAction` folgt demselben Prinzip wie die `Fetch-ReferenceDataForOrderAction`, jedoch muss sie für jede Zeile des Auftrages ausgeführt werden. Dafür gibt es eine spezielle technische `Action`, welche als Parameter die Zeilen des Auftrages (als `Collection`) und die auszuführende `Action` bekommt. Danach führt die `Action` für jede Zeile die übergebene `Action` aus.

C.5 Zusammenspiel der Komponenten für einen Testfall

C.5.1 Testfall und TestSuite-Konfiguration

Nachdem die Einzelkomponenten vorgestellt und ihre Funktionsweise näher beleuchtet wurden wird es nun Zeit, die Komponenten im Zusammenspiel zu betrachten. Als Testfall ziehen wir das Testszenario 6 aus Anhang B.7 heran. Die Konfiguration der `TestSuite` sieht wie folgt aus:

```
<TestSuite>
    <Bid>SUITE_001</Bid>
    <Name>NewOrder-TestSuite</Name>
    <Description>NewOrder-TestSuite</Description>
    <TestCase>
          <Bid>CASE_001</Bid>
          <Name>Isolation Write-Write of two NewOrder</Name>
          <Description>
                This test demonstrates isolation for write-write
                conflicts of two New-Order transactions
          </Description>

          <TestConfiguration>
                <Bid>CONFIG_001</Bid>
                <Name>BasisTransactionMix</Name>
                <Description>BasisTransactionMix</Description>
                <InstanceTotalCount>10</InstanceTotalCount>
                <LoopDelay>100</LoopDelay>
                <RequestCase>
                      com.byteacademy.research.
                      tts.tcs.test.NewOrderProcessTestSuite
                </RequestCase>
                <ShallLoopInfinite>true</ShallLoopInfinite>
                <TransactionMode>Required</TransactionMode>
          </TestConfiguration>

          <TestConfiguration>
                <Bid>CONFIG_002</Bid>
                <Name>Transaction T1</Name>
                <Description>Transaction which starts first</Description>
                <InstanceTotalCount>1</InstanceTotalCount>
                <SynchronizeSpot>02-Exit</SynchronizeSpot>
                <ReferenceObjectSpot>02-Exit</ReferenceObjectSpot>
                <ReleaseSynchronizationOrder>
                      2
```

```
                </ReleaseSynchronizationOrder>
                <ReportingToTcsRequired>true</ReportingToTcsRequired>
                <TestingForWaitNecessary>true</TestingForWaitNecessary>
                <ShallWaitOccure>false</ShallWaitOccure>
                <MinTimeForWait>4000</MinTimeForWait>
                <RequestCase>
                      com.byteacademy.research.
                      tts.tcs.test.NewOrderProcessTestSuite
                </RequestCase>
                <EvaluationCase>
                      com.byteacademy.research.
                      tts.tcs.test.NewOrderProcessCheckSuite
                </EvaluationCase>
                <ShallLoopInfinite>false</ShallLoopInfinite>
                <TransactionMode>Required</TransactionMode>
          </TestConfiguration>

          <TestConfiguration>
                <Bid>CONFIG_003</Bid>
                <Name>Transaction T2</Name>
                <Description>
                      Transaction which starts second - must wait
                </Description>
                <InstanceTotalCount>1</InstanceTotalCount>
                <SynchronizeSpot>10</SynchronizeSpot>
                <ReferenceObjectSpot>02-Exit</ReferenceObjectSpot>
                <ReleaseSynchronizationOrder>
                      1
                </ReleaseSynchronizationOrder>
                <ReportingToTcsRequired>true</ReportingToTcsRequired>
                <TestingForWaitNecessary>true</TestingForWaitNecessary>
                <ShallWaitOccure>true</ShallWaitOccure>
                <MinTimeForWait>4000</MinTimeForWait>
                <RequestCase>
                      com.byteacademy.research.
                      tts.tcs.test.NewOrderProcessTestSuite
                </RequestCase>
                <EvaluationCase>
                      com.byteacademy.research.
                      tts.tcs.test.NewOrderProcessCheckSuite
                </EvaluationCase>
                <ShallLoopInfinite>false</ShallLoopInfinite>
                <TransactionMode>Required</TransactionMode>
          </TestConfiguration>
    </TestCase>
</TestSuite>
```

C.5.2 TestClient erzeugt TestSuite und ruft TestController

Diese Test-Konfiguration wird auf dem `TestClient` als `TestSuiteBO` erstellt. `BO` ist die Abkürzung for *BusinessObject*, was in diesem Fall ein transienter Datenbehälter ist mit einer 1:1 Entsprechung zu einem persistenten EntityBean-Typ. Nachdem das `TestSuiteBO` erzeugt und mit den obigen Daten gefüllt wurde, wird das `BO` über die `EJBObject`-Schnittstelle dem `TestController` übergeben mittels der Methode:

```
public
      Handle runTestSuite(TestSuiteBO suite)
throws
      java.lang.IllegalArgumentException,
      CommunicationException,
      javax.ejb.FinderException,
      java.rmi.RemoteException;
```

Innerhalb dieser Methode erzeugt der `TestController` ein `Handle` für die `TestSuite`, das Property `TestSuiteRunId` wird mit einer eindeutigen Identifizierung gefüllt. Danach wird die `TestSuite` in XML-Format umgewandelt und als eine JMS-Textmessage an die `TestProcessorQ` gesendet. Ist dies erfolgreich durchgeführt worden kehrt *`runTestSuite`* mit dem Rückgabewert des erzeugten `Handles` zurück. Mittels des zurückgegebenen `Handles` hat der `TestClient` eine Referenz auf die Laufzeitinstanz der übergebenen `TestSuite`. Damit kann der `TestClient` jederzeit von der `TestSuite` den aktuellen Status und die vorliegenden Ergebnisse vom `TestController` erfragen mit:

```
public
    java.util.Collection getTestRunForTestSuiteHandle(Handle handle)
throws
    java.lang.IllegalArgumentException,
    CommunicationException,
    java.rmi.RemoteException;
```

Der Rückgabewert erhält ein `Collection`-Objekt mit Instanzen vom Typ `TestRunBO`. Für jeden ausgeführten Testprozess existiert ein Objekt. Jedes `TestRunBO` lässt sich über die Methode *`getEvaluationRunState`* nach dem Status fragen. Definiert sind:

- `STATUS_RUN_NEW = 1`
- `STATUS_RUN_STARTED = 2`
- `STATUS_RUN_WAIT = 11`
- `STATUS_RUN_RELEASED = 12`
- `STATUS_RUN_RUNNING = 50`

- STATUS_RUN_CANCELED = 100
- STATUS_RUN_FINISHED_SUCCESSFUL = 101
- STATUS_RUN_FINISHED_WITH_ERRORS = 102
- STATUS_RUN_FINISHED_WITH_FAILURES = 103
- STATUS_RUN_FINISHED_WITH_EXCEPTION = 104

Ebenfalls lassen sich die relevanten Daten für die Bewertungsmetrik abfragen.

C.5.3 Ausführung der TestSuite im TestController

Die TestSuite liegt in der TestProcessorQ als JMS-Textmessage. Auf der TestProcessorQ arbeitet das *Message-Driven Bean* TestProcessorMDB. Sobald eine neue Message in der TestProcessorQ eintrifft wird *onMessage* auf dem MDB aufgerufen und die eingetroffene Message als Parameter übergeben. Wenn die Message erfolgreich verarbeitet werden konnte wird ein *acknowledge* auf der Message aufgerufen, welches dazu führt das die JMS-Implementierung die Message als verarbeitet ansieht und die Ressourcen zu dieser Message freigibt (bei einer persistenten Message betrifft dies auch eine unterliegende Datenbank).

Das TestProcessorMDB untersucht das Header-Property MsgFunctionTarget und MsgContentType, welche vom Sender (TestController) mit den richtigen Werten belegt wurden. Das MDB erkennt aus diesen Werten, dass der Inhalt der Textmessage eine TestSuite in XML-Format ist. Das MDB delegiert die Verarbeitung an die entsprechende Funktionalität welche das XML wieder zu einem TestSuiteBO umwandelt. Danach wird die Liste der TestCases (in diesem Fall ist dies nur ein TestCase) aus dem TestSuiteBO geholt und über diese Liste iteriert. Jedes TestCaseBO wird dabei sequentiell an den TestController zur weiteren Abarbeitung übergeben. Dieser Aufruf ist blockierend, d.h. der nächste TestCase kann erst nach der erfolgreichen Abarbeitung des aktuellen TestCases an den TestController übergeben werden. Diese Eigenschaft erzwingt die sequentielle Ausführung der einzelnen TestCases innerhalb einer TestSuite. Das TestCaseBO wird mit der Methode

```
public
    void runTestCase(TestCaseBO testCase)
throws
    java.lang.IllegalArgumentException,
    CommunicationException,
    javax.ejb.FinderException,
    java.lang.InterruptedException;
```

an den `TestController` übergeben. Der `TestController` liest die Liste der `TestConfiguration`-Objekte von dem `TestCase` und benutzt diese, um die Testprozesse für den `TestCase` zu erzeugen. Für jede `TestConfiguration` wird die Anzahl der Instanzen über das Attribut *`instanceTotalCount`* bestimmt. Dies ist die Anzahl der Testprozesse, welche parallel mit derselben `TestConfiguration` ausgeführt werden sollen. Die erste `TestConfiguration` stellt die Basislast für den anschließenden Isolationstest zur Verfügung. Es sollen zehn Instanzen parallel ausgeführt werden, welche ständig (`ShallLoopInfinite` = true) eine *NewOrder*-Transaktion (`RequestCase`) ausführen sollen. Zwischen den einzelnen *NewOrder*-Transaktionen soll eine Verzögerung von 100 ms (`LoopDelay`) liegen. Die *NewOrder*-Transaktion soll mit dem Transaktionsattribut *Required* (`TransactionMode`) ausgeführt werden.

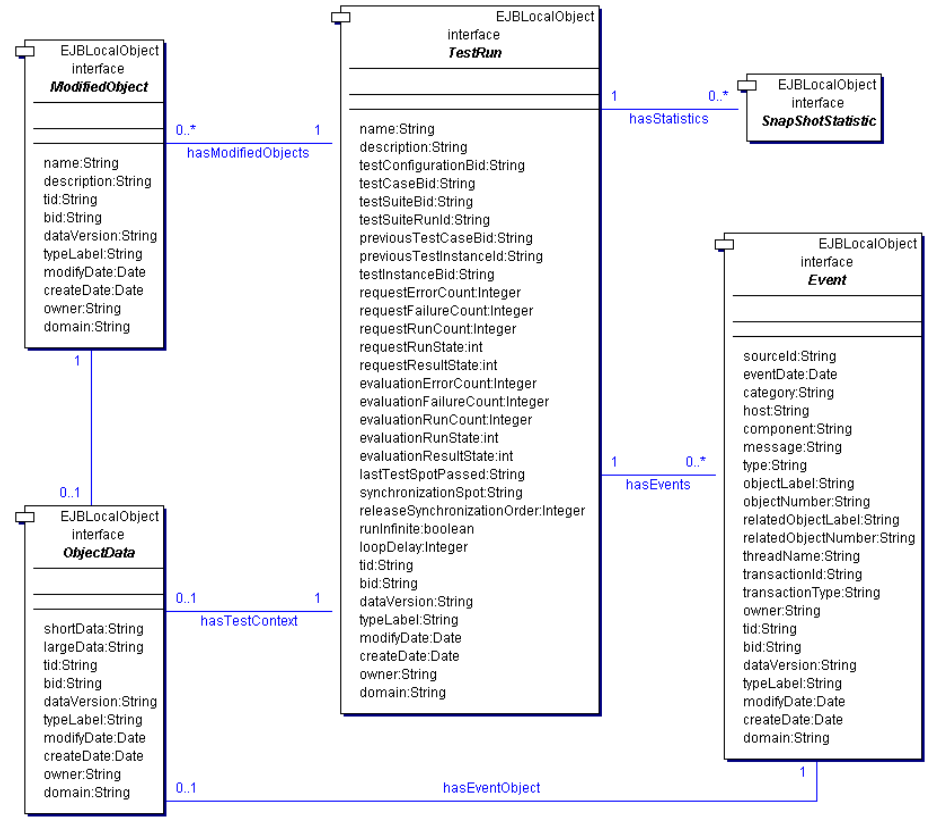

Abbildung 49 Datenmodell einer TestRun-Instanz

Für jeden Testprozess wird eine neue `TestRun`-Entität erzeugt Abbildung 49, mit den Daten der `TestConfiguration` sowie Laufzeitattributen (z.B. eindeutige Bezeichner `tid` und `bid`) versehen und in der Datenbank gespeichert. Danach wird ein `Handle` auf die konkrete `TestRun`-Instanz erzeugt. Dieses `Handle` wird in ein XML-Format umgewandelt und

per JMS-Textmessage wiederum an die `TestProcessorQ` gesendet. So wird mit allen weiteren `TestConfiguration`-Objekten desselben `TestCase` verfahren. Danach wartet die Methode bis alle Testprozesse des aktive `TestCase` beendet sind. Erst danach wird der nächste `TestCase` in der selben Weise verarbeitet.

Wieder wird *`onMessage`* von `TestProcessorMDB` mit der Textmessage als Parameter von der JMS-Implementierung aufgerufen und das `MsgFuncTarget` und der `MsgContentType` ausgewertet. Diesmal wird an den Werten dieser *Header-Properties* erkannt, dass es sich um ein konkretes `Handle` einer `TestRun`-Instanz handelt. Das MDB wandelt das XML-Format wieder in ein `Handle`-Objekt um. Danach wird es der `EJBLocalObject`-Schnittstelle des `TestControllers` zur Ausführung des korrespondierenden Testprozesses übergeben mit:

```
public
    void runTest(Handle handle)
throws
    java.lang.IllegalArgumentException,
    com.byteacademy.research.base.CommunicationException,
    javax.ejb.FinderException,
    java.lang.InterruptedException;
```

Über das `Handle` holt der `TestController` die `TestRun`-Entität aus der Datenbank und hat damit Zugriff auf die Daten der `TestConfiguration`. Bevor der Test gestartet wird, wird zunächst mit der Methode:

```
public
    void createSnapShot(Handle handle, TestContext context, String label)
throws
    java.lang.IllegalArgumentException,
    com.byteacademy.research.base.CommunicationException,
    javax.ejb.FinderException;
```

des `EJBLocalObject` von `TestController` eine `SnapShotStatistic` der relevanten Tabellen der J2EE-Anwendung erzeugt und mit diesen Informationen eine neue `SnapShotStatistic` in der Datenbank erzeugt. Das `Handle` wird benötigt um die korrespondierende `TestRun`-Instanz zu finden, zu welcher die `SnapShotStatistic` aggregiert werden soll. Der `TestContext` beinhaltet die Start- und Endezeit der Testausführung und wird zusammen mit dem `TestRun-Handle` als Eingrenzungsparameter für die Ermittlung, welche Datensätze durch diesen Prozess erzeugt wurden, benötigt. In diesem Fall wurden selbstverständlich noch keine Werte erzeugt, es werden deshalb nur Werte für die *All-*

Kategorie gefunden (wenn ein initiales Laden stattgefunden hat – wovon man ausgehen kann), die *Created*- und *Modified*-Kategorie sind bei 0 Datensätzen.

Als nächstes wird der `RequestCase` ausgewertet und in diesem Fall eine Instanz der Klasse `com.byteacademy.research.tts.tcs.test.NewOrderProcessTestSuite` erzeugt. Diese Klasse basiert auf dem JUnit-Framework (näheres zu JUnit in [Mas03]) mit leicht modifizierter Basis-Klasse. Durch das Ausführen der `NewOrderProcessTestSuite` wird eine neue Instanz der Klasse `NewOrderTransactionProcess` erzeugt und der `Context` mit den notwendigen Daten befüllt. In diesem Fall ist es ein neuer Auftrag mit einer zufälligen Anzahl von Auftragszeilen (zwischen 5 und 15) sowie dem betreffenden Kunden. Diese Daten werden in einem `OrderDataBO`-Objekt, einer Liste aus `OrderLineBO`-Objekten, sowie einem `CustomerBO`-Objekt gehalten. Sind die Daten gefüllt wird in der Basis-Testklasse (`TTSBaseTestCase`) die Methode

```
protected
    void doRequest(Process process)
throws
    CommunicationException
```

aufgerufen. In dieser Methode wird das befüllte `Context`-Objekt dem erzeugten Prozess übergeben. Danach wird das Transaktionsattribut (`TransactionMode`) ausgewertet, um den `ProcessManager` mit der richtigen Methode aufzurufen. Hier hat das Transaktionsattribut den Wert *Required*. Es wird also eine Transaktion benötigt. Damit ruft der `TestController` mit dem *NewOrder*-Testprozess als Parameter die `EJBObject`-Methode:

```
public
    Context executeProcessWithinTransaction(Process process)
throws
    java.lang.IllegalArgumentException,
    com.byteacademy.research.base.CommunicationException,
    java.rmi.RemoteException;
```

des `ProcessManager` auf. Damit wurde der Testprozess dem SUT zur Ausführung übergeben. Wenn der Wert von `ShallLoopInfinite` auf „true“ steht sorgt der `TestController` dafür, dass nach der Rückkehr der Methode vom `ProcessManager` dieser beschriebene Vorgang solange wiederholt wird bis das Abbruchkriterium (alle Testprozesse für welche `ShallLoopInfinite` „false“ ist sind beendet worden) erfüllt ist. Ansonsten wird dieser Vorgang nur einmal ausgeführt.

C.5.4 Ausführung des Testprozesses durch den ProcessManager

Der `ProcessManager` initialisiert die *Logging*-Infrastruktur des Prozesses und ruft die *`startProcess`*-Methode auf dem `TestContext` auf, damit der aktuelle Zeitstempel genommen wird. Danach ruft er die Methode:

```
public
    void execute()
throws
    CommunicationException,
    UnknownApplicationException;
```

auf dem `NewOrderTransactionProcess` auf. Der Prozess wird in der Weise ausgeführt wie im Anhang C.4 beschrieben. Im folgenden wird sich deshalb nur auf die Interaktion des `NewOrderTransactionProcess` mit dem `ProcessManager` und dem `TestController` konzentriert. Der `NewOrderTransactionProcess` definiert die folgenden Testpunkte:

- `TEST_SPOT_INIT = "01-Init"`
- `TEST_SPOT_EXIT = "02-Exit"`
- `TEST_SPOT_FETCH_REFERENCE_DATA_BEFORE = "10"`
- `TEST_SPOT_FETCH_REFERENCE_DATA_AFTER = "11"`
- `TEST_SPOT_INCREASE_ORDER_NUMBER_BEFORE = "12"`
- `TEST_SPOT_INCREASE_ORDER_NUMBER_AFTER = "13"`
- `TEST_SPOT_STORE_ORDER_BEFORE = "14"`
- `TEST_SPOT_STORE_ORDER_AFTER = "15"`
- `TEST_SPOT_STORE_ORDER_LINE_BEFORE = "16"`
- `TEST_SPOT_STORE_ORDER_LINE_AFTER = "17"`

Zum Beginn und Ende des Prozesses sowie jeweils vor dem Beginn und nach der Ausführung einer `Action` existiert ein Testpunkt. Was an diesen Testpunkten geschieht entscheidet der Testprozess bzw. die `TestConfiguration`. Es sind prinzipiell drei Möglichkeiten denkbar. Bei der ersten Möglichkeit würde der Testpunkt als solcher ignoriert oder nur benutzt, um lokale Funktionalitäten auszuführen. In der zweiten Variante würde der Testpunkt dazu benutzt werden, um ein `TestEvent` zu erzeugen und an den `TestController` zu übertragen. Bevor der `TestEvent` an den `TestController` übertragen wird wird überprüft, ob dies überhaupt erwünscht ist (`ReportingToTcsRequired`). Wenn ja, dann wird die Unterbrechung der Businesslogik-Ausführung im `TestContext` mit der Methode *`interruptExecutionBusinesslogic`* protokolliert. Danach wird mit der Methode:

```
public
    void logTestEvent(Handle handle, String testSpot,
                                                LogEntryObject logObj)
throws
    java.lang.IllegalArgumentException,
    com.byteacademy.research.base.CommunicationException,
    java.rmi.RemoteException;
```

Beispiel:

```
logTestEvent(handle, TEST_SPOT_STORE_ORDER_BEFORE, getLogObject());
```

der `TestEvent` an die `EJBObject`-Schnittstelle des `TestControllers` übergeben. Mittels des `TestRun-Handles` welches als Parameter mit übergeben wird, kann der `TestController` die entsprechende *TestRun*-Entität von der Datenbank laden. Aus dem übergebenen `TestEvent` (in diesem Fall das `LogEntryObject`) wird eine `Event`-EntityBean befüllt, auf die Datenbank gespeichert und zu der `TestRun`-Instanz aggregiert. Danach kehrt die *`logEvent`*-Methode zum `NewOrderTransactionProcess` zurück, die Wiederaufnahme der Ausführung wird dem `TestContext` über *`resumeExecution-Businesslogic`* bekannt gegeben, bevor der Prozess weiterläuft. Die dritte Möglichkeit hätte die Ausführung des Testpunktes als möglichen Synchronisationspunkt zur Folge. Dazu ruft der `NewOrderTransactionProcess` die Methode:

```
protected
    void executeTestSpot(String spotId, String spotName, BO bo)
throws
    CommunicationException;
```

seiner Basisklasse `Task` auf. Hier wird wie schon im oben dargestellten Fall des `Test-Events` überprüft ob überhaupt eine Kommunikation mit dem `TestController` erwünscht ist, falls ja wird die Unterbrechung dem `TestContext` zur Kenntnis gebracht. Danach wird über die `EJBObject`-Schnittstelle des `TestControllers` die Methode:

```
public
    TestContext reportToExecutionMonitor(
                    TestContext context,
                    String testSpot,
                    BO bo )
throws
    java.lang.IllegalArgumentException,
    com.byteacademy.research.base.CommunicationException,
    java.rmi.RemoteException;
```

aufgerufen. Es wird der `TestContext` übergeben, die symbolische Bezeichnung des Testpunktes sowie optional ein transientes `BO` (z.B. `OrderDataBO`). Mit der Übergabe eines `BO` wird der `TestController` dazu aufgefordert, den Inhalt des `BOs` als *ModifiedObject*-Entität zu speichern und an die *TestRun*-Entität zu aggregieren, welche durch das `Handle`-Objekt des `TestContexts` referenziert wird. Der `TestController` benutzt dazu seine eigene Methode

```
public
    void storeReportedDataToTestRun(
                        Handle handle,
                        String testSpot,
                        BO bo )
throws
    java.lang.IllegalArgumentException,
    com.byteacademy.research.base.CommunicationException;
```

Der `TestController` prüft danach ob sich der aufrufende Testprozess schon an seinem Synchronisationspunkt befindet. Dazu prüft er den übergebenen symbolischen Bezeichner gegen den Wert des Feldes `SynchronizeSpot` aus der dem Testprozess zugrundeliegenden `TestConfiguration`. Stimmen diese nicht überein kehrt der Methodenaufruf wieder zum `NewOrderTransactionProcess` zurück. Stimmen im anderen Fall die Werte überein wird als nächstes überprüft ob sich auch alle anderen Testprozesse des `TestCases` an ihrem vorgesehenen Synchronisationspunkt befinden. Testprozesse welche keinen Synchronisationspunkt definiert haben, werden bei dieser Abfrage nicht berücksichtigt. Dazu wird die Methode:

```
public
    boolean isAllTestCaseInPlace(
                String testSuiteRunId,
                String testCaseId )
throws
    javax.ejb.FinderException,
    com.byteacademy.research.base.CommunicationException,
    java.lang.InterruptedException;
```

solange aufgerufen (aus Effizienzgründen mit einer kleinen Verzögerung von einer Sekunde) bis die anderen Testprozesse ebenfalls ihren Synchronisationspunkt erreicht haben, d.h. der Methodenaufruf wird solange blockiert und damit auch die Ausführung des `NewOrderTransactionProcesses`. Vor dem ersten Aufruf dieser Methode wird der Status des Testprozesses auf `STATUS_RUN_WAIT` gesetzt. Sobald auch die anderen Testprozesse ihren Synchronisationspunkt erreicht haben lässt der `TestController` die Aufrufe

der Testprozesse in der unter ReleaseSynchronizationOrder definierten Reihenfolge und der in ReleaseDelay angegebenen Verzögerung wieder zurückkehren. Der Status des jeweiligen Testprozesses wird auf STATUS_RUN_RUNNING gesetzt.

Ein weiterer Faktor welche die Verzögerung der Rückkehr beeinflusst ist, ob auf eine Blockierungssituation (ShallWaitOccure) getestet werden soll. Ist dies der Fall wird die Rückkehr des Methodenaufrufs pro Testprozess nochmals um zweimal den Wert von MinTimeForWait verzögert. Damit ist eine eindeutige Erkennung einer Blockierungssituation für einen Testprozess gesichert.

Bevor der Methodenaufruf nun zurückkehrt wird noch aus der TestConfiguration des Testprozesses ermittelt ob und wenn ja, welcher Fehlerfall ausgeführt wird. Es wird der Klassenname der ErrorCase-Klasse in den TestContext gesetzt. Im NewOrderTransactionProcess angekommen wird noch in der technischen Methode der Basisklasse überprüft ob ein ErrorCase gesetzt ist, falls ja wird die Klasse erzeugt und ausgeführt. Falls nein, wird die Wiederaufnahme der Ausführung der *Businesslogik* des Prozesses dem TestContext signalisiert. Danach verlässt der Prozess den Testpunkt und läuft in seiner Verarbeitung weiter.

Die zehn Prozesse welche durch die Basislast-TestConfiguration erzeugt wurden haben alle den Wert von ReportingToTcsRequired auf „false“, d.h. bei der Überprüfung innerhalb des NewOrderTransactionProcess wäre entschieden worden, dass eine Kommunikation an den TestController nicht erwünscht ist und damit wäre die Ausführung des Prozesses weiter gelaufen. Die beiden anderen TestConfiguration-Datensätze waren im Gegensatz so konfiguriert, dass sie mit dem TestController kommunizieren sollen. Nur so kann die Ausführung des TestCases aufeinander abgestimmt werden. In diesem Fall ist die Transaktion T1 (TestConfiguration CONFIG_002) ein NewOrderTransactionProcess. Sie wird unmittelbar nach der letzten Aktion der *Businesslogik* am letzten Testpunkt (02-Exit) durch die *executeTestspot*-Methode und des damit verbundenen Aufrufs der *reportToExecutionMonitor*-Methode des TestControllers blockiert. Zwar hat dieser Testprozess seinen Synchronisationspunkt gemäß TestConfiguration erreicht, jedoch die Transaktion T2 (TestConfiguration CONFIG_003), welche ebenfalls ein NewOrderTransactionProcess ist, noch nicht den ihren. Für die Transaktion T2 ist als Synchronisationspunkt der Testpunkt mit der Nummer 10 (TEST_SPOT_FETCH_REFERENCE _DATA_BEFORE) ausgewählt worden, da unmittelbar danach die Ausführung des Testprozesses wegen Ressourcen-Überschneidung blockiert werden muss. Dies ist die in der Anforderungsanalyse aufgezeigte kritische Stelle für diese Transaktionskonstellation. Wenn die Isolation nicht korrekt funktioniert, würde die Transaktion T2 ohne Blockade weiterlaufen und damit eine falsche Auftragsnummer zugewiesen bekommen.

Die Transaktion T2 wird gemäß Reihenfolge als erste Transaktion vom `TestController` wieder freigegeben, erst nach einem Verstreichen von der doppelten `MinTimeForWait` zuzüglich des `ReleaseDelay` wird die Transaktion T1 freigegeben. Die Transaktion T2 ruft nach ihrer Rückkehr sofort auf dem `TestContext` die Methode *`startPossibleWait`* und führt dann die erste Action (`FetchReferenceDataAction`) aus und tätigt damit ihren ersten Zugriff auf von der Transaktion T1 gesperrte Daten-Entitäten (*Customer*-Entität). Die Transaktion T2 wird blockiert. Nach der obigen Verzögerung kehrt nun auch Transaktion T1 zurück. Sie befand sich am letzten Testpunkt kurz vor verlassen der *`execute`*-Methode des `NewOrderTransactionProcesses`. Der Aufruf kehrt zum `ProcessManager` zurück, welcher dann den Aufruf zum `TestController` zurückkehren lässt. Nach dem Verlassen der Bean-Klasse des `ProcessManagers` führt der EJB-Container des SUT ein Commit auf die Transaktion T1 aus, da das Transaktionsattribut *Required* war und die Transaktion von ihm gestartet wurde.

Nachdem Commit von Transaktion T1, läuft nun auch Transaktion T2 weiter. Es wird auf dem `TestContext` *`stopPossibleWait`* aufgerufen. Damit sind der Beginn und das Ende des `TestForWait` im `TestContext` notiert und können am Ende des Testfalls ausgewertet werden. Die Transaktion T2 durchläuft nun den Rest des `NewOrderTransactionProcesses` in derselben Weise wie Transaktion T1. Der Aufruf kehrt an den `ProcessManager` zurück und der EJB-Container führt auch auf dieser Transaktion ein Commit bei der Rückkehr des Aufrufs an den `TestController` durch.

C.5.5 Evaluierung des Testfalls

Zuletzt wird für jeden Testprozess wieder eine *SnapShotStatistic*-Entität in der schon beschriebenen Weise angefertigt und dann die noch immer *NewOrder*-Transaktionen durchführenden Testprozesse (`ShallLoopInfinite` == true) beendet. Danach wird der `EvaluationCase` für die Testprozesse der Transaktion T1 und T2 ausgeführt. Dies ist bei unserer Konfiguration die Klasse `com.byteacademy.research.tts.tcs.test.NewOrderProcessCheckSuite`. In ihr werden die Werte nach der Ausführung des `TestCases` mit den erwarteten Werten verglichen. In diesem Fall gilt, da kein Fehler aufgetreten ist, folgendes:

1. Auftragsnummer für T1 ist niedriger als die von T2 (auf gar keinen Fall gleich).
2. Die Auftragsnummer von T1 existiert nicht doppelt in einem Auftrag (da als Basislast auch *NewOrder*-Transaktionen ausgeführt wurden muss diese Bedingung ebenfalls überprüft werden).

3. Der Test der Transaktion T2 auf eine Blockade durch eine aufgetretene Sperre muss erfolgreich sein.
4. Die in Kapitel 6.4 definierten Kriterien müssen erfüllt sein.
5. Die Entitäten auf der Datenbank müssen mit den Daten der `ModifiedObjects` für den in der `TestConfiguration` festgelegten `ReferenceObjectSpot` übereinstimmen.
6. Die `SnapShotStatistic` für die Transaktionen müssen die Anzahl der erzeugten und modifizierten Datensätze reflektieren.

Wäre ein Fehlerfall aufgetreten, wäre die Transaktion T1 zurückgerollt worden. Dann müssten folgende Bedingungen gelten:

1. Die Entitäten welche durch die Transaktion T1 neu erzeugt wurden (*OrderData*, *OrderLine*, *NewOrder*) dürfen nicht auf der Datenbank zu finden sein. Dieser Check kann über den Vergleich mit den Daten der `ModifiedObjects` erfolgen.
2. Der Test der Transaktion T2 auf eine Blockade durch eine aufgetretene Sperre muss erfolgreich sein.
3. Die in Kapitel 6.4 definierten Kriterien müssen erfüllt sein.
4. Die `SnapShotStatistic` für die Transaktion T2 muss die Anzahl der erzeugten und modifizierten Datensätze reflektieren. Die `SnapShotStatistic` für die Transaktion T1 muss reflektieren, dass keine Objekte erzeugt oder modifiziert wurden.

Nachdem für jeden Testprozess der `EvaluationCase` ausgeführt worden ist wird das Ergebnis ebenfalls bei der `TestRun`-Instanz abgelegt, sodass es jederzeit abrufbar ist. Als letztes wird der Status der `TestRun`-Instanzen noch auf beendet gesetzt:

- `STATUS_RUN_FINISHED_SUCCESSFUL`, wenn keine Fehler aufgetreten sind und die erwarteten Werte mit den tatsächlichen Werten übereinstimmt.
- `STATUS_RUN_FINISHED_WITH_ERRORS`, wenn technische Fehler (*Exceptions*) innerhalb des `EvaluationCase` aufgetreten sind.
- `STATUS_RUN_FINISHED_WITH_FAILURES`, wenn fachliche Fehler innerhalb des `EvaluationCase` aufgetreten sind.
- `STATUS_RUN_FINISHED_WITH_EXCEPTION`, wenn bei der Ausführung des `TestCases` innerhalb des `TestControllers` ein Fehler aufgetreten ist. Dies sollte im Normalfall nicht vorkommen.

Danach ist die Ausführung des aktuellen `TestCases` beendet und der nächste kann ausgeführt werden. War dies – wie in diesem Fall – der letzte `TestCase` ist die Ausführung der `TestSuite` beendet.

Der `TestClient` welcher während der Testausführung immer wieder mittels seines `Handles` auf die `TestSuite`-Laufzeitinstanz beim `TestController` den Status der einzelnen Testprozesse erfragt hat, bekommt nun für alle Testprozesse der `TestSuite` das Endergebnis mitgeteilt. Damit lässt sich nun die Bewertungsmetrik (Kapitel 6.5) mit Daten versorgen.

Anhang D : Konfiguration und Deploymentprozess

D.1 Konfiguration und Deploymentprozess für JBoss

D.1.1 Konfiguration von JBoss

JBoss wurde so konfiguriert, dass eine Schreiboperation auf die Datenbank erst nach dem Aufruf der `ejbPostCreate` durchgeführt wird. Dadurch lassen sich die Datenbankoperationen für das Schreiben von Daten um die Hälfte reduzieren. Die Commit-Option für die J2EE-Anwendung wurde im allgemeinen *Deployment Descriptor* auf die Option A eingestellt. Dadurch wird dem EJB-Container mitgeteilt, dass er den alleinigen Zugriff auf die Datenquelle hat und somit nicht die Bean-Instanz zwischen den Transaktionen mit dem Zustand der Datenbank synchronisieren muss. Die Bibliotheken für den Zugriff auf die MaxDB-Datenbank (`sapdbc.jar`) werden in das JBoss-Server-Lib-Verzeichnis eingespielt (für die JBoss-Konfiguration des TCS muss noch zusätzlich die für die Websphere-Kommunikation notwendigen Bibliotheken eingespielt werden, wie im Kapitel 4.2.3 dargelegt).

Die *Datasource* wurde wie folgt konfiguriert:

```
<datasources>
  <local-tx-datasource>
    <jndi-name>RS_TTS_DS</jndi-name>
    <connection-url>jdbc:sapdb://localhost/RS_TTSDB</connection-url>
    <driver-class>com.sap.dbtech.jdbc.DriverSapDB</driver-class>
    <user-name>DBA</user-name>
    <password>DBA</password>
    <transaction-isolation>
          TRANSACTION_REPEATABLE_READ
    </transaction-isolation>
    <min-pool-size>5</min-pool-size>
    <max-pool-size>50</max-pool-size>
  </local-tx-datasource>
</datasources>
```

Die *Datasource*-Definition wird im JBoss-Deployment-Verzeichnis abgelegt, in einem speziellen Unterverzeichnis für das TTS (`<JBOSS_HOME>/deploy/rs_tts`).

D.1.2 Erstellung des Deployments

Die Entwicklung des Transaction Testsystems (TTS) fand unter Eclipse statt (näheres zu Eclipse in [Sha03] und [Gam03]). Der gesamte Buildvorgang welcher das EAR sowie den *Deployment Descriptor* erstellt wird mittels eines ANT-Skripts gesteuert. ANT ist eine Java-Bibliothek mit einer XML-basierten Sprache, um den Buildprozess in Softwareprojekten zu automatisieren (näheres in [Til02]). Zur Erstellung der *Deployment Descriptors* wurde das Quellcode-Generierungstool XDoclet eingesetzt. Ebenso wie ANT ist es eine Java-Bibliothek. XDoclet-Anweisungen werden mit Tags als Kommentare direkt im Quellcode einer betreffenden Klasse angegeben. Anweisungen welche sich auf eine komplette Bean beziehen werden vor der Klassendeklaration angegeben. Anweisungen welche sich auf eine Methode beziehen werden vor der entsprechenden Methode als Kommentare geschrieben. Beispielsweise beinhalten die folgenden XDoclet-Anweisungen in der `CustomerBean`-Klasse alle Informationen zur Erzeugung der 1-n Relation zwischen *District* und *Customer*:

```
/**
 * @ejb.interface-method view-type="local"
 *
 * @ejb.relation
 *    name="hasCustomer"
 *    role-name="District"
 *
 * @ejb.value-object
 *    aggregate="com.byteacademy.research.tts.tpcc.bo.CustomerBO"
 *    aggregate-name="Customer"
 *    members="com.byteacademy.research.tts.tpcc.po.Customer"
 *    members-name="Customer"
 *    relation="external"
 *    type="java.util.Collection"
 *
 **/

public abstract Collection getCustomers();
```

In einem *Precompile*-Vorgang werden aus den einzelnen Anweisungen Quellcode und *Deployment Descriptor* erstellt. In unserem Fall wurden zusätzlich noch Datenobjekte (`BOs`) generiert, ebenso Hilfsmethoden für das Befüllen eines `BOs` von einem EntityBean und umgekehrt. Der *Precompile*-Vorgang von XDoclet lässt sich über ANT steuern (mehr in [Wal03]). Sowohl der allgemeine als auch der für JBoss spezifische *Deployment Descriptor* wurde mit XDoclet und ANT erstellt. Das *Deployment* auf den JBoss-Applicationserver findet durch ein einfaches kopieren des durch den ANT-Build erstellten EARs sowie aller relevanten Konfigurationsdateien in ein Unterverzeichnis des JBoss-Deployment-Verzeichnisses statt (`<JBOSS_HOME>/deploy/rs_tts`). Durch die *Hot-Deployment*-Funktionalität von JBoss ist kein Stoppen bzw. Starten von JBoss notwendig damit die Anwendung aktiv wird. Eine evtl. schon existierende Version der Anwendung wird beendet und deinstalliert bevor die

neue Anwendung installiert wird. Wenn die Tabellen in der Datenbank noch nicht existieren, werden sie auf Basis des JBoss-spezifischen *Deployment Descriptors* automatisch angelegt.

D.2 Konfiguration und Deploymentprozess für Websphere

D.2.1 Konfiguration von Websphere

Die Commit-Option für die J2EE-Anwendung wurde im allgemeinen *Deployment Descriptor* auf die Option A eingestellt. Die Bibliotheken für den Zugriff auf die DB2-Datenbank (`db2jcc.jar`) werden in das Websphere-Lib-Verzeichnis eingespielt. Für die Kommunikation mit dem TCS müssen noch zusätzlich das JBoss-Client-JAR sowie das TCS-Client-JAR im Websphere-Lib-Verzeichnis eingespielt werden. Die Datenquellen werden über die Adminkonsole konfiguriert. Zur Konfiguration einer Datenquelle muss zunächst ein JDBC-Provider über den Menüpunkt *`Ressourcen->JDBC-Provider->Neu`* erstellt werden (in unserem Fall für DB2 ohne XA). Es muss neben dem Namen des JDBC-Providers auch der Klassenpfad für die DB2-JARs angegeben werden:

```
${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc.jar
${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cu.jar
${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cisuz.jar
```

sowie der Name der Implementierung des JDBC-Treibers:

```
com.ibm.db2.jcc.DB2ConnectionPoolDataSource
```

In einem nächsten Schritt kann nun die Datenquelle angelegt werden über den Menüpunkt *Datenquellen->Neu*. Die Konfiguration der Datenquelle ist in Tabelle 26 aufgelistet.

Tabelle 26 Konfigurationseinstellungen der TTS Datasource unter Websphere

Konfigurationsmerkmale	***Werte***
Name	RS_TTS_DB_PLAIN
JNDI-Name	jdbc/RS_TTS_DB_PLAIN
Über Container realisierte Transaktionspersistenz (CMP)	Verwenden Sie diese Datenquelle für die über Container realisierte Transaktionspersistenz ? -> Checkbox selektieren.
Größe des Anweisungs-Cache	10
Klassenname des Datasource-Helper	`com.ibm.websphere.rsadapter.DB2UniversalDataStoreHelper`
Aliasname für komponentengestützte Authentifizierung	draal/db2admin (bevor dieser Eintrag ausgewählt werden kann muss er über den Menüpunkt *Datenquellen->Dateneinträge für J2C-Authentifizierung->Neu* eingetragen werden)
Aliasname für containergestützte Authentifizierung	draal/db2admin (bevor dieser Eintrag ausgewählt werden kann muss er über den Menüpunkt *Datenquellen->Dateneinträge für J2C-Authentifizierung->Neu* eingetragen werden)
databaseName	RS_TTS_1
driverType	4
serverName	localhost
traceLevel	0 für die eigentlichen Tests, 1024 für Studienzwecke

Da diese Datenquellen-Definition für CMP benutzt werden soll wird von Websphere an den angegebenen JNDI-Namen die Endung `_CMP` angehängt. Etwas versteckt unter dem Menüpunkt *Datenquellen->Benutzerspezifische Angaben* liegen die Angaben für den Datenbanknamen und den Server.

D.2.2 Erstellung des Deployments

Bei Websphere wurde ein etwas anderer Ansatz als bei JBoss gewählt, da XDoclet zwar eine Reihe von Applicationservern (z.B. Bea Weblogic, Oracle, Borland, Sun) mit ihren spezifischen Funktionalitäten unterstützt, hingegen Websphere jedoch nur äußerst rudimentär. Anstelle von XDoclet wurde deshalb der für Websphere typische Entwicklungsprozess über das *Websphere Application Developer Studio (WAD)* gewählt. Da für das SUT immer derselbe Code verwendet wird und nur die Konfiguration und das *Deployment* unterschiedlich ist wurde das fertige EAR aus dem vorherigen JBoss-*Deployment* in das WAD importiert (Menüpunkt `File->Import->EAR File`). Der WAD-Import erstellt zwei Projekte für welche die Namen anzugeben sind:

- das EAR-Projekt in dem der *Assembly Descriptor* abgelegt ist sowie die unterstützenden Bibliotheken.
- das EJB-Projekt, welches als Modul den *Deployment Descriptor* enthält sowie allen Quellcode bzw. *Class*-Files für das EJB-Modul.

Vor dem Import des EARs empfiehlt es sich noch die Default-*Datasource* in den Einstellungen des WAD zu überprüfen (Menüpunkt `Window->Preferences->Data`), ob das richtige Datenbankmanagementsystem eingestellt ist. In diesem Fall sollte es *DB2 Version 8.1* sein. Dadurch wird beim Import schon die richtige `BackendId` für die Persistierung vergeben und die Websphere-spezifischen *Deployment Descriptors* mit den richtigen Daten gefüllt. Des weiteren werden beim Import automatisch die Projekte angelegt und die Class-Dateien aus dem EAR extrahiert. Es wird auch der *Assembly* und *Deployment Descriptor* ausgewertet und in separaten Bereichen angezeigt.

Als nächstes müssen die Daten für die Persistierung (OR-Mapping) in der Datenbank noch fein konfiguriert werden. Dies wird in der *Data*-Perspektive des WAD durchgeführt. Es wird hier beispielsweise die Datenbank und das Schema eingegeben, darüber hinaus können noch datenbankspezifische Optimierungen vorgenommen werden, z.B. eine Vergabe von Indizes. Aus dem OR-Mapping kann ein DDL-Skript generiert werden für die Erstellung der Tabellen über DB2-Mittel. Es ist jedoch auch möglich, direkt aus dem WAD die Tabellen auf einer Datenbank zu erstellen. Dazu wird in der *Data*-Perspektive des WAD eine neue DB-Verbindung definiert und mit den relevanten Daten für den Datenbankzugriff über JDBC befüllt, welche schon im vorherigen Konfigurationskapitel aufgelistet wurden. Danach wird das gewünschte *Backend* im EJB-Projekt selektiert und aus dem Kontext-Menü der Menüpunkt `Export to server...` aufgerufen, die vorher definierte DB-Verbindung ausgewählt und ausgeführt. Falls bei Teilen der DB-Operationen Fehler aufgetreten sind wird der Status aller Operationen angezeigt und es ist dem Benutzer überlassen ob er ein Commit durchführt oder ein Rollback. Nach dem Festschreiben sind die Tabellen im Schema angelegt und können von der

Anwendung benutzt werden. Für ein erfolgreiches Anlegen aller Tabellen werden Tabellenbereiche (*Tablespaces*) in DB2 benötigt, welche ein *Pagesize* von 4, 8 und 16 kbytes haben. Die *Pagesize* ist ausschlaggebend wie groß ein Datensatz in einer Tabelle dieses Tabellenbereiches werden darf.

In der J2EE-Perspektive müssen noch Informationen nachgetragen werden, welche sich bisher nur im JBoss-spezifischen *Deployment Descriptor* befanden (und in den XDoclet-Tags welche vom WAD nicht ausgewertet werden). Es handelt sich dabei um die JNDI-Namen für die Remote-Schnittstellen der einzelnen `EJBHomes` und `EJBObjects` sowie die JNDI-Namen der Komponentenbeziehungen zwischen aufrufendem Bean und aufgerufenem Bean, welche mit einem speziellen Präfix (`java:comp/env`) versehen werden.

Danach kann in der J2EE-Perspektive über den Menüpunkt *`Generate Deployment Code`* der für die Installation der Anwendung in Websphere notwendige Code für das EAR-Projekt erzeugt werden. Hierbei wird zunächst der allgemeine und der spezifische *Deployment Descriptor* erstellt, danach werden mit dem RMIC-Compiler die Stubs, Ties und Helferklassen für die EJBs erzeugt und anschließend compiliert. Um die Anwendung in Websphere zu installieren, wird das gesamte EAR-Projekt als EAR in das Dateisystem exportiert (*`File->Export->EAR File`*). Dies beinhaltet auch das EJB-Projekt welches mit dem EAR-Projekt assoziiert ist.

Vom Dateisystem wird es nun mittels der Websphere Adminkonsole in die Websphere-Instanz des SUT installiert (Menüpunkt *`Anwendungen->Enterprise Anwendungen->Installieren`*). Es wird ein Dialog angezeigt mit dem die EAR-Datei im Dateisystem gesucht und selektiert werden kann. Danach wird die Installation gestartet. Die Installationslogik wertet das EAR, im speziellen den *Deployment Descriptor*, aus. Danach werden noch benötigte Konfigurationsdaten über einen Konfigurationsdialog abgefragt. Hier müssen noch die folgenden Informationen für den Menüpunkt *`Zuordnung der Standarddatenquelle für Module mit 2.0-Entity-Beans angeben`* und *`Datenquellen für alle 2.0-CMP-Beans zuordnen`* bereitgestellt werden:

- JNDI-Name der zu verwendenden Datenquelle: hier muss die zuvor definierte Datenquelle `RS_TTS_DB_PLAIN_CMP` ausgewählt werden.
- Ressourcen-Berechtigung für die Datenquelle: hier den Wert `Pro Verbindungs-Factory` einstellen.

Die weiteren Seiten des Konfigurationsdialogs haben nur Informationscharakter und können über die Aktion *`Weiter`* übersprungen werden. Als letzter Schritt erscheint eine Zusammenfassung auf der die Aktion *`Fertigstellen`* ausgewählt werden sollte. Damit wird das EAR in die Websphere-Serverinstanz installiert. Wenn der Installationsvorgang erfolgreich abgeschlossen wurde kann die Anwendung *Transaction TS* unter dem Menüpunkt *`Anwendungen->Enterprise Anwendungen`* ausgewählt werden. Über den Menü-

punkt *Anwendungen->Enterprise Anwendungen->Starten* wird die Anwendung aktiviert. Wenn die Anwendung erfolgreich gestartet wurde ist sie nun bereit, Anfragen zu bearbeiten.

Anhang E : Inhalt der CD

Auf der beigefügten CD befinden sich elektronische Versionen der Diplomarbeit sowie die in dieser Arbeit benutzten Programme, Konfigurationen, Quellcode und getesteten EARs.

Die CD beinhaltet folgende Unterverzeichnisse:

- **Documents:** hier sind die elektronischen Versionen der Diplomarbeit abgelegt. Für die OpenOffice-Version (*.sxw) ist die Programmversion 1.1.0 erforderlich, da sonst Abweichungen in der Formatierung auftreten können.
- **Resources:** hier sind alle in der Diplomarbeit verwendeten und in elektronischer Form verfügbaren Ressourcen abgelegt.
- **Installables:** in diesem Verzeichnis sind die installationsfähigen Versionen der Programme abgelegt, welche verwendet wurden. Bei kommerzieller Software wurde aus lizenzrechtlichen Gründen nur eine Trial-Version beigelegt.
- **Development:** in diesem Verzeichnis sind die Projekte aus Eclipse bzw. dem Websphere Studio for Application Developer (WAD) abgelegt. Das Eclipse-Projekt beinhaltet den kompletten Quellcode des Transaction Testsystems (TTS) sowie das ANT-Buildskript zur Erstellung der EARs. Das WAD-Projekt beinhaltet die Version mit den modifizierten *Deployment Descriptors* für Websphere.
- **Distribution:** beinhaltet die *Deployment*-Verzeichnisse des JBoss- und Websphere-Applicationservers. Die hier abgelegten EARs wurden in den Tests verwendet.

Selbstständigkeitserklärung

Ich erkläre, dass ich die hier vorgelegte Arbeit selbstständig und ohne fremde Hilfe verfasst und andere als die von mir angegebenen Quellen und Hilfsmittel nicht benutzt habe.

Filderstadt, den 30.09.2004

Oliver Raible
(Matrikelnr.: 1640757)

CD zur Diplomarbeit

Transaktionsverarbeitung in J2EE Systemen