Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Modulare Integration von konfigurierbaren Transaktionsmechanismen in Cloud-basierte Datenspeichersysteme

Verfasser:

Ivonne Schröter

14. September 2012

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Thomas Leich,
Dipl.-Inform. Thomas Thüm

Schröter, Ivonne:
Modulare Integration von konfigurierbaren Transaktionsmechanismen in Cloud-basierte Datenspeichersysteme
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2012.

Danksagung

Bei all denjenigen, die mich auf dem Weg durch das Studium begleitet und beim Erstellen dieser Diplomarbeit unterstützt haben, möchte ich mich herzlich bedanken. Besonderer Dank gebührt hierbei:

- Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Wirtsch.-Inf. Thomas Leich und Dipl.-Inform. Thomas Thüm für die intensive Betreuung meiner Diplomarbeit sowie für ihre zahlreichen Ratschläge und Anmerkungen.
- Matthias Ritter, Katja Matthes und Martin Schäler für ihre kritischen Anmerkungen zu Form und Inhalt sowie Rechtschreibung und Grammatik.
- meiner Familie, die mich unterstützt hat, so gut es ihnen möglich war und welche mir durch ihre Geduld und ihr Verständnis in dieser Zeit einen starken Rückhalt gegeben hat.
- sowie abschließend allen Mitarbeitern der Firma METOP, die mir stets mit Rat und Tat beiseite standen.

Inhaltsverzeichnis

Abbildungsverzeichnis v

Tabellenverzeichnis ix

Abkürzungsverzeichnis xi

1 Einleitung 1

2 Grundlagen 5
2.1 Cloud Computing . . . 6
2.2 Transaktionsverwaltung . . . 19
2.3 Speichermanager . . . 28
2.4 Erweiterung der objektorientierten Programmierung . . . 32
2.4.1 Aspektorientierte Programmierung . . . 35
2.4.2 Merkmalsorientierte Programmierung . . . 37
2.4.3 Aspectual Feature Modules . . . 38
2.5 Zusammenfassung . . . 40

3 Merkmalsanalyse der Transaktionsunterstützung für Cloud Computing 41
3.1 Notation des Merkmalsdiagramms . . . 42
3.2 Transaktionsmanager . . . 44
3.2.1 Transaktion . . . 44
3.2.2 Kaskadierende Abbrüche und Wartegraph . . . 46
3.2.3 Scheduler . . . 47
3.2.4 Transaktionsmodell . . . 54
3.2.5 Kompensierende Transaktionen . . . 57
3.3 Speichermanager . . . 58
3.3.1 Puffermanager . . . 58
3.3.2 Recoverymanager . . . 59
3.4 Zusammenfassung . . . 64

4 Merkmalsbasierter Entwurf anhand von Aspekten und Verfeinerungen 69
4.1 Notation des Kollaborationsdiagramms . . . 69
4.2 Basis- und Kernmerkmale . . . 70
4.2.1 Basisklassen . . . 70
4.2.2 Transaktionsunterstützung . . . 71
4.3 Variable Merkmale des Transaktionsmanagers . . . 73
4.3.1 Transaktionszustände . . . 74
4.3.2 Kompensierende Transaktionen . . . 75
4.3.3 Erweiterte Transaktionsmodelle . . . 76
4.3.4 Transaktionszeitmarken . . . 78
4.3.5 Kaskadierende Abbrüche und Wartegraph . . . 79

4.3.6 Synchronisationsverfahren 80
4.4 Variable Merkmale des Speichermanagers 85
4.4.1 Log 86
4.4.2 Datenpuffer 87
4.4.3 Sicherungspunkte 87
4.4.4 Recovery-Verfahren 88
4.5 Zusammenfassung 89

5 Ergebnisdiskussion **93**
5.1 Merkmalsdiagramm einer Transaktionsunterstützung 93
5.2 Kollaborationsdiagramm 94
5.3 Probleme und Erkenntnisse 96
5.3.1 Erweitern von Parameterlisten 96
5.3.2 Optionales Weben 96
5.3.3 Hook-Methoden 97
5.3.4 Aspect Refinement 97
5.4 Zusammenfassung 97

6 Ausblick und Zusammenfassung **99**
6.1 Ausblick 99
6.2 Zusammenfassung 100

Literaturverzeichnis **103**

Abbildungsverzeichnis

2.1 Die NIST *Cloud Computing*-Definition . . . 7
2.2 Wesentliche Besonderheiten von *Cloud Computing* . . . 8
2.3 Dienstmodell-Architekturen von *Cloud Computing* . . . 9
2.4 *Cloud*-Verwendungsmodelle . . . 10
2.5 *Public Cloud* . . . 11
2.6 *Private Cloud* . . . 11
2.7 *Hybrid Cloud* . . . 12
2.8 *Amazon S3* Entwickler-*Workflow* . . . 18
2.9 CAP Theorem . . . 20
2.10 Zwei-Phasen-Sperr-Protokoll . . . 21
2.11 Striktes Zwei-Phasen-Sperr-Protokoll . . . 22
2.12 *Timestamp-Ordering*-Algorithmus . . . 24
2.13 Optimierter *Timestamp-Ordering*-Algorithmus . . . 25
2.14 Validierung von Transaktionen . . . 26
2.15 *Multiversion Concurrency Control*-Algorithmus . . . 27
2.16 Mögliche Situation nach einem Systemfehler . . . 29
2.17 Eine Vererbungshierarchie (Auszug) . . . 32
2.18 Beispiel: Objektorientierte Implementierung der *Queue*-Klasse . . . 33
2.19 Beispiel: Erweiterung der *Queue*-Klasse . . . 34
2.20 Aspektcode weben . . . 36
2.21 *Collaborationsstack* . . . 38
2.22 Erweiterung eines Basisgraphen zu einem gewichteten Graph mit FOP . 38
2.23 Vor- und Nachteile von AOP und FOP . . . 39
2.24 Eine merkmalsorientierte Dekomposition eines aspektorientierten Designs 39
2.25 Erweiterung des gewichteten Graphen durch das Merkmal *Color* mit Hilfe von AFM . . . 40

3.1 Aufbau auf Basis einer Drei-Schichten-Architektur . . . 42
3.2 Beziehungsnotation im Merkmalsdiagramm . . . 43
3.3 Beziehungsnotation am Beispiel: Kraftfahrzeug . . . 43
3.4 Ausschnitt Merkmalsdiagramm: Transaktionsmanager . . . 44
3.5 Ausschnitt Merkmalsdiagramm: Transaktion . . . 44
3.6 Ausschnitt Merkmalsdiagramm: Transaktionszustand *Abgebrochen* . . . 45
3.7 Ausschnitt Merkmalsdiagramm: Kaskadierende Abbrüche und *Deadlock*-Erkennung durch Wartegraph . . . 46
3.8 Ausschnitt Merkmalsdiagramm: *Deadlock*-Auflösung . . . 47
3.9 Ausschnitt Merkmalsdiagramm: Scheduler . . . 48
3.10 Ausschnitt Merkmalsdiagramm: Sperrverfahren . . . 49
3.11 Ausschnitt Merkmalsdiagramm: Art und Überprüfung der Sperren und Freigaben . . . 49

3.12 Ausschnitt Merkmalsdiagramm: Benötigte Elemente des Zeitmarkenverfahrens . . . 50
3.13 Ausschnitt Merkmalsdiagramm: Varianten des Zeitmarkenverfahrens . . 51
3.14 Ausschnitt Merkmalsdiagramm: Versionierung durch die *Multiversion Concurrency Control* . . . 51
3.15 Ausschnitt Merkmalsdiagramm: Löschen alter Versionen der *Multiversion Concurrency Control* . . . 51
3.16 Ausschnitt Merkmalsdiagramm: *Constraint* durch annotationsfreie Verfahren . . . 52
3.17 Ausschnitt Merkmalsdiagramm: *Constraint* durch seriellen Scheduler . . 52
3.18 Ausschnitt Merkmalsdiagramm: Serialisierbarkeitsgraphentester . . . 53
3.19 Ausschnitt Merkmalsdiagramm: Optimistische Synchronisation . . . 54
3.20 Ausschnitt Merkmalsdiagramm: Transaktionspuffer . . . 54
3.21 Ausschnitt Merkmalsdiagramm: Transaktionsmodelle . . . 55
3.22 Ausschnitt Merkmalsdiagramm: Abbrüche von Subtransaktionen bei offen geschachtelten Transaktionen . . . 56
3.23 Ausschnitt Merkmalsdiagramm: Kompensierende Transaktionen . . . 57
3.24 Ausschnitt Merkmalsdiagramm: *Constraints* für die kompensierenden Transaktionen . . . 57
3.25 Ausschnitt Merkmalsdiagramm: Speichermanager . . . 58
3.26 Ausschnitt Merkmalsdiagramm: Puffermanager . . . 58
3.27 Ausschnitt Merkmalsdiagramm: Datenpuffer . . . 59
3.28 Ausschnitt Merkmalsdiagramm: Log-Puffer . . . 59
3.29 Ausschnitt Merkmalsdiagramm: *Recoverymanager* . . . 60
3.30 Ausschnitt Merkmalsdiagramm: Log . . . 60
3.31 Ausschnitt Merkmalsdiagramm: *redo*- und *undo*-Informationen . . . 61
3.32 Ausschnitt Merkmalsdiagramm: Sicherungspunkte . . . 62
3.33 Ausschnitt Merkmalsdiagramm: *Constraints* der *Recovery*-Strategien . . 62
3.34 Ausschnitt Merkmalsdiagramm: ARIES . . . 63
3.35 Ausschnitt Merkmalsdiagramm: Schattenspeicherverfahren . . . 64
3.36 Vollständiges Merkmalsdiagramm: Farblegende . . . 65
3.37 Vollständiges Merkmalsdiagramm: Transaktionsmanager . . . 66
3.38 Vollständiges Merkmalsdiagramm: Speichermanager . . . 67
3.39 Vollständiges Merkmalsdiagramm: *Constraints* . . . 68

4.1 Kollaborationsdiagramm: Legende . . . 70
4.2 Kollaborationsdiagramm: Basis . . . 70
4.3 Kollaborationsdiagramm: Transaktionsunterstützung . . . 71
4.4 Ablauf: Hinzufügen einer Transaktion . . . 72
4.5 Ablauf: Scheduler . . . 73
4.6 Ablauf: Speichermanager . . . 73
4.7 Kollaborationsdiagramm: Optionale Transaktionszustände . . . 74
4.8 Kollaborationsdiagramm: Kompensierende Transaktionen . . . 75
4.9 Kollaborationsdiagramm: Erweiterte Transaktionsmodelle Teil 1 . . . 76
4.10 Kollaborationsdiagramm: Erweiterte Transaktionsmodelle Teil 2 . . . 77
4.11 Kollaborationsdiagramm: Erweiterte Transaktionsmodelle Teil 3 . . . 78
4.12 Kollaborationsdiagramm: Transaktionszeitmarken . . . 78
4.13 Kollaborationsdiagramm: Kaskadierende Abbrüche und Wartegraph . . 79

4.14 Kollaborationsdiagramm: Serielle Synchronisation 80
4.15 Kollaborationsdiagramm: Serialisierbarkeitsgraphentester 81
4.16 Kollaborationsdiagramm: Optimistische Synchronisation 82
4.17 Kollaborationsdiagramm: Sperrverfahren 83
4.18 Kollaborationsdiagramm: Zeitmarkenverfahren 84
4.19 Kollaborationsdiagramm: *Multiversion Concurrency Control* 85
4.20 Kollaborationsdiagramm: Log und Log-Puffer 86
4.21 Kollaborationsdiagramm: Datenpuffer 87
4.22 Kollaborationsdiagramm: Sicherungspunkte 88
4.23 Kollaborationsdiagramm: *Recovery*-Verfahren 89

Tabellenverzeichnis

2.1 Übersicht: *Cloud*-Dienste Teil 1 16
2.2 Übersicht: *Cloud*-Dienste Teil 2 17
2.3 Verträglichkeitsmatrix für elementare Sperren 23
2.4 Kompatibilitätsmatrix für hierarchische Sperren 23

4.1 Kollaborationsdiagramm: Aspekte 90
4.2 Kollaborationsdiagramm: Verfeinerungen Teil 1 91
4.3 Kollaborationsdiagramm: Verfeinerungen Teil 2 92

5.1 Unterteilung: Merkmalsdiagramm 93
5.2 Unterteilung: Konkrete Merkmale 94
5.3 Unterteilung: Klassen und Aspekte 95
5.4 Aufteilung der Verfeinerungen auf Aspekte und Klassen 95

Abkürzungsverzeichnis

2PL . . . . . *Two Phase Locking*

ACID . . . . . *Atomicity*, *Consistency*, *Isolation*, *Durability*

AFM . . . . . *Aspectual Feature Modules*

AOP . . . . . *Aspect-Oriented Programming*

API . . . . . *Application Programming Interface*

ARIES . . . . . *Algorithm for Recovery and Isolation Exploiting Semantics*

AWS . . . . . *Amazon Web Services*

BOT . . . . . *Begin of Transaction*

C2PL . . . . . *Conservative Two Phase Locking*

CAP . . . . . *Consistency*, *Availability*, *Partition Tolerance*

EBS . . . . . *Elastic Block Storage*

EC2 . . . . . *Elastic Compute Cloud*

EOT . . . . . *End of Transaction*

FOP . . . . . *Feature-Oriented Programming*

GB . . . . . Gigabyte

IaaS . . . . . *Infrastructure as a Service*

IT . . . . . Informationstechnik

MGL . . . . . *Multi Granularity Locking*

MVCC . . . . . *Multiversion Concurrency Control*

NIST . . . . . *National Institute for Standards and Technology*

OOP . . . . . Objektorientierte Programmierung

PaaS . . . . . *Platform as a Service*

PB . . . . . Petabyte

PIMS . . . . . *Personal Information Management Systems*

RDS . . . . . *Relational Database Service*

S2PL . . . . . *Strict Two Phase Locking*

S3 . . . . . *Secure Storage Service*

SaaS . . . . . *Software as a Service*

SGT . . . . . Serialisierbarkeitsgraphentester

SQS . . . . . *Simple Queue Service*

TB . . . . . Terabyte

TO . . . . . *Timestamp-Ordering*

VM . . . . . Virtuelle Maschine

VPC . . . . . *Virtual Private Cloud*

VPN . . . . . *Virtual Private Network*

WAL . . . . . *Write Ahead Log*

XML . . . . . *Extensible Markup Language*

1 Kapitel 1
Einleitung

In der IT-Landschaft gewinnt *Cloud Computing* zunehmend an Bedeutung. Es ermöglicht das Mieten von Speicher, Infrastruktur und Software, die von Drittanbietern über das Internet bereitgestellt werden [Michelmann, 2010; Buxmann et al., 2011]. *Cloud Computing* zeichnet sich vor allem dadurch aus, dass die angebotenen Dienste skalierbar sind [Deussen et al., 2010] und ihre Nutzung bedarfsgerecht abgerechnet wird [Napper und Bientinesi, 2009; Mirzaei, 2009]. Auch im Bereich der Anwendungsentwicklung bietet *Cloud Computing* ein umfassendes Betätigungsfeld. Insbesondere verteilte Anwendungen, bei denen mehrere Nutzer gleichzeitig an einem Projekt zusammenarbeiten, stellen für die Entwickler eine spezielle Herausforderung dar. So bevorzugen viele Anbieter von *Cloud*-basierten Datenspeicherdiensten Verfügbarkeit und Partitionstoleranz über Konsistenzgarantien, da das CAP-Theorem besagt, dass in verteilten Systemen nur zwei der drei Punkte Verfügbarkeit, Partitionstoleranz und Konsistenz gewährleistet werden können [Brewer, 2000]. Dies führt dazu, dass die Implementierung von Transaktionseigenschaften in die Verantwortung des Anwendungsentwicklers fällt [Gropengießer und Sattler, 2011].

Eine Transaktionsunterstützung kann jedoch durch verschiedene Techniken und Mechanismen umgesetzt werden, was dazu führt, dass ein Anwendungsentwickler sich beispielsweise für ein konkretes Synchronisationsverfahren entscheiden und dieses entsprechend umsetzen muss und somit nicht die Möglichkeit hat, ohne zusätzlichen Aufwand auf ein anderes, vielleicht besser geeignetes Synchronisationsverfahren zu wechseln. So kann ein Entwickler zum Beispiel ein Zeitmarkenverfahren implementiert und in seine Anwendung integriert haben und im Nachhinein feststellen, dass er hauptsächlich Lesezugriffe verwalten muss. Da sich dafür die Umsetzung der *Multiversion Concurrency Control* (MVCC) anbieten würde, müsste er die MVCC implementieren, in die Anwendung integrieren und das Zeitmarkenverfahren insofern anpassen, dass es nur noch die Schreibzugriffe synchronisiert. Ebenso kann er weiterhin feststellen, dass er kaskadierende Abbrüche vermeiden möchte, was zur Folge hat, dass das Zeitmarkenverfahren zu einem strikten Zeitmarkenverfahren abgeändert werden muss. Anstatt diese Änderungen und Algorithmen mit größerem Zeitaufwand eigenständig einzubringen, ist es vorteilhafter eine große Auswahl bereits umgesetzter Algorithmen und Mechanismen zur Verfügung zu haben, die mit Hilfe einer Konfigurationsmöglichkeit individuell ausgewählt werden können und automatisch in die jeweilige Anwendung integriert werden.

Motivation

Die Motivation für diese Arbeit liegt in der Frage, wie es möglich gemacht werden kann, auch für NoSQL-Systeme strikte Konsistenzgarantien zu gewährleisten, wenn diese vom Anwendungsentwickler gewünscht werden. NoSQL-Systeme zeichnen sich vor allem dadurch aus, dass sie entwickelt wurden, um große Datenmengen zu verwalten. Des Weiteren sind NoSQL-Systeme im Gegensatz zu relationalen Datenbanksystemen besser skalierbar [Peterson, 2011], bieten aber gleichzeitig nur einen eingeschränkten Transaktionssupport. Dies führt dazu, dass sie zum Beispiel nur schwache Garantien bezüglich der Konsistenz bieten. Es ist für viele Anwendungen akzeptabel, keine strikte Konsistenzgarantie zur Verfügung zu haben, aber für ebenso viele Anwendungen ist sie notwendig und wünschenswert [Gropengießer und Sattler, 2011]. Für Entwickler dieser Anwendungen ist es von Interesse, Transaktionseigenschaften, wie beispielsweise die Konsistenz oder aber auch vollständige Transaktionsprotokolle mit wenig Aufwand integrieren beziehungsweise sogar individuell zwischen verschiedenen Eigenschaften und Protokollen wechseln zu können.

Anstatt für jeden Entwickler manuell eine speziell auf ihn zugeschnittene Transaktionsunterstützung oder für alle Entwickler die gleiche Transaktionsunterstützung zu implementieren und zur Verfügung zu stellen, bietet sich die Umsetzung und automatische Integration einer umfangreichen und individuell konfigurierbaren Transaktionsunterstützung auf Basis von Softwareproduktlinien an. Durch die Unterteilung in Merkmale, welche jede Transaktionsunterstützung gemeinsam hat und in variable Merkmale, welche je nach Anforderung hinzugefügt werden können, erhöht sich die Wiederverwendbarkeit der entwickelten Transaktionsunterstützung und die Anzahl der möglichen Zielsysteme [Clements und Northrop, 2001].

Zielstellung

Ziel dieser Arbeit ist eine Domänenanalyse des Zielsystems auf der Grundlage einer umfassenden Betrachtung von *Cloud Computing*, der Transaktionsverwaltung und dem Speichermanager, wobei *Amazon Simple Storage Service (S3)* den *Cloud*-Dienst darstellt, für den die konfigurierbare Transaktionsunterstützung entworfen werden soll. Im Zuge der Domänenanalyse erfolgt eine Analyse von Kernmerkmalen, die jede Transaktionsunterstützung gemeinsam hat und variablen Merkmalen, welche zusätzliche Funktionalitäten bereitstellen. Aufbauend auf der Analyse der benötigten Kern- und variablen Merkmale soll ein theoretisches Modell, zur Unterstützung der Entwicklung von *Cloud Computing*-Anwendungen durch das Bereitstellen von maßgeschneiderten und konfigurierbaren Transaktionseigenschaften und -protokollen, entwickelt werden. Dabei soll das Modell die Entwicklung einer modularen und konfigurierbaren Transaktionsunterstützung ermöglichen, welche den Bedürfnissen verschiedener Nutzergruppen gerecht wird und nicht mehr für jede Nutzergruppe separat entwickelt werden muss. Die Wiederverwendbarkeit und Konfigurierbarkeit nach einer einmaligen Umsetzung dieser Eigenschaften und Protokolle soll mit Hilfe der aspektorientierten Programmierung (engl. *Aspect-Oriented Programming* (AOP)) [Kiczales et al., 1997], der merkmalsorientierten Programmierung (engl. *Feature-Oriented Programming* (FOP)) [Czarnecki und Eisenecker, 2000] oder den *Aspectual Feature Modules* (AFM) [Apel et al., 2005a] gewährleistet werden.

Gliederung der Arbeit

Im Folgenden wird die Gliederung der Arbeit kurz erläutert.

Kapitel 2

In Kapitel 2 erfolgt eine Einführung in die Grundlagen, welche aus den vier Teilgebieten *Cloud Computing*, Transaktionsverwaltung und Speichermanager sowie der Erweiterung der objektorientierten Programmierung durch die aspektorientierte und merkmalsorientierte Programmierung beziehungsweise den *Aspectual Feature Modules* besteht.

Kapitel 3

Anschließend erfolgt in Kapitel 3 die Vorstellung des entwickelten Merkmalsdiagramms zur Unterstützung der Konfigurierbarkeit der Transaktionsunterstützung.

Kapitel 4

Aufbauend auf dem Merkmalsdiagramm aus Kapitel 3 wird in Kapitel 4 das Kollaborationsdiagramm zur Unterstützung einer möglichen Implementierung der benötigten Klassen und Erweiterungen beschrieben.

Kapitel 5

In Kapitel 5 werden die Ergebnisse sowie die gewonnenen Erkenntnisse und Probleme inklusive möglicher Lösungen vorgestellt.

Kapitel 6

Abschließend wird in Kapitel 6 ein Ausblick auf die zukünftigen, darauf aufbauenden Entwicklungen gegeben und die Arbeit zusammengefasst.

Kapitel 2

2 Grundlagen

In den folgenden Abschnitten erfolgt ein Einblick in die Grundlagen, die für das Verständnis dieser Arbeit notwendig sind. Dazu wird zunächst näher auf das Themengebiet *Cloud Computing* eingegangen. Da die modulare und konfigurierbare Transaktionsunterstützung für *Cloud*-basierte Datenspeichersysteme entworfen werden soll, wird der Fokus sowohl auf die Definition von *Cloud Computing* gelegt, als auch auf dessen Merkmale sowie dessen Unterteilung in Dienst- und Verwendungsmodelle. Anschließend erfolgt eine Übersicht der Vor- sowie Nachteile von *Cloud Computing* und der verschiedenen *Cloud*-Anbieter. Der Schwerpunkt wird dabei auf den *Cloud*-Dienst *Amazon Web Services* (AWS) *Secure Storage Service* (S3) gelegt, da auf Grundlage dieses *Cloud*-Speicherdienstes im Laufe der Arbeit eine Transaktionsunterstützung entwickelt werden soll. *Amazon S3* eignet sich vor allem deshalb als mögliches Zielsystem, weil es als NoSQL[1]-System nur eine eingeschränkte Transaktionsunterstützung aufweist [Gropengießer und Sattler, 2011] und somit eine geeignete Grundlage für mögliche Erweiterungen durch eine modulare und konfigurierbare Transaktionsunterstützung bietet.

Der anschließende Grundlagenbestandteil betrifft zum einen die Transaktionsverwaltung und zum anderen den Speichermanager. Diese Abschnitte bestehen aus Erläuterungen zu den ACID-Eigenschaften, ihre Notwendigkeit in *Cloud Computing*-Speicherdiensten und Techniken zur Gewährleistung einer fehlerfreien Synchronisation bei Mehrbenutzerzugriffen auf geteilte Speicherbereiche. Des Weiteren werden Pufferstrategien und Verfahren zur Datenwiederherstellung nach einem Fehlerfall näher erläutert. Den Abschluss des Grundlagenkapitels bilden die Techniken zur Erweiterung von Anwendungen mit Hilfe der aspektorientierten Programmierung, der merkmalsorientierten Programmierung und deren Kombination, den *Aspectual Feature Modules* (AFM), die auch für die Einbindung der Transaktionsverwaltung in Anwendungen für *Cloud*-Speicherdienste genutzt werden sollen.

[1] NoSQL ist ein Sammelbegriff für eine Anzahl von verschiedenen Datenbankkonzepten. Besonders an NoSQL-Datenbanken sind das nicht relationale Datenmodell, die Ausrichtung auf eine verteilte und horizontale Skalierbarkeit und das Fehlen der ACID-Eigenschaften, im Speziellen der fehlenden Konsistenz. Sie wurden vor allem dazu entwickelt, große Datenmengen zu verwalten [Edlich et al., 2011].

2.1 Cloud Computing

Als einer der zurzeit wichtigsten IT[2]-Trends ist *Cloud Computing* ein Paradigma, welches angibt, dass Anwendungen, Rechenressourcen und Daten nicht mehr über die eigene Infrastruktur bearbeitet, gespeichert oder bereitgestellt werden, sondern von einem Drittanbieter gekauft beziehungsweise gemietet werden können [Michelmann, 2010; Buxmann et al., 2011]. Bezüglich einer eindeutigen Definition von *Cloud Computing* herrscht wenig Einigkeit [Foster et al., 2008]. So haben sich im Laufe der Zeit viele verschiedene Definitionen entwickelt. Gonzalez et al. untersuchten zum Beispiel über 20 Definitionen aus diversen Quellen, um eine Standarddefinition zu finden, die *Cloud Computing* weitestgehend vollständig beschreibt [Gonzalez et al., 2009]. Der Hauptgrund für die unterschiedlichen Auffassungen von *Cloud Computing* besteht darin, dass mit *Cloud Computing* ein neues Betriebsmodell vorliegt, aber keine neue Technologie, da es eine Vielzahl bereits existierender Technologien zusammenbringt, um Aufgaben auf eine andere Art und Weise auszuführen [Zhang et al., 2010]. Das *National Institute for Standards and Technology* (NIST) bündelte verschiedene Ansätze und definierte 2009 *Cloud Computing* mit diesen Worten:

> *„Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models.“ [Mell und Grance, 2009]*

Cloud Computing ist folglich ein Modell, welches einen praktischen, bedarfsgerechten und netzbasierten Zugriff auf einen gemeinsamen Pool von konfigurierbaren Rechenressourcen (zum Beispiel Netzwerke, Speicher, Server, Dienste und Anwendungen) ermöglicht, die schnell und mit minimalem Verwaltungsaufwand und minimaler Interaktion des Anbieters bereitgestellt und wieder freigegeben werden können [Mell und Grance, 2009].

In den nächsten Abschnitten werden die vom *National Institute for Standards and Technology* erwähnten vier Verwendungsmodelle, die fünf wesentlichen Besonderheiten sowie die drei Dienstmodelle näher erläutert. Die Verwendungsmodelle (engl. *Deployment models*) beschreiben die Art der *Cloud* in Bezug auf die Öffnung der *Cloud* nach außen [Haselmann und Vossen, 2010b]. Die Dienstmodelle (engl. *Service models*) geben Aufschluss über die verschiedenen Dienste, die *Cloud*-Lieferanten anbieten. Diese unterscheiden sich darin, an welcher Stelle die Verantwortung beziehungsweise die Verwaltung durch den Nutzer aufhört und wo die des *Cloud*-Anbieters beginnt. Diese beiden Modelle sowie ihre Beziehung zu den charakterisierenden Eigenschaften (engl. *Service attributes*) [Sosinsky, 2011], die *Cloud Computing*-Anbieter gewährleisten [Haselmann und Vossen, 2010b], werden in Abbildung 2.1 aufgezeigt.

[2] Informationstechnik (IT)

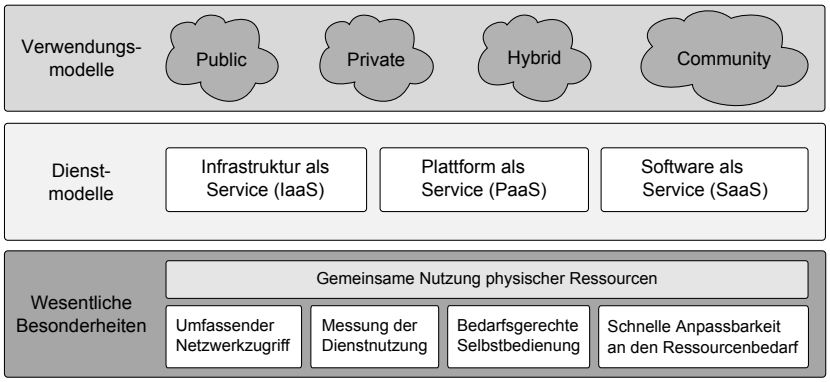

Abbildung 2.1: Die NIST *Cloud Computing*-Definition nach [Sosinsky, 2011]

Wesentliche Besonderheiten

Die wesentlichen Besonderheiten von *Cloud Computing* beinhalten, bezogen auf die vorangegangene Definition, die fünf Schlüsselmerkmale *On-demand self-service* (dt. bedarfsgerechte Selbstbedienung), *Broad network access* (dt. umfassender Netzwerkzugriff), *Resource pooling* (dt. gemeinsame Nutzung physischer Ressourcen), *Rapid elasticity* (dt. schnelle Anpassbarkeit an den Ressourcenbedarf) und *Measured service* (dt. Messung der Dienstnutzung) (siehe Abbildung 2.2).

- **Bedarfsgerechte Selbstbedienung**

  Die Eigenschaft *On-demand self-service* beschreibt die Möglichkeit des Kunden, zu jeder Zeit selbstständig und flexibel Ressourcen vom *Cloud*-Anbieter anzufordern, ohne dabei von einer menschlichen Interaktion auf Seiten dieses Anbieters abhängig zu sein [Baun et al., 2009].

- **Umfassender Netzwerkzugriff**

  Auf die Leistungen des Anbieters kann in Echtzeit über das Netzwerk mit Hilfe von Standardtechnologien zugegriffen werden. Solche Standardtechnologien sind beispielsweise das Internet, dedizierte Netzwerke[3] und *Virtual Private Networks* (VPN) (dt. virtuelle private Netzwerke). Dafür können durch den Nutzer Geräte wie Desktop-Computer, Mobiltelefone, Laptops oder PDAs[4] verwendet werden [Deussen et al., 2010].

- **Gemeinsame Nutzung physischer Ressourcen**

  Die Ressourcen (Hardware, Entwicklungsplattformen und Dienste), die vom Anbieter bereitgestellt werden, sind in Pools zusammengefasst. Die Zuordnung der Ressourcen zu verschiedenen Nutzern erfolgt parallel, automatisiert und wird dynamisch an den Bedarf des Kunden angepasst, um eine optimierte Ressourcennutzung zu erreichen [Gonzalez et al., 2009].

[3] Mit dem Begriff dediziertes Netzwerk werden Schnittstellen bezeichnet, die eine dauerhafte Verbindung zum Telefonnetz besitzen [Boswell, 2003].

[4] Ein PDA (dt. persönlicher digitaler Assistent) ist ein sehr leichter, vom Stromnetz unabhängig arbeitender Kleinstrechner, der die Aufgaben eines *Personal Information Management Systems* (PIMS) (dt. persönliches Informationsverwaltungssystem) unterstützen kann. Dazu gehören zum Beispiel das Verwalten und Organisieren von Terminen, Kontakten und Notizen [Back, 2001].

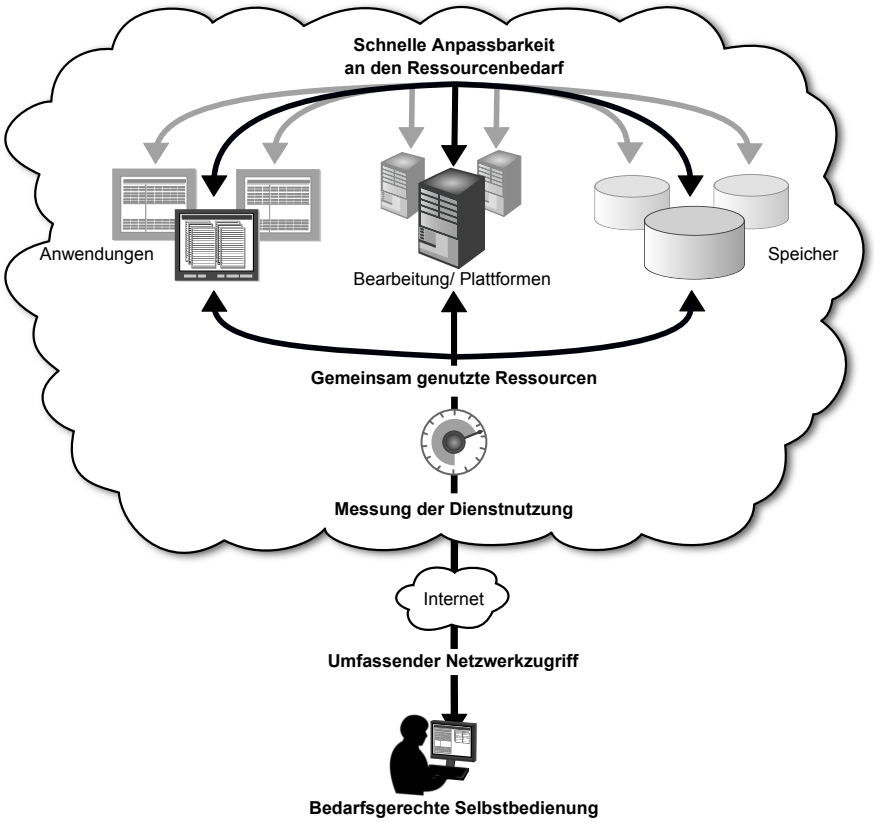

Abbildung 2.2: Wesentliche Besonderheiten von *Cloud Computing* nach [Wilshusen, 2010]

- **Schnelle Anpassbarkeit an den Ressourcenbedarf**

  Die Verfügbarkeit der Ressourcen erfolgt elastisch. Benötigt der Nutzer oder eine Anwendung zusätzliche Ressourcen, werden diese innerhalb einer angemessenen Zeitspanne und ohne großen Aufwand automatisch in ausreichender Menge zugeteilt. So erscheint für den Nutzer die Illusion unbeschränkter Verfügbarkeit der Ressourcen [Deussen et al., 2010]. Vorteilhaft ist dies beispielsweise für Entwickler mit innovativen Ideen für neue Internet-Anwendungen, da es nicht länger notwendig wird, vor der eigentlichen Umsetzung der Idee Zeit und Geld in Hardware zu investieren. Stattdessen können diese Ressourcen flexibel von *Cloud*-Anbietern bezogen und an den steigenden Bedarf angepasst werden [Baun et al., 2009].

- **Messung der Dienstnutzung**

  Cloud-Systeme sind mit integrierten Monitoring- und Messfunktionen, die Dienstprogrammen[5] ähnlich sind [Chen et al., 2010], ausgestattet, welche die Ressourcennutzung automatisch kontrollieren und optimieren sowie die erreichte Dienstqualität und -quantität auf Seiten der Nutzer bewerten [Deussen et al., 2010]. Dabei fließen verschiedene Variablen wie zum Beispiel die aktiven Nutzeraccounts oder die Bandbreite mit ein [Mell und Grance, 2009].

[5] Mit der Hilfe von Dienstprogrammen werden systemorientierte, häufig wiederkehrende, anwendungsneutrale Aufgaben ausgeführt [König, 2003].

Dienstmodelle

Abbildung 2.3 zeigt die drei gängigen Grundkategorien für *Cloud Computing*-Angebote: *Infrastructure as a Service* (IaaS) (dt. Infrastruktur als Service), *Platform as a Service* (PaaS) (dt. Plattform als Service) und *Software as a Service* (SaaS) (dt. Software als Service) [Haselmann und Vossen, 2010a].

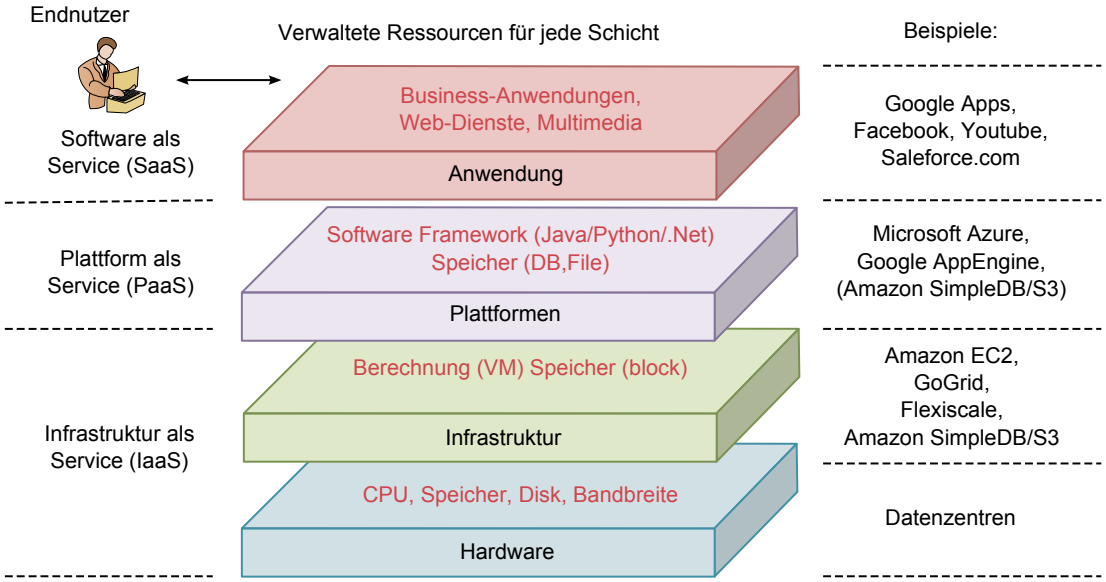

Abbildung 2.3: Dienstmodell-Architekturen von *Cloud Computing* nach [Zhang et al., 2010]

- **Infrastruktur als Service**

  Infrastructure as a Service ist die unterste Schicht in der *Cloud Computing*-Unterteilung und besteht aus der Hardware- und der Infrastruktur-Schicht. Verantwortlich für die Bereitstellung der Hardware-Schicht ist das Datenzentrum, welches meist aus mehreren Tausend verbundenen Servern besteht. Sämtliche technischen Ressourcen inklusive Kühl- und Stromversorgungsysteme sowie Router und Server werden hier verwaltet [Zhang et al., 2010]. Die Infrastruktur-Schicht hingegen liefert dem Nutzer Zugriff auf alle virtuellen Maschinen, die der Anbieter zur Verfügung stellt. Jedoch ist der Nutzer dafür verantwortlich, die von ihm dort eingebrachten Recheninstanzen selbst zu verwalten [Santos et al., 2009]. Beispiele für *Infrastructure as a Service*-Anbieter sind *AWS Elastic Compute Cloud* (EC2) und *AWS* S3 [Bhardwaj et al., 2010].

- **Plattform als Service**

  Platform as a Service bietet eine Anwendungs- oder Entwicklungsumgebung, in welcher der Nutzer seine eigene Applikation erstellen und anschließend in die *Cloud* laden kann [Buyya et al., 2009]. Um die Entwicklung zu beschleunigen und die Skalierbarkeit zu unterstützen, die solche *Cloud*-Anwendungen benötigen, sowie die Interaktion zwischen der Programmiersprachenumgebung und der Applikation zu erleichtern, liefern *Cloud*-Anbieter dem Entwickler eine Programmierumgebung, welche aus einer Menge von wohldefinierten Programmierschnittstellen (engl. *Application Programming Interfaces* (API)) besteht [Youseff et al.,

2008]. Die *Google App Engine* ist ein Beispiel für eine *Web-Platform as a Service*, die es möglich macht, Web-Applikationen zu erstellen, welche auf Java oder Python basieren [Jensen et al., 2009].

- **Software als Service**

  Software as a Service stellt das höchste Level der *Cloud Computing*-Architektur dar und liefert dem Kunden die Möglichkeit Anwendungen zu nutzen, welche auf der *Cloud*-Infrastruktur bereits ausgeführt werden [Lee und Hughes, 2010]. Der Nutzer muss infolgedessen nicht selbst eine Applikation schreiben und braucht auch nicht die darunter liegende Infrastruktur, wie beispielsweise das Netzwerk, die Server, den Speicher oder die ausführenden Systeme, verwalten oder kontrollieren [Kundra, 2010]. Die Vielzahl von *Software as a Service*-Anwendungen beinhalten zum Beispiel Programme wie *Microsoft Windows Live*[6], *Google Docs*[7] und *Zoho Office Suite*[8] [Kim, 2009].

Verwendungsmodelle

Zusätzlich zu den Dienstmodellen wird die *Cloud* in Bezug auf die vier Verwendungsmodelle *Private Cloud* (dt. private Cloud), *Public Cloud* (dt. öffentliche Cloud), *Hybrid Cloud* (dt. gemischte Cloud) und *Community Cloud* (dt. gemeinschaftliche Cloud) unterteilt [Rodriguez-Martinez et al., 2010] (siehe Abbildung 2.4).

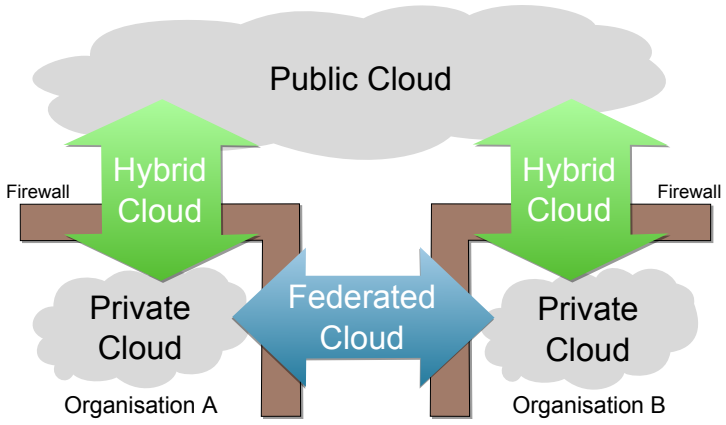

Abbildung 2.4: *Cloud*-Verwendungsmodelle *Private Cloud*, *Public Cloud*, *Hybrid Cloud* und *Community/ Federated Cloud* nach [Lee, 2010][9]

[6] *Microsoft Windows Live* ist eine Sammlung von Onlinediensten, die über einen Webbrowser zu erreichen sind beziehungsweise von Programmen, die aus dem Internet nachgeladen werden müssen. Dazu gehören zum Beispiel *Windows Live Mail* oder *Windows Live Messanger* [Seimert, 2010].

[7] *Google Docs* (*Google Docs & Spreadsheets*) ist ein webbasiertes Text- und Tabellenkalkulationsprogramm zur Erstellung und Bearbeitung von Dokumenten, wobei die Dokumente gemeinsam mit anderen Nutzern bearbeitet werden können [Merkt et al., 2007].

[8] *Zoho Office Suite* ist ein webbasiertes Anwendungspaket für die Erstellung und Bearbeitung von beispielsweise Textdokumenten, Präsentationen und Datenbanken [Malita und Boffo, 2010].

[9] „Ein Sicherheitsgateway (oft auch Firewall genannt) ist ein System aus soft- und hardwaretechnischen Komponenten, das die sichere Kopplung von IP-Netzen durch Einschränkung der Kommunikation gemäß den IT-Sicherheitsrichtlinien gewährleistet. Dies bedeutet, dass ausschließlich erwünschte Zugriffe oder Datenströme zwischen verschiedenen Netzen zugelassen werden. Zudem können mit Sicherheitsgateways je nach Ausbaustufe die übertragenen Daten kontrolliert und dementsprechend problematische Inhalte ausgefiltert werden.“ [Bun, 2005]

- **Public Cloud**

  In einer *Public Cloud* besitzt und verwaltet eine größere Organisation, die *Cloud-Dienste* anbietet, die *Cloud*-Infrastruktur. Zu diesen Organisationen gehören beispielsweise *Microsoft*, *Google* oder *Amazon* [Sriram und Khajeh-Hosseini, 2010] (siehe Abbildung 2.5).

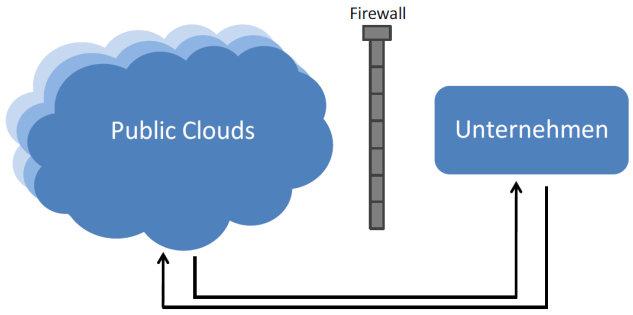

Abbildung 2.5: *Public Cloud* [Adelsberger und Drechsler, 2010]

Microsoft Azure Storage Service und *Amazon S3* sind Beispiele für Speicherdienste, die auf *Public Clouds* basieren. So kann der Kunde seine Daten in die *Cloud* laden und benötigt keine eigene private Speicher-Infrastruktur. Dies bedeutet aber auch, dass sich die Kundendaten außerhalb der Reichweite und Kontrolle des Kunden befinden und dies folglich gewisse Sicherheits- und Datenschutzgefahren mit sich bringt [Kamara und Lauter, 2010].

- **Private Cloud**

  Die *Private Cloud* ist die sicherste aller *Cloud*-Verwendungsmodelle, da sich bei diesem Modell sowohl Anbieter als auch Kunde in derselben Organisation befinden und infolgedessen der Aufbau als auch die Verwaltung unter vollständiger Kontrolle des Unternehmens steht [Wald, 2010] (siehe Abbildung 2.6). Dadurch, dass die *Cloud*-Umgebung innerhalb einer Firewall erstellt wird, bietet sie dem Nutzer mehr Schutz vor Sicherheitsrisiken im Internet [Yan et al., 2009].

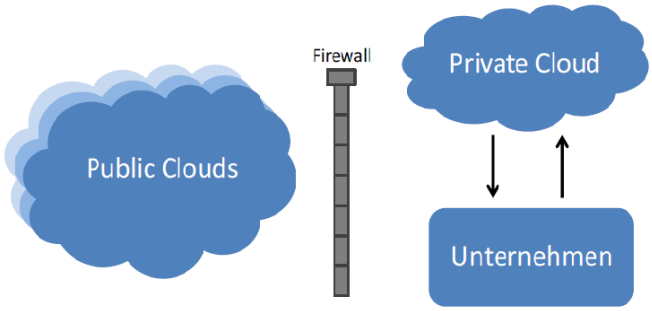

Abbildung 2.6: *Private Cloud* [Adelsberger und Drechsler, 2010]

Dies bedeutet jedoch auch, dass die Unterhaltskosten der *Private Cloud* vom Unternehmen selbst getragen werden müssen. Das schließt sowohl den Aufbau und den Betrieb als auch das notwendige Fachpersonal mit ein. Von Vorteil für die

Unternehmen ist aber, dass *Private Clouds* im Gegensatz zu *Public Clouds* direkt auf die Geschäftsprozesse des Unternehmens zugeschnitten werden können [Adelsberger und Drechsler, 2010].

- **Hybrid Cloud**

  Die *Hybrid Cloud* stellt eine Mischform aus *Private* und *Public Cloud* dar [Awessou, 2009] (siehe Abbildung 2.7). Dieser Zusammenschluss mehrerer *Clouds* ermöglicht zum Beispiel den Austausch von Daten und Programmen zwischen den *Clouds* [Haselmann und Vossen, 2010b]. Der Hauptgrund für die Entwicklung dieses Modells ist die Möglichkeit für Firmen, wichtige Daten im eigenen internen Netzwerk zu lagern und die übrige Funktionalität beziehungsweise nicht-sensible Daten (zum Beispiel höchst skalierbare Webseiten oder Massendatenverarbeitungen auf anonymen Logdateien [Haselmann und Vossen, 2010b]) in die *Public Cloud* auszulagern [Annapureddy, 2010].

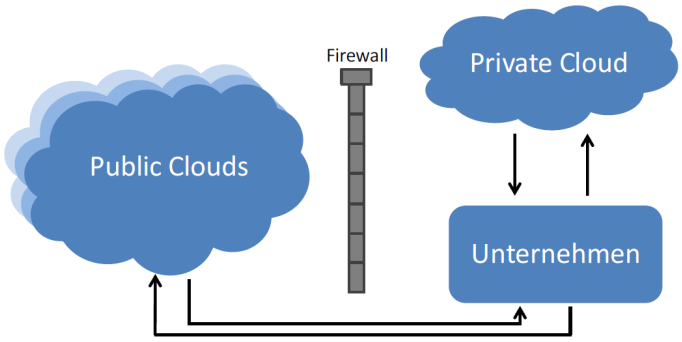

Abbildung 2.7: *Hybrid Cloud* [Adelsberger und Drechsler, 2010]

- **Federated Cloud oder Community Cloud**

  Neben der *Hybrid Cloud* stellt auch die *Community Cloud* eine Sonderform dar [Birk und Wegener, 2010]. In diesem Fall teilen sich mehrere Organisationen mit ähnlichen Anforderungen und Interessen die *Cloud*-Infrastruktur für einen bestimmten Zweck, bewahren dabei aber ein gewisses Level an Sicherheit und Datenschutz [Jimenez, 2010; Neto, 2011]. Die Verwaltung der *Cloud* übernehmen entweder die beteiligten Organisationen selbst oder geben diese an externe Dienstleister ab [Haselmann und Vossen, 2010b].

Vor- und Nachteile von Cloud Computing

Cloud Computing bringt sowohl Vorteile als auch Nachteile mit sich. Diese fallen für die unterschiedlichen Nutzer von *Cloud Computing* verschieden stark ins Gewicht [Mirzaei, 2009].

- **Vorteile von Cloud Computing**

  Zu den Vorteilen von *Cloud Computing* gehören die Verfügbarkeit und Elastizität von Rechen- und Speicherkapazitäten sowie ein günstiges Kostenmodell. Auch eine erhöhte Datenausfallsicherheit, eine Garantie auf sofortige Software-Updates, eine Kompatibilitätsverbesserung zwischen Dokumenten und ein mobiler Zugriff auf die eigenen Daten werden durch *Cloud Computing* ermöglicht.

1. **Elastizität**

   Ein Vorteil von *Cloud Computing* ist dessen Elastizität in Bezug auf sich ändernde Bedingungen. Im Falle von stark ansteigenden Anfragen an ein und dasselbe Produkt oder exponentiell wachsende soziale Netzwerke, egal ob zeitlich begrenzt oder auf Dauer, können zusätzliche Ressourcen in wenigen Minuten dazu geschaltet und wieder freigegeben werden [Abadi, 2009]. Dafür ist es nicht notwendig, auf eine größere leistungsstärkere Maschine umzusteigen [Aymerich et al., 2008]. Stattdessen werden diese Ressourcen beschafft, indem zusätzliche Server-Instanzen für diese Aufgabe angefragt werden, denn *Cloud Computing* ist dazu fähig, eine variable Anzahl von physischen und virtuellen Maschinen (VM) nutzen zu können [Cavilla et al., 2009]. *Amazons EC2* teilt beispielsweise Ressourcen in kleine, große und besonders große virtuelle Server-Instanzen auf. Die größte beinhaltet dabei nicht mehr als vier Prozessorkerne [Abadi, 2009].

2. **Geringere Hardware-Kosten**

   Der Endnutzer profitiert beim *Cloud Computing* davon, selbst keinen leistungsstarken Computer kaufen zu müssen. Er benötigt weder einen guten Prozessor noch eine große Festplatte und nicht einmal ein CD- oder DVD-Laufwerk, wie es für Software, die auf dem eigenen Desktop-Computer läuft, häufig erforderlich ist. Da *Cloud Computing* Anwendungen webbasiert laufen, muss kaum Software heruntergeladen oder installiert werden. Auch das Speichern von Dokumenten auf dem eigenen Computer ist nicht notwendig, da diese in der *Cloud* gesichert werden [Miller, 2008]. Ist der Kunde eine ganze Firma, entfällt für diese, wenn sie *Cloud Computing* nutzt, die Anschaffung einer eigenen umfangreichen Hardware- und Softwareinfrastruktur, was sich sowohl in den Kauf- als auch in den Instandhaltungskosten widerspiegelt. Auch Personal, welches sich um die eigenen Systeme kümmern müsste, muss nicht extra eingestellt werden beziehungsweise nicht so viele Personen umfassen [Leavitt, 2009].

3. **Reduzierte Software-Kosten**

   Anstatt teure Software zu kaufen, kann der Kunde die meisten *Cloud*-Anwendungen kostenlos bekommen. So sind unter Umständen Applikationen wie *Google Docs Suite* beispielsweise vollkommen kostenfrei in der *Cloud* verwendbar. Desktop-Anwendungen mit ähnlichen Funktionen wie zum Beispiel *Microsoft Office*[10] kosten hingegen 100 Euro und mehr. [Miller, 2008; Maggiani, 2009].

4. **Pay as you go**

   Die Kosten für den *Cloud*-Konsumenten können nahe Null sein, wenn Ressourcen nicht verwendet werden, da die Nutzung der *Cloud* an den Kunden angepasst bezahlt wird und nicht bezogen auf Durchschnitts- oder Spitzenwerte [Napper und Bientinesi, 2009; Mirzaei, 2009]. Dafür wählt der Kunde ein System, welches seinen Bedürfnissen gerecht wird und bezahlt nur für

[10] *Microsoft Office* ist eine Sammlung von verschiedenen *Microsoft*-Programmen. Zu diesen gehören zum Bespiel *Microsoft Word* und *Microsoft PowerPoint* zur Erstellung und Bearbeitung von Textdokumenten und Präsentationen [Born, 2010].

die Zeit, in der er eine Instanz dieses Systems in Anspruch nimmt. Administrative Funktionen wie *Backups* (dt. Datensicherungen) und *Recovery* (dt. Datenwiederherstellung) sind in den Kosten bereits enthalten [Schadt et al., 2010].

5. **Verbesserte Performanz**

   Durch die Nutzung von Anwendungen, die sich in einer *Cloud* befinden, müssen weniger Programme und Prozesse in den Speicher des eigenen Computers geladen werden. Demgemäß erhöht sich dessen Leistungsfähigkeit für andere Aufgaben [Miller, 2008; Maggiani, 2009]. Des Weiteren ist es möglich, die Rechenleistung der *Cloud* zu nutzen, um performanzlastige Tätigkeiten wie das Durchsuchen von mehreren Gigabyte E-Mails ausführen zu lassen [Aymerich et al., 2008].

6. **Sofortige Software-Updates**

   Der Nutzer einer *Cloud*-Anwendung erhält immer automatisch die neueste Version. So muss er sich nicht mehr entscheiden, ob er eine veraltete Software behält oder Geld in ein Upgrade dieser Software investiert [Aymerich et al., 2008; Miller, 2008; Cosoi und Vossen, 2010].

7. **Verbesserte Kompatibilität von Dokumentformaten**

   Es besteht beim Erstellen von Dokumenten mit Hilfe einer *Cloud*-Anwendung kein Problem mehr in Bezug auf die Verträglichkeit verschiedener Dokumentformate. Anders als bei *Microsoft Word* 2007 und 2003[11] ist es für jeden *Cloud*-Kunden möglich, Dokumente zu öffnen, die ihm von einem anderen Kunden bereitgestellt werden und welche in derselben *Cloud*-Applikation erstellt wurden [Aymerich et al., 2008; Miller, 2008], was auf den zuvor erläuterten Punkt *Sofortige Software-Updates* zurückzuführen ist.

8. **Uneingeschränkte Speicherkapazität**

   Cloud Computing bietet virtuell unbegrenzten Speicher. Ist das private Computersystem auf wenige Terabyte (TB)[12] große Festplatten beschränkt, eröffnet die *Cloud* Zugriff auf mehrere hundert Petabyte (PB)[13] Speicherplatz [Aymerich et al., 2008; Miller, 2008; Mirzaei, 2009].

9. **Erhöhte Datenausfallsicherheit**

   Da die Daten in der *Cloud* gesichert werden, besteht im Falle eines Absturzes des eigenen Computers nicht mehr die Gefahr, dass Daten verloren gehen [Miller, 2008; Pocatilu et al., 2010].

10. **Größere Mobilität und Geräteunabhängigkeit**

    Durch die Inanspruchnahme der *Cloud* ist der Zugriff auf erstellte Dokumente beispielsweise nicht mehr auf einen einzelnen Computer beschränkt. Möglich wird dies dadurch, dass auf die *Cloud* zugegriffen werden kann, solange ein Computer und eine Internetverbindung zur Verfügung stehen [Aymerich et al., 2008; Mirzaei, 2009]. Für diesen Zugriff ist dabei jedes internetfähige

[11] Das Öffnen von Dateien aus *Microsoft Office* 2007 Programmen ist inzwischen durch die Nachinstallation eines *Compatibility Packs* für *Microsoft Office* 2000 und 2003 Programme möglich [Microsoft, 2011].

[12] 1 Terabyte = 1000 Gigabyte (GB)

[13] 1 Petabyte = 1000 Terabyte

Gerät (Desktop-Computer, Laptops, Handys, PDAs) geeignet. Es ist dabei nicht notwendig eine spezielle Version eines Programms für ein bestimmtes Gerät zu kaufen, noch müssen Dokumente in einem gerätespezifischen Format gespeichert werden [Pocatilu et al., 2010]. Auch die Version des erstellten Dokumentes bleibt auf dem aktuellen Stand, da die *Cloud* immer die letzte Version der Dokumente behält [Miller, 2008].

- **Nachteile von Cloud Computing**

  Nachteile von *Cloud Computing* sind die Notwendigkeit einer ausreichend guten Internetverbindung, die eingeschränkte Funktionalität der Anwendungen, die Gefahr vor Diebstahl oder Verlust der eigenen Daten sowie die Möglichkeit von Missbrauch der *Cloud*-Dienste.

  1. **Notwendigkeit einer ständigen Internetverbindung**

     Cloud Computing ist unmöglich, solange keine Verbindung mit dem Internet besteht. Nicht einmal der Zugriff auf seine eigenen Dokumente und Anwendungen ist ohne eine Internetverbindung möglich [Miller, 2008; Mirzaei, 2009; Cosoi und Vossen, 2010].

  2. **Eingeschränkte Nutzung bei langsamen Verbindungen**

     Die Internetverbindungsgeschwindigkeit beeinflusst die Arbeitsperformanz mit der *Cloud*. Webbasierte Anwendungen benötigen eine gewisse Bandbreite für den Download. So ist selbst das Wechseln von einer Seite zur nächsten im Dokument schwierig, wenn beispielsweise langsame Verbindungen genutzt werden [Pocatilu et al., 2010]. Doch auch beim Vorhandensein einer schnellen Verbindung können webbasierte Anwendungen gelegentlich langsamer sein als ähnliche auf dem eigenen Computer installierte Produkte. Das liegt daran, dass jeder Teil der Anwendung, wie zum Beispiel die Schnittstelle (engl. *Interface*) und das Dokument, welches gerade bearbeitet wird, häufig zur *Cloud* und zurück geschickt werden [Miller, 2008].

  3. **Eingeschränkte Features**

     Gegenwärtig sind noch viele *Cloud*-Anwendungen weniger umfangreich in ihrer Funktionalität als vergleichbare Desktop-Anwendungen. So bietet zum Beispiel *Microsoft PowerPoint*[14] mehr Features als *Google Presentations*[15] [Miller, 2008; Hewitt, 2011].

  4. **Datenschutz**

     Auch wenn es keinen Sinn fürs Geschäft macht, die Privatsphäre von Kunden anzugreifen und auf ihre Daten ohne Erlaubnis zuzugreifen, erhöht es die Anzahl der potenziellen Sicherheitsrisiken, die Daten aus der eigenen Kontrolle zu geben [Abadi, 2009]. Des Weiteren ist es auch möglich, dass Systemfehler ungewollte Lücken verursachen, so dass Informationen von einem Nutzer zu einem anderen weitergegeben werden. Auf diese Art und Weise hat eine Schwachstelle der *Google Docs* Anwendung dafür gesorgt, dass einige Nutzer unbeabsichtigt ihre Dokumente mit anderen teilten [Mowbray, 2009].

[14] *Microsoft PowerPoint* ist ein Desktop-Programm von *Microsoft* zur Erstellung und Bearbeitung von Präsentationen [Born, 2010].

[15] *Google Presentations* ist eine webbasierte Anwendung zur Erstellung und zum Bearbeiten von Präsentationen [Hewitt, 2011].

5. **Verlust von Daten**

   Die Daten, die in der *Cloud* gespeichert sind, werden auf mehrere Datenträger kopiert, was die Wahrscheinlichkeit für den Verlust der Daten minimiert. Abhängig vom gewählten *Cloud*-Anbieter und gemieteten Dienstmodell erfolgt die Vervielfältigung der Daten unterschiedlich häufig und verteilt. Große *Cloud*-Anbieter, die Datenzentren auf der ganzen Welt besitzen, haben die Möglichkeit, eine hohe Fehlertoleranz anzubieten sowie die Sicherung der Daten über eine große geographische Distanz zu verteilen [Abadi, 2009]. In dem Fall, dass Daten trotzdem verloren gehen, hat der Nutzer selbst aber kein eigenes lokales *Backup*, solange er die *Cloud*-Dokumente nicht regelmäßig auf das eigene System lädt [Miller, 2008].

6. **Missbrauch der Cloud-Dienste**

   Cloud Computing bietet Kriminellen die Möglichkeit in relativer Anonymität zu agieren. Dabei nutzen sie die *Cloud*, um die Reichweite und Effektivität ihrer Aktivitäten zu erhöhen und ihre Entdeckung zu verhindern. Solche Aktivitäten beinhalten zum Beispiel das Knacken von Passwörtern und Schlüsseln, die Einrichtung von dynamischen Angriffspunkten sowie das Bereitstellen von schädlichen Daten [Zwienenberg und Willems, 2010].

Cloud-Dienste

Die Tabellen 2.1 und 2.2 zeigen eine Übersicht einer Auswahl an *Cloud*-Diensten, welche von den drei größten *Cloud*-Anbietern (*Amazon*, *Google* und *Microsoft*) bereitgestellt werden. Da noch kein Standard für *Cloud*-Dienste existiert, unterscheiden sich diese Angebote in einigen Punkten [Kossmann et al., 2010], so dass jeder Nutzer der *Cloud* entscheiden muss, welcher Dienst seinen Bedürfnissen am ehesten gerecht wird. Für diese Arbeit ist insbesondere der Datenspeicherdienst *Amazon Simple Storage Service* (*S3*) von Interesse, da mit *S3* ein so genanntes NoSQL-System mit eingeschränkter Transaktionsunterstützung (zum Beispiel eventuelle[16] statt starke Konsistenz), aber besserer Skalierung besteht [Gropengießer und Sattler, 2011].

	AWS RDS	**AWS SimpleDB**
Business Model	PaaS	PaaS
Cloud Provider	Amazon	Amazon
Web/app Server	Tomcat	Tomcat
Database	MySQL	SimpleDB
Storage/File Sys.	-	-
Consistency	Repeatable Read	Ev. Consistency
App-Language	Java	Java
DB-Language	SQL	SimpleDB Queries
Architecture	Classic	Part.+Repl.
HW Config.	manual	manual/automatic

Tabelle 2.1: Übersicht: *Cloud*-Dienste nach [Kossmann et al., 2010] Teil 1

[16] *Eventual consistency* ist eine spezielle Form der schwachen Konsistenz und gibt an, dass zum Beispiel in einem Speichersystem alle Anfragen letztendlich auf den zuletzt aktualisierten Wert zurückgreifen, wenn keine neuen Updates auf das Objekt durchgeführt werden [Afek et al., 2010].

	Google AppEng	**MS Azure**
Business Model	PaaS	PaaS
Cloud Provider	Google	Microsoft
Web/app Server	AppEngine	.Net Azure
Database	DataStore	SQL Azure
Storage/File Sys.	GFS	Windows Azure
Consistency	Snapshot Isolation	Snapshot Isolation
App-Language	Java/AppEngine	C#
DB-Language	GQL	SQL
Architecture	Part.+Repl.(+C)	Replication
HW Config.	automatic	manual/automatic

	AWS MySQL	**AWS MySQL/R**	**AWS S3**
Business Model	IaaS	IaaS	IaaS
Cloud Provider	Flexible	Flexible	Flexible
Web/app Server	Tomcat	Tomcat	Tomcat
Database	MySQL	MySQL Rep	none
Storage/File Sys.	EBS	EC2&EBS	S3
Consistency	Repeatable Read	Repeatable Read	Ev. Consistency
App-Language	Java	Java	Java
DB-Language	SQL	SQL	low-level API
Architecture	Classic	Replication	Distr.Control
HW Config.	manual	manual	manual

Tabelle 2.2: Übersicht: *Cloud*-Dienste nach [Kossmann et al., 2010] Teil 2

- **Amazon (AWS)**

  Amazon ist ein *Infrastructure as a Service*-Anbieter für die flexible Nutzung von Rechenleistung, Speicherkapazität und für den Zugriff auf eine Reihe elastischer IT-Infrastruktur-Dienstleistungen, wobei *Amazon Web Services* alle *Cloud*-Angebote von *Amazon* umfasst [Amazon.com, 2011b]. Zu diesen Angeboten[17] gehören nach [Amazon.com, 2011b] beispielsweise:

  - *Datenverarbeitung*
    - *Amazon Elastic Compute Cloud* (EC2)
  - Bereitstellung von Inhalten
    - *Amazon CloudFront*
  - Datenbanken
    - *Amazon SimpleDB*
    - *Amazon Relational Database Service* (RDS)
  - *Messaging*
    - *Amazon Simple Queue Service* (SQS)

[17] Vollständige Liste unter http://aws.amazon.com/de/products/

- Netzwerk
  - *Amazon Virtual Private Cloud* (VPC)
- Speicherung
  - *Amazon Simple Storage Service (S3)*
  - *Amazon Elastic Block Storage* (EBS)

Amazon Simple Storage Service (S3) *Amazon S3* ist ein Datenspeicher, der über das Internet erreichbar ist. Dabei können zu jeder Zeit beliebige Datenmengen über eine *Webservice*-Schnittstelle gespeichert und abgerufen werden. Jedes gespeicherte Objekt befindet sich in einem *Bucket* und kann über einen individuellen Schlüssel abgerufen und mit Authentifizierungsmechanismen vor unberechtigtem Zugriff geschützt werden. Auch wenn die Anzahl der speicherbaren Objekte unbegrenzt ist, besteht eine Einschränkung bezüglich der Dateigröße. So dürfen nur Dateien mit einer Größe von einem Byte bis fünf Terabyte geschrieben, gelesen und gelöscht werden [Amazon.com, 2011a].

Abbildung 2.8 zeigt einen typischen Entwickler-*Workflow* in einer *Amazon S3*-Umgebung [Redkar, 2009].

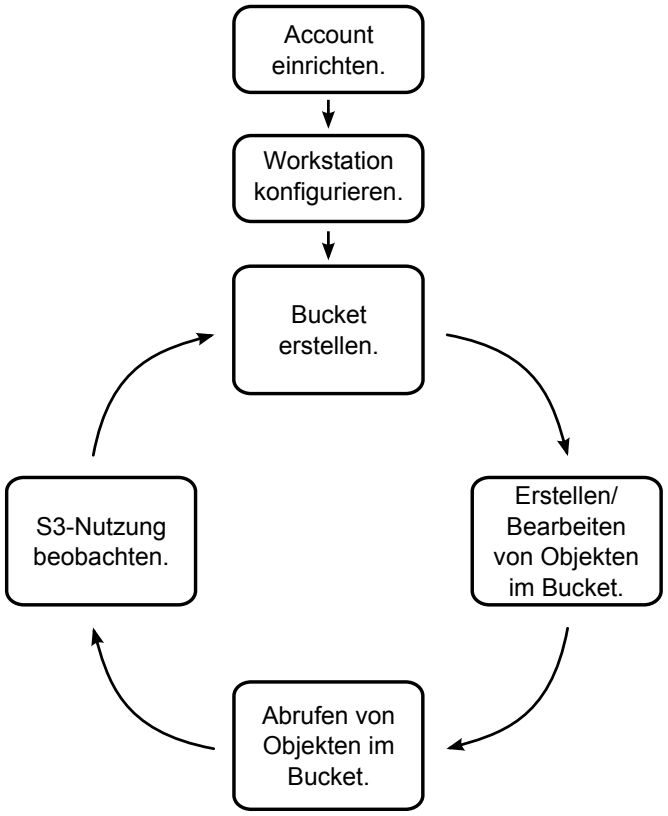

Abbildung 2.8: *Amazon S3* Entwickler-*Workflow* nach [Redkar, 2009]

2.2 Transaktionsverwaltung

Um in einem verteilten System Konsistenzverletzungen auf dem Datenspeicher ausschließen zu können, ist eine fehlerfrei arbeitende Transaktionsverwaltung erforderlich, die nebenläufig ablaufende Transaktionen[18] verarbeitet und synchronisiert [Kemper und Eickler, 2006]. Zu diesem Zweck werden im Folgenden die ACID-Eigenschaften und ihre Notwendigkeit in verteilten Systemen auf der *Cloud* sowie Verfahren zur Nebenläufigkeitskontrolle (engl. *Concurrency Control*) näher erläutert.

ACID-Eigenschaften

Das ACID-Prinzip einer Transaktion besteht aus den vier Eigenschaften *Atomicity* (dt. Atomarität), *Consistency* (dt. Konsistenz), *Isolation* (dt. Isolation) und *Durability* (dt. Dauerhaftigkeit). Diese Eigenschaften müssen bei einer Transaktion eingehalten werden und garantieren das Überführen einer Datenbank von einem konsistenten in einen neuen, konsistenten Zustand beim Abarbeiten der Operationen einer Transaktion [Saake et al., 2005; Meier, 2010].

- **Atomarität**

  Die Atomarität bedeutet, dass eine Transaktion eine atomare Einheit ist und dementsprechend entweder vollständig oder gar nicht erfolgt [Camp et al., 1995; Ozsu und Valduriez, 1999; Schatten et al., 2010]. Die zur Transaktion gehörende Operationsfolge darf nur komplett ausgeführt werden. Ist dies nicht möglich, so müssen alle bereits durchgeführten Änderungen wieder zurückgesetzt werden [Dadam, 1996].

- **Konsistenz**

  Durch die Konsistenzerhaltung wird gewährleistet, dass nach dem Ausführen einer Transaktion ein ebenso konsistenter Zustand des Systems vorliegt wie vor dem Start der Transaktion [Vossen, 2000; Balling, 2004]. Dabei ist es jedoch nicht notwendig, dass während der Ausführung der Transaktion ein konsistenter Zustand herrscht [Großmann und Koschek, 2005].

- **Isolation**

  Die Eigenschaft Isolation beschreibt die Forderung nach einem isolierten Ablauf der Transaktion. Das heißt, dass erst nach einem erfolgreichen Beenden einer Transaktion ihre Auswirkungen für andere Transaktionen sichtbar werden [Balling, 2004]. Dies muss auch dann der Fall sein, wenn mehrere andere Transaktionen zur gleichen Zeit ausgeführt werden und auf denselben Datenbeständen aktiv gearbeitet haben. Auch wenn sich parallel ausgeführte Transaktionen überlappen, soll der Anschein einer seriellen Verarbeitung gewahrt werden [Beyerle et al., 2005; Saake et al., 2005].

- **Dauerhaftigkeit**

  Die Dauerhaftigkeit fordert, dass das Ergebnis einer Transaktion nach ihrem Beenden persistent gespeichert wird [Masak, 2006; Melzer, 2010], so dass auch nach

[18] Unter einer Transaktion wird eine Folge von Datenbankoperationen verstanden, die insbesondere bezüglich der Integritätsüberwachung als Einheit (atomar) angesehen wird [Heuer et al., 2008].

einem Systemzusammenbruch keine vorgenommenen Änderungen verloren gehen können [Niemann, 2001; Weikum und Vossen, 2002].

Einschränkungen der ACID-Anforderungen

In Bezug auf Speicherlösungen auf der *Cloud* entscheidet zum Beispiel die Gewährleistung der Konsistenzeigenschaft zwischen hohen und niedrigen Transaktionskosten beziehungsweise der Verfügbarkeit der aktuellen Daten. Von daher ist zu entscheiden, welche ACID-Anforderungen für eine Anwendung bestehen. *Web Shops*, Kreditkarten- und Kontobilanzen benötigen beispielsweise eine hohe Konsistenzgarantie, wohingegen das Anzeigen von Nutzereigenschaften, wie weitere gekaufte Produkte, auch mit geringer Konsistenzabwicklung unproblematisch ist [Kraska et al., 2009]. Einen weiteren Grund für den eventuellen Verzicht auf strikte Konsistenzeigenschaften liefert auch das CAP[19]-Theorem [Gropengießer und Sattler, 2011].

- **CAP-Theorem**

  Das CAP-Theorem (auch Brewers Theorem) besagt, dass für große verteilte Systeme – wie es auch bei verteilten Anwendungen auf der *Cloud* der Fall ist – nur zwei der drei benötigten Eigenschaften Konsistenz, Verfügbarkeit und Partitionstoleranz unterstützt werden können und folglich, wie in Abbildung 2.9 zu sehen ist, keine überlappenden Bereiche bestehen, für die alle drei Eigenschaften verfügbar sind [Brewer, 2000].

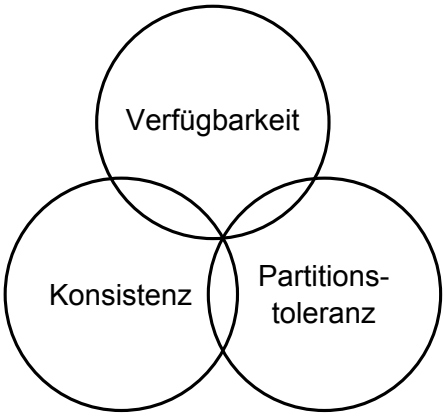

Abbildung 2.9: CAP-Theorem nach [Hewitt, 2011]

Concurrency Control

Wenn Transaktionen nebenläufig beziehungsweise parallel zueinander ausgeführt werden und auf denselben Speicherbereich zugreifen, sind Mechanismen notwendig, die den globalen *Schedule*[20] (dt. Ablaufplan) serialisieren [Abbott und Garcia-Molina, 1992]. So erlaubt *Concurrency Control* den Zugriff auf eine Datenbank, auf die mehrere Nutzer

[19] CAP steht für Konsistenz (engl. *Consistency*), Verfügbarkeit (engl. *Availability*) und Partitionstoleranz (engl. *Partition tolerance*).

[20] Ein *Schedule* ist eine beliebige Aneinanderreihung der Operationen aller Transaktionen, wobei jedoch die Operationen einer Transaktion in ihrer Reihenfolge nicht vertauscht werden dürfen [Pernul und Unland, 2003].

gleichzeitig zugreifen, während es für jeden Anwender so scheint, als würde er alleine auf dem System arbeiten [Bernstein und Goodman, 1981]. In den nachstehenden Abschnitten werden Verfahren beziehungsweise Algorithmen aufgezeigt, die das Koordinieren von Mehrbenutzerzugriffen verbessern und einen fehlerfreien Ablauf garantieren. Die Verfahren zur Synchronisation nebenläufiger Transaktionen lassen sich dabei in sperrende und nichtsperrende Verfahren unterteilen. Eine Ausnahme hierbei bildet das Verfahren der *Multiversion Concurrency Control* (MVCC), bei dem die Synchronisation der Änderungsoperationen sowohl durch sperrende als auch durch nichtsperrende Verfahren erfolgen kann.

- **Sperrende Verfahren**

  Sperrende Verfahren zeichnen sich vor allem dadurch aus, dass sie pessimistisch agieren [Guruzu und Mak, 2010], da bei ihnen schon vor dem Ausführen der Operation mit Konflikten gerechnet wird. Zu ihnen gehören beispielsweise das Zwei-Phasen-Sperr-Protokoll (engl. *Two Phase Locking* (2PL)) und das strikte Zwei-Phasen-Sperr-Protokoll (engl. *Strict Two Phase Locking* (S2PL). Hinzu kommt das hierarchische Sperren (engl. *Multi Granularity Locking* (MGL)) als Anwendung dieser Protokolle auf verschiedene Sperrgranulate.

  1. **Zwei-Phasen-Sperr-Protokoll**

     Das Zwei-Phasen-Sperr-Protokoll synchronisiert Schreib- und Leseoperationen, indem es Konflikte zwischen nebenläufigen Operationen erkennt und verhindert. Zu diesem Zweck erhält eine Transaktion erst durch das Setzen von Schreib- beziehungsweise Lesesperren auf ein Objekt die Berechtigung, dieses Objekt zu lesen oder zu verändern [Bernstein und Goodman, 1981]. Um Mehrbenutzeranomalien zu vermeiden, gilt beim Zwei-Phasen-Sperr-Protokoll zusätzlich die Regel, dass nach dem ersten Entsperren einer Transaktion keine weiteren Sperren mehr gesetzt werden dürfen [Saltzer und Kaashoek, 2009] (siehe Abbildung 2.10).

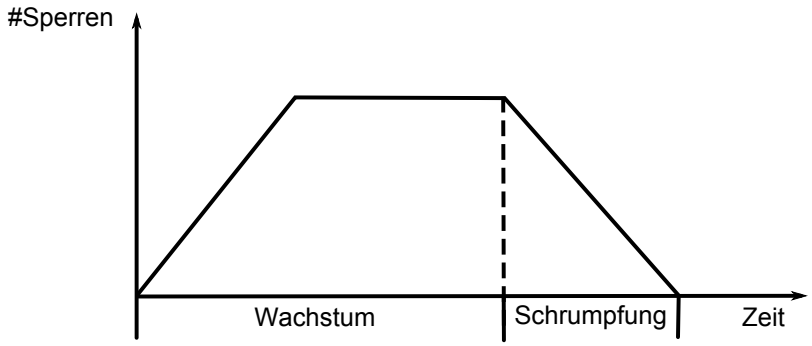

Abbildung 2.10: Zwei-Phasen-Sperr-Protokoll nach [Kemper und Eickler, 2006]

  2. **Striktes Zwei-Phasen-Sperr-Protokoll**

     Werden alle Sperrfreigaben erst am Ende einer Transaktion (engl. *End of Transaction* (EOT)) atomar ausgeführt, dann ist das Zwei-Phasen-Sperr-Protokoll strikt. Dies gewährleistet, dass kaskadierende Abbrüche nicht auftreten können (siehe Abbildung 2.11).

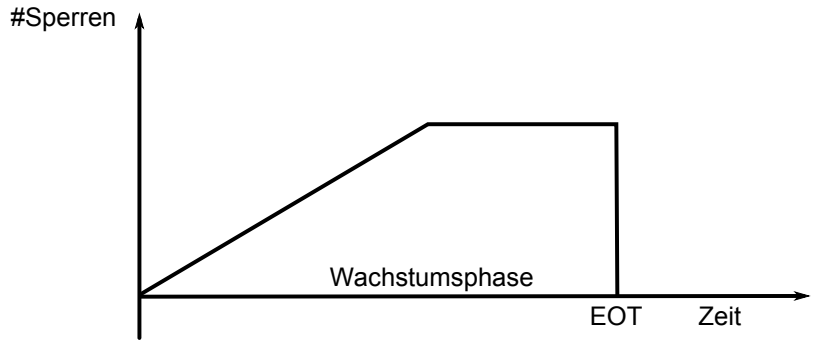

Abbildung 2.11: Striktes Zwei-Phasen-Sperr-Protokoll nach [Kemper und Eickler, 2006]

Wird kein striktes Zwei-Phasen-Sperr-Protokoll verwendet und eine Transaktion, die einen Datensatz frühzeitig freigegeben hat, muss neu gestartet werden, dann müssen aufgrund der kaskadierenden Abbrüche alle Transaktionen, die nach der Freigabe auf dem Datensatz gearbeitet haben, ebenfalls neu gestartet werden [Schubert, 2007].

3. **Konservatives Zwei-Phasen-Sperr-Protokoll**

   Ein weiteres Sperrprotokoll, welches das Zwei-Phasen-Sperr-Protokoll erweitert, stellt das konservative Zwei-Phasen-Sperr-Protokoll (engl. *Conservative Two Phase Locking* (C2PL)) dar. Durch das atomare Sperren der Daten zu Beginn einer Transaktion werden *Deadlocks*[21] ausgeschlossen, aber gleichzeitig die Nebenläufigkeit weiter eingeschränkt. Auch das S2PL kann durch die Kombination mit dem C2PL zu einem *Deadlock*-freien Protokoll (CS2PL) erweitert werden, bei dem sowohl die Sperr- als auch Entsperrphase atomar ausgeführt wird [Ward und Dafoulas, 2006]. Nachteilig für diese konservativen Sperrprotokolle ist, dass die benötigten Sperren schon zu Beginn der Transaktion bekannt sein müssen, was nur in einem Teil der Anwendungsfälle der Fall ist. Gleichzeitig sinkt jedoch auch der Verwaltungsaufwand für die Synchronisation, da weder auf *Deadlocks* überprüft noch Transaktionen zurückgesetzt werden müssen [Saake et al., 2005].

– **Hierarchische Sperrgranulate**

  Sperrgranulate geben an, wie groß die zu sperrenden Objekte sind [Lee und Liou, 1996]. Unterschieden wird zwischen groben Sperrgranulaten, wie beispielsweise das Sperren von einer ganzen Datenbank oder einer Relation, und feinen Sperrgranulaten, wie das Sperren von Tupeln oder Attributen [Levene und Loizou, 1999]. Dabei verringert sich der Verwaltungsaufwand für die Sperren umso mehr, je grobgranularer die Sperren erfolgen, da weniger Sperren gesetzt, getestet und gehalten werden müssen, wobei aber auch gleichzeitig der Grad der Nebenläufigkeit sinkt [Lee und Liou, 1996].

[21] Ein *Deadlock* ist der Zustand eines Systems, in dem Ausführungen nicht mehr fortschreiten können, obwohl mindestens noch ein Prozess aktiv ist, aber alle aktiven Prozesse warten [Taubenfeld und Francez, 1984].

Für lange Transaktionen wäre es wünschenswert grobe Granulate sperren zu können, um wenige Sperren setzen zu müssen, wenn sie auf viele kleinere Granulate zugreifen möchten. Dem entgegen steht das Sperren feiner Granulate durch kurze Transaktionen. Diese würden durch die Verwendung von feinen Sperrgranulaten verhindern, dass sie andere Transaktionen sinnlos blockieren [Vossen, 2000]. Um eine flexible Wahl eines bestimmten zu sperrenden Granulates je Transaktion zu ermöglichen, werden durch hierarchische Sperrgranulate zusätzliche Sperren eingeführt. Durch diese intentionalen Sperren werden Sperrabsichten, die sich weiter unten in der Sperrhierarchie befinden, angemeldet. Im speziellen sind dies die intentionale Lesesperre *irl* und die intentionale Schreibsperre *iwl* sowie deren Zusammenschluss *riwl* [Kemper und Eickler, 2006]. Durch diese zusätzlichen Sperren ergibt sich eine neue Kompatibilitätsmatrix (siehe Tabelle 2.4), welche die Verträglichkeitsmatrix für elementare Sperren (siehe Tabelle 2.3) erweitert.

	$rl_i(x)$	$wl_i(x)$
$rl_j(x)$	✓	–
$wl_j(x)$	–	–

Tabelle 2.3: Verträglichkeitsmatrix für elementare Sperren nach [Saake et al., 2005]

	$rl_i(x)$	$wl_i(x)$	$irl_i(x)$	$iwl_i(x)$	$riwl_i(x)$
$rl_j(x)$	✓	–	✓	–	–
$wl_j(x)$	–	–	–	–	–
$irl_j(x)$	✓	–	✓	✓	✓
$iwl_j(x)$	–	–	✓	✓	–
$riwl_j(x)$	–	–	✓	–	–

Tabelle 2.4: Kompatibilitätsmatrix für hierarchische Sperren nach [Saake et al., 2005]

Der Ablauf des hierarchischen Sperrens läuft mit der Hilfe der neuen Sperren wie folgt ab [Saake et al., 2005; Kemper und Eickler, 2006]:

1. Gesperrt wird *top-down* von der Wurzel zum Zielobjekt, wobei Lesesperren oder intentionale Lesesperren nur gesetzt werden können, wenn die Vorgängerknoten bereits eine intentionale Lese- oder Schreibsperre durch diese Transaktion erhalten haben. Ebenso darf eine Schreibsperre oder intentionale Schreibsperre nur dann erfolgen, wenn vorher in der Hierarchie bereits eine intentionale Schreibsperre gesetzt wurde.
2. Sperren dürfen nur bei Einhaltung der Kompatibilitätsmatrix erfolgen, wobei Sperren auch verschärft werden dürfen, indem eine Lesesperre zu einer Schreibsperre wird oder eine intentionale Lesesperre zu einer Lesesperre beziehungsweise intentionalen Schreibsperre.
3. Freigegeben wird *bottom-up* in umgekehrter Reihenfolge wie zum Sperrvorgang.

- **Nichtsperrende Verfahren**

  Im Kontrast zu den sperrenden Verfahren stehen die nichtsperrenden Verfahren, welche davon ausgehen, dass Konflikte selten auftreten und die demnach optimistischer als Sperrverfahren vorgehen [Saake et al., 2005]. Zu diesen Synchronisationsmodellen gehören zum Beispiel Zeitmarkenverfahren, der Serialisierbarkeitsgraphentester (SGT) und die optimistischen Verfahren. Die optimistischen Verfahren führen beispielsweise Operationen aus, ohne vorher zu überprüfen, ob Konflikte mit anderen Transaktionen vorliegen [Mandl, 2009]. Bevor solch ein optimistisches Verfahren das endgültige Abschließen einer Operation erlaubt, prüft es, ob sie die Atomarität bewahrt. Ist dies der Fall, wird die Transaktion abgeschlossen, ansonsten muss sie abgebrochen und neu gestartet werden [Herlihy, 1990].

  1. **Zeitmarkenverfahren**

     Essentiell für Zeitmarkenverfahren sind eindeutige Zeitmarken, die angeben, wann eine Transaktion gestartet wurde. Diese unterstützen die Zeitmarken-Reihenfolge-Regel (engl. *Timestamp-Ordering* (TO)), welche angibt, dass eine Transaktion mit einem früheren BOT[22]-Zeitstempel als eine zweite Transaktion ihre Lese- und Schreiboperationen auch vor dieser zweiten Transaktion ausführen muss. Voraussetzung dafür ist, dass beide Transaktionen auf dasselbe Datenobjekt zugreifen möchten [Zhong und Kambayashi, 1989a]. Abbildung 2.12 zeigt einen *Timestamp-Ordering*-Algorithmus. Mit Hilfe der maximalen Lesezeitstempel $max - rts(x)$ (Zeile 3) und Schreibzeitstempel $max - wts(x)$ (Zeile 2) auf ein Datenobjekt x kann durch den Algorithmus bestimmt werden, wann eine Transaktion zu spät kommt (Zeilen 5 und 12) und zurückgesetzt beziehungsweise neu gestartet werden muss (Zeilen 6 bis 7/ Zeilen 13 bis 14) [Saake et al., 2005].

```
1  Algorithmus Timestamp-Ordering (Input: T_i, x)
2  max − wts(x): Zeitstempel der letzten Schreiboperation auf x
3  max − rts(x): Zeitstempel der letzten Leseoperation auf x
4    IF    "T_i ist Leseoperation"
5    THEN  IF    ts(T_i) < max − wts(x)
6          THEN  T_i zurücksetzen
7                Neustart von T_i mit neuem Zeitstempel
8          ELSE  Leseoperation ausführen
9                Setze max − rts(x) auf max(ts(T_i), max − rts(x))
10         END IF
11   ELSE  "T_i ist Schreiboperation"
12         IF    ts(T_i) < max − rts(x) oder ts(T_i) < max − wts(x)
13         THEN  T_i zurücksetzen
14               Neustart von T_i mit neuem Zeitstempel
15         ELSE  Schreiboperation ausführen
16               Setze max − rts(x) auf ts(T_i)
17         END IF
18   END IF
19 Ende Algorithmus Timestamp-Ordering
```

Abbildung 2.12: *Timestamp-Ordering*-Algorithmus nach [Group, 2007]

[22] Beginn einer Transaktion (engl. *Begin of Transaction* (BOT))

Ein Vorteil des Zeitmarkenverfahrens ist, dass im Gegensatz zum Zwei-Phasen-Sperr-Protokoll keine *Deadlocks* auftreten können. Dem entgegen steht, dass die Abbruchwahrscheinlichkeit von Transaktionen von der Länge ihrer Ausführungszeit abhängig ist. So ist die Wahrscheinlichkeit für einen Abbruch sehr groß, wenn eine Transaktion sehr lang ist [Zhong und Kambayashi, 1989b]. Zusätzlich kann es vorkommen, dass diese Transaktion gegebenenfalls nie beendet wird, wenn viele kürzere Transaktionen ein wiederholtes Starten dieser Transaktion verursachen [Kedar, 2009a].

2. **Optimiertes Zeitmarkenverfahren**

   Der *Timestamp-Ordering*-Algorithmus lässt sich für blindes Schreiben optimieren. Zu diesem Zweck wird der Algorithmus, welcher in Abbildung 2.12 aufgezeigt wird, wie folgt abgeändert (siehe Abbildung 2.13).

```
10 ...
11   ELSE   "T_i ist Schreiboperation"
12          IF     ts(T_i) < max − rts(x)
13          THEN   T_i zurücksetzen
14                 Neustart von T_i mit neuem Zeitstempel
15          ELSE   IF     ts(T_i) < max − wts(x)
16                 THEN   Ignoriere Schreiboperation
17                 ELSE   Schreiboperation ausführen
18                        Setze max − rts(x) auf ts(T_i)
19                 END IF
20          END IF
21 ...
```

Abbildung 2.13: Optimierter *Timestamp-Ordering*-Algorithmus

Schreibende Operationen einer Transaktion werden somit ignoriert (Zeile 16), wenn sie in Konflikt mit dem maximalen Schreibzeitstempel, aber nicht in Konflikt mit dem maximalen Lesezeitstempel stehen. Diese Transaktionen werden folglich nicht abgebrochen und erlauben eine größere Menge an *Schedules* [Saake et al., 2005].

3. **Serialisierbarkeitsgraphentester**

   Das Verfahren des Serialisierbarkeitsgraphentesters besteht im Aufbau eines gerichteten Konfliktgraphen. Für die Konstruktion eines solchen Graphen gilt, dass er aus einer Menge Knoten und einer Menge Kanten besteht, wobei die Knoten die Transaktionen im *Schedule* darstellen und die Kanten zwischen einem Transaktionsknoten T_i und einem Transaktionsknotens T_j gebildet werden, wenn eine der folgenden Bedingungen erfüllt ist [Kedar, 2009b]:

   - T_i schreibt ein Objekt x bevor T_j dieses Objekt liest.
   - T_i liest ein Objekt x bevor T_j dieses Objekt schreibt.
   - T_i schreibt ein Objekt x bevor T_j dieses Objekt schreibt.

   Tritt während des Aufbaus eines Serialisierbarkeitsgraphen ein Zyklus auf, so wird die auslösende Operation zurückgewiesen und die dazugehörige Transaktion abgebrochen. Des Weiteren können freigegebene Transaktionen aus diesem Graphen nur entfernt werden, wenn sie keine eingehenden Kanten mehr haben [Saake et al., 2005].

4. **Optimistische Synchronisation**

   Im Gegensatz zu den bereits vorgestellten pessimistischen sperrenden Verfahren zur Synchronisation von nebenläufigen Transaktionen wird bei den optimistischen Verfahren davon ausgegangen, dass Konflikte selten vorkommen und aus diesem Grund auf das Setzen von Sperren verzichtet wird. Dies soll die Parallelität erhöhen sowie die Wartezeiten verkürzen [Meier, 2004; Schill und Springer, 2007]. Erst am Ende einer Transaktion erfolgt eine Überprüfung, ob Konflikte mit konkurrierenden Transaktionen bestehen. Dazu wird eine Transaktion in Lesephase (engl. *Read phase*), Validierungsphase (engl. *Validation phase*) und Schreibphase (engl. *Write phase*) unterteilt [Singh, 2009; Ossher, 2009]. Während der Lesephase arbeitet jede Transaktion auf Kopien der Datenobjekte, die zuletzt ein *Commit* ausgeführt haben. Dies erlaubt das Abbrechen der Transaktionen ohne Auswirkungen auf die Objekte. Wenn ein Objekt geschrieben wird, erfolgt das Erstellen einer neuen Version mit dem neuen Wert dieses Datenobjektes. Die neue Objektversion ist dabei für andere Transaktionen nicht sichtbar [Coulouris et al., 2005]. Die Validierungsphase stellt die wichtigste Phase der optimistischen Synchronisation dar und lässt sich auf zwei verschiedene Arten ausführen. So gibt es zum einen die Rückwärtsvalidierung und zum anderen die Vorwärtsvalidierung [Kienzle, 2003]. Bei der Rückwärtsvalidierung wird die zu bewertende Transaktion gegen bereits abgeschlossene Transaktionen validiert, während bei der Vorwärtsvalidierung die noch laufenden Transaktionen als Validierungspartner fungieren (siehe Abbildung 2.14). Dabei erkennt und löst die Vorwärtsvalidierung Konflikte früher als die Rückwärtsvalidierung [Lee und Lam, 1999].

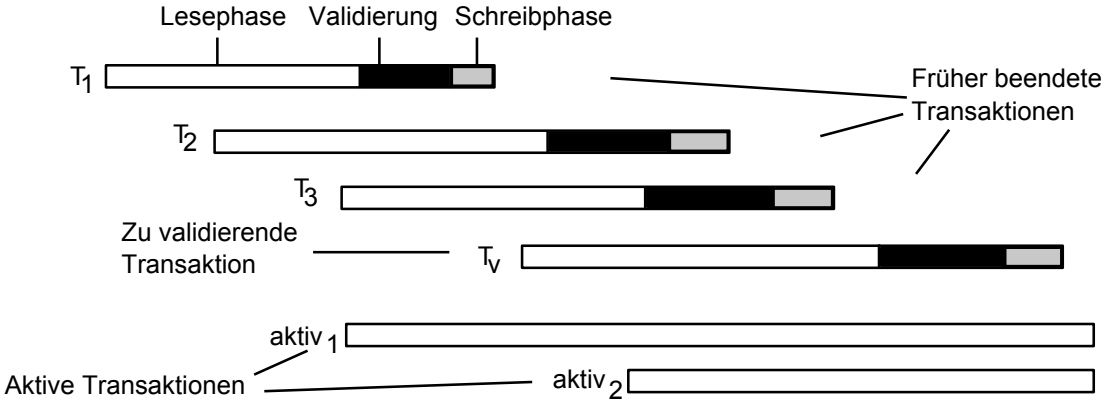

Abbildung 2.14: Validierung von Transaktionen nach [Coulouris et al., 2005]

   Nur wenn die Validierung erfolgreich abgeschlossen und kein Konflikt ermittelt wurde, werden die Änderungen in der Schreibphase dauerhaft übernommen. Reine Lesetransaktionen können sogar direkt nach der Validierung ein *Commit* ausführen [Coulouris et al., 2005].

- **Multiversion Concurrency Control (MVCC)**

  Das Grundprinzip der *Multiversion Concurrency Control* liegt darin, dass zu jedem geänderten Objekt mehrere Versionen angelegt und verwaltet werden [Bernstein und Goodman, 1983; Kudraß, 2007; Binder et al., 2007] und infolgedessen Sperren auf Objekte nicht benötigt werden [Eisentraut und Helmle, 2010]. Nach jeder Schreiboperation wird eine neue Version des geänderten Objektes angelegt, so dass keine Leseoperationen verboten werden müssen und auch keine Synchronisation von Schreibtransaktionen gegenüber Lesetransaktionen notwendig ist [Kudraß, 2007; Rahimi und Haug, 2010]. Im Gegensatz zu den Leseoperationen müssen Schreiboperationen aber auf die jeweils aktuelle Version eines Objektes zugreifen. Für deren Synchronisation können zum Beispiel Sperr- oder Zeitmarkenverfahren verwendet werden [Härder und Rahm, 2001].

  Abbildung 2.15 zeigt einen MVCC-Algorithmus. Schreibanfragen von einer Transaktion T_i auf ein Objekt X (Zeile 8) werden genau dann abgelehnt (Zeilen 10 bis 11), wenn eine jüngere Transaktion X gelesen hat, bevor T_i die Schreibanfrage gestellt hat (Zeile 9).

```
1  Algorithmus Multiversion (Input: T_i, X)
2  x_k: Version von X mit wts(x_k) ist größter ts ≤ ts(T_i)
3  rts(x_k): Lesezeitstempel der jüngsten Transaktion, die x_k gelesen hat
4  wts(x_k): Schreibzeitstempel der Transaktion die x_k geschrieben hat
5    IF    "T_i ist Leseoperation"
6    THEN  T_i darf x_k lesen
7          Setze rts(x_k) auf max(ts(T_i), rts(x_k))
8    ELSE  "T_i ist Schreiboperation"
9          IF    ts(T_i) < rts(x_k)
10         THEN  T_i zurücksetzen
11               Neustart von T_i mit neuem Zeitstempel
12         ELSE  Erstelle neue Version von x(x_n)
13               rts(x_n) = ts(T_i)
14               wts(x_n) = ts(T_i)
15         END IF
16   END IF
17 Ende Algorithmus Multiversion
```

Abbildung 2.15: *Multiversion Concurrency Control*-Algorithmus nach [Rahimi und Haug, 2010]

Neben dem Koordinieren, welche Version von einer Transaktion gelesen werden darf, muss die *Multiversion Concurrency Control* auch alte Versionen und deren Löschung verwalten. Gelöscht werden dürfen sie genau dann, wenn sie für den weiteren Ablauf und zukünftige Transaktionen nicht mehr von Interesse sind [Khoshafian und Baker, 1996]. Dies ist der Fall, wenn eine Version x_i von Objekt X einen geringeren Schreibzeitstempel wts hat als eine Version x_j und gleichzeitig $wts(x_j)$ kleiner ist als der BOT-Zeitstempel der ältesten Lesetransaktion [Saake et al., 2005].

2.3 Speichermanager

Zusätzlich zur Transaktionsverwaltung sind Maßnahmen notwendig, die im Falle von Transaktionsabbrüchen oder Systemfehlern verlorengegangene Daten wiederherstellen sowie durch den Transaktionsmanager freigegebene Operationen an den Puffer beziehungsweise persistenten Speicher weitergeben können [Lorie, 1977]. So besteht der Speichermanager aus einer Wiederherstellungskomponente, dem *Recoverymanager* und dem Puffermanager zur Verwaltung des flüchtigen Speichers [Bernstein et al., 1987].

Puffermanager

Da nicht der gesamte persistente Speicher zur selben Zeit in den Puffer geladen werden kann, was für schnelle Datenzugriffe wünschenswert wäre, müssen Daten mit Hilfe eines geeigneten Verfahrens aus dem Puffer wieder in den persistenten Speicher überführt werden und umgekehrt [Effelsberg und Härder, 1984; Bernstein et al., 1987]. Zu diesem Zweck gibt es nach Härder und Reuter die folgenden Seitenersetzungs-, Propagierungs- und Einbringstrategien, die festlegen, wann und wie Daten aus dem Puffer verdrängt, ersetzt und auf den persistenten Speicher zurückgeschrieben werden [Härder und Reuter, 1983].

- **Seitenersetzungsstrategien**

  Seitenersetzungsstrategien geben an, welche Daten aus dem Puffer verdrängt werden, wenn neue Daten in einen voll belegten Puffer eingebracht werden sollen.

  1. **UNDO/STEAL**

     Bei dieser Variante können auch dann Daten aus dem Puffer in den persistenten Speicher überführt werden, wenn die Transaktion, welche die Änderungen vorgenommen hat, noch kein *Commit* ausgeführt hat.

  2. **NO-UNDO/¬STEAL**

     Um zu vermeiden, dass keine Daten im persistenten Speicher stehen, welche von nicht erfolgreich beendeten Transaktionen geändert wurden, kann die alternative NO-UNDO Strategie angewandt werden. Dies bedeutet jedoch, dass ein Zustand auftreten kann, in dessen Folge keine Daten mehr ausgelagert werden können, weil der Puffer voll ist.

- **Propagierungsstrategien**

  Durch die Propagierungsstrategien wird festgelegt, ob Daten beim *Commit* einer Transaktion aus dem Puffer in den persistenten Speicher geschrieben werden müssen oder nicht.

  1. **REDO/¬FORCE**

     Bei der REDO-Strategie ist es nicht notwendig, dass zum *Commit* alle Daten einer Transaktion in den persistenten Speicher geschrieben werden.

  2. **NO-REDO/FORCE**

     Dahingegen ist es bei der NO-REDO-Strategie angedacht, dass spätestens zum *Commit* einer Transaktion alle durch sie geänderten Daten in den persistenten Speicher überführt werden müssen.

- **Einbringstrategien**

  Die Einbringstrategien legen fest, wie das Schreiben von Daten aus dem Puffer in den persistenten Speicher durchgeführt wird. Dabei wird zwischen der direkten und der indirekten Zuordnung unterschieden.

  1. **Direkte Zuordnung/¬ATOMIC**

     Bei der direkten Zuordnung erfolgt das Ersetzen der Daten direkt auf dem ursprünglichen Datenbestand, was ein Löschen der alten Daten zur Folge hat.

  2. **Indirekte Zuordnung/ATOMIC**

     Im Gegensatz zur direkten Zuordnung werden bei der indirekten Variante separate Blöcke gehalten, welche die letzten beiden konsistenten Zustände beinhalten, so dass die verdrängten Daten aus dem Puffer in den vorletzten Block kopiert werden können und so der letzte konsistente Zustand erhalten bleibt.

Recoverymanager

Der *Recoverymanager* ist eine essentielle Komponente zur Wiederherstellung eines konsistenten Zustandes der persistent gespeicherten Daten nach Transaktions- oder Systemfehlern [Gray et al., 1981b]. Dabei steht er sowohl mit dem Transaktionsmanager als auch dem Puffermanager in Verbindung. Zum einen muss er garantieren, dass Änderungen von abgebrochenen Transaktionen erfolgreich zurückgesetzt werden beziehungsweise Effekte von Transaktionen, die bereits ein *Commit* ausgeführt haben, auch wirklich im persistenten Speicher vorhanden sind. Zum anderen muss der *Recoverymanager* die Daten, welche sich noch nicht im persistenten Speicher, sondern im Puffer befinden, in die Datensicherung einbeziehen können [Bernstein et al., 1987].

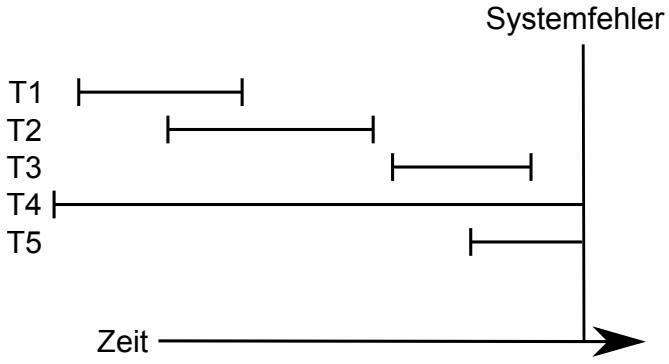

Abbildung 2.16: Mögliche Situation nach einem Systemfehler nach [Gray, 1978]

Abbildung 2.16 zeigt eine mögliche Situation nach einem Systemfehler. Von Interesse für die Datenwiederherstellung sind die zum Zeitpunkt des aufgetretenen Systemfehlers schon beendeten Transaktionen T1, T2 und T3, deren Ergebnisse entweder schon persistent gespeichert sind oder durch eine *redo*-Prozedur nachvollzogen werden können. Ebenso wichtig für die Datenwiederherstellung sind die noch nicht beendeten und zum Fehlerzeitpunkt aktiven Transaktionen T4 und T5, von denen keine Änderungen im persistenten Speicher sein dürfen und ansonsten durch eine *undo*-Prozedur rückgängig gemacht werden müssen [Lorie, 1977; Gray, 1978].

- **Logbuch**

  Das Logbuch ist eine chronologische Liste aller Änderungen, die in den stabilen Speicher geschrieben wurden, wobei auch der Status der Transaktionen festgehalten wird. Im Gegensatz zu Daten, die durch Transaktionen manipuliert wurden, ist es angebracht, Logeinträge schnellstmöglich persistent zu sichern [Härder und Reuter, 1983] und nicht lange in einem Puffer zwischenzuspeichern. Zu diesem Zweck wird ein separater Logpuffer benötigt, welcher kontinuierlich ausgelesen werden muss.

  1. **Aufbau**

     Ein Logeintrag besteht aus allen für die Wiederherstellung von Daten notwendigen Informationen. Dazu wird jeder Eintrag mit einer einzigartigen ID durchnummeriert, um eine nachvollziehbare Verkettung der Einträge zu erhalten. Zusätzlich wird ein Merkmal benötigt, welches die Transaktion, die den Eintrag ausgelöst hat, eindeutig identifizieren kann. Weitere Informationen sind die ID des geänderten Speicherobjektes und die eventuell benötigten *undo*- und *redo*-Informationen. Zusätzlich kann auch ein Verweis auf den vorhergehenden Eintrag ein und derselben Transaktion mitgeführt werden [Gray et al., 1981b].

     Von dieser Struktur ausgenommene Einträge bilden die BOT-, *Commit*- und *Abort*-Hinweise, die angeben, welche Transaktionen gestartet und wann diese Transaktionen beendet wurden [Härder und Reuter, 1983]. Wenn mit Sicherungspunkten gearbeitet wird, muss auch hierfür ein gesonderter Log-Eintrag verfasst werden [Gray, 1978].

  2. **Physisch und logisch**

     Die *redo*- und *undo*-Informationen können auf zwei verschiedene Arten gesichert werden. Zum einen durch das physische und zum anderen durch das logische Protokollieren. Beim physischen Protokollieren werden die Zustände der Daten als *Before*- beziehungsweise *After-Images* gesichert, wohingegen bei der logischen Variante die ausgeführten Operationen geloggt werden, wodurch deren inverse Operationen hergeleitet werden können [Härder und Reuter, 1983].

  3. **Write Ahead Log**

     Das *Write Ahead Log* (WAL) ist ein spezielles Prinzip, welches festlegt, dass eine Transaktion ein *Commit* erst ausführen darf, wenn deren Logeinträge ausgeschrieben wurden und dass vor dem Auslagern von veränderten Daten die dazugehörigen Logeinträge in das Logbuch geschrieben werden müssen [Gray, 1978].

- **Sicherungspunkte**

  Sicherungspunkte haben den Zweck, Punkte festzulegen, vor denen eine Wiederherstellung nicht mehr notwendig ist, da zu diesem Zeitpunkt bereits alle Änderungen in den persistenten Speicher geschrieben wurden. Folglich muss nicht das ganze Log bis zum Anfang für die Wiederherstellung abgearbeitet werden, sondern nur bis zum Sicherungspunkt [Gray, 1978]. Dabei wird zwischen transaktionskonsistenten, aktionskonsistenten und unscharfen Sicherungspunkten unterschieden.

1. **Transaktionskonsistent**

   Wird ein transaktionskonsistenter Sicherungspunkt angemeldet, werden alle aktiven Transaktionen bis zum Ende ausgeführt und solange keine neuen Transaktionen zugelassen, bis sämtliche geänderten Daten aus dem Puffer in den stabilen Speicher geschrieben wurden [Gray et al., 1981b].

2. **Aktionskonsistent**

   Bei den aktionskonsistenten Sicherungspunkten wird hingegen nur vorausgesetzt, dass keine Änderungsoperationen zum Sicherungszeitpunkt mehr aktiv sind. Dies führt dazu, dass weniger und kürzere Unterbrechungen stattfinden. Gleichzeitig sorgt es jedoch auch dafür, dass nur noch die *redo*-Phase unnötig wird, jedoch nicht das Zurücksetzen von Änderungen unvollständiger Transaktionen, welche mit in den stabilen Speicher geschrieben werden [Härder und Reuter, 1983].

3. **Unscharf**

   Beim Einsatz von unscharfen Sicherungspunkten verbleibt der persistente Speicher in einem unscharfen Zustand, da nicht versucht wird, alle Daten synchron auszuschreiben, sondern asynchron. Der Aufwand zum Anlegen von Sicherungspunkten wird bei diesem Verfahren minimiert, es müssen aber gegebenenfalls noch *redo*-Informationen vor dem letzten Sicherungspunkt berücksichtigt und *Hot-Spot*-Seiten[23] extra behandelt werden. Als zusätzliche Datenstruktur wird außerdem eine *Dirty-Page*-Liste benötigt, welche die noch nicht persistent gesicherten Änderungen enthält sowie die Verweise auf die Logeinträge, welche die Daten in den Zustand *dirty* überführt haben [Härder und Reuter, 1983; Härder und Rahm, 2001].

- **ARIES**

  Der *Algorithm for Recovery and Isolation Exploiting Semantics* (ARIES) stellt ein spezielles Recovery-Verfahren dar, welches das WAL-Protokoll sowie die Pufferstrategien UNDO und REDO verwendet. Des Weiteren kombiniert ARIES ein physisches *redo* mit einem logischen *undo* und nutzt unscharfe Sicherungspunkte. Zusätzlich dazu erhalten die Daten des persistenten Speichers einen Verweis auf den Logeintrag, welcher angibt, wann diese Daten das letzte Mal geändert wurden sowie jede Transaktion einen Verweis auf den letzten von ihr geschriebenen Logeintrag. Eine zusätzliche Datenstruktur namens *Dirty-Page*-Liste verwaltet alle Daten, die zwar geändert, aber noch nicht in den persistenten Speicher überführt wurden. Sie enthält auch einen Verweis auf den Logeintrag, der angelegt wurde, als die Daten in den Zustand *dirty* versetzt wurden [Mohan et al., 1992].

- **Schattenspeicherverfahren**

  Mit dem Schattenspeicherverfahren bietet sich eine Alternative zu den logbasierten Recovery-Verfahren, indem Änderungen nicht auf den Originaldaten, sondern auf Kopien durchgeführt werden. Zu diesem Zweck werden virtuelle Seitentabellen angelegt, die zum einen auf die Originaldaten und zum anderen auf deren Kopien verweisen [Silberschatz et al., 2001]. Zwischen den auf die Originale verweisenden Tabellen und den Tabellen für die Kopien kann atomar gewechselt werden, so dass

[23] *Hot-Spot*-Seiten sind Seiten, welche häufig durch verschiedene Transaktionen modifiziert werden [Gray und Reuter, 1992].

beim Abbruch einer Transaktion nur auf die Originaltabelle zugegriffen werden muss, um den alten Zustand der Daten zu erhalten. Ist eine Transaktion jedoch erfolgreich, muss nur die Kopietabelle zur Originaltabelle deklariert werden [Gray et al., 1981b]. Dem vereinfachten Rücksetzen auf den letzten konsistenten Zustand des persistenten Speichers und der fehlenden Notwendigkeit für das Führen eines Logs stehen jedoch der zusätzlich für Kopien benötigte Speicherbedarf sowie die Zerstreuung der Daten gegenüber [Lorie, 1977].

2.4 Erweiterung der objektorientierten Programmierung

Für die Entwicklungsplanung einer Software ist es hilfreich, das zu entstehende System zunächst in kleinere, leichter überschaubare Einheiten zu zerbrechen, so dass durch deren Zusammenfügen das Gesamtsystem entsteht. Programmiersprachen haben sich im Laufe der Zeit dahingehend entwickelt, solche Designtechniken zu unterstützen [Kiczales et al., 1997; Prehofer, 1997]. So besteht zum Beispiel mit der prozeduralen Programmierung ein Programmierparadigma zur Entwicklung von Prozeduren und Funktionen, die bestimmte Aufgaben erfüllen, in beliebigen Umgebungen eingesetzt und mit Hilfe von inhaltlichen beziehungsweise logischen Kriterien zu Funktionsbibliotheken zusammengefasst werden können [Hering, 2005]. Als eines noch näher an der Designplanung orientiertes Programmierschema etablierte sich dann zunehmend die objektorientierte Programmierung (OOP). Ziele dieses Programmierparadigmas sind vor allem die Verbesserung der Erweiterbarkeit, Wiederverwendbarkeit, Testbarkeit und Wartbarkeit von Software mit Hilfe von Datenkapselung, Polymorphie und Vererbung (siehe Abbildung 2.17) [Lahres und Rayman, 2009]. Bei der objektorientierten Programmierung erfolgt der Zusammenschluss von zueinander gehörenden Daten und Anweisungen zu geschlossenen und eigenständigen Einheiten in Objekte [Wegner, 1990; Steyer, 2010], welche ihre Daten schützen und die Art und Weise des Zugriffs anderer Objekte auf diese Daten definieren [Rodewig und Negm-Awad, 2006].

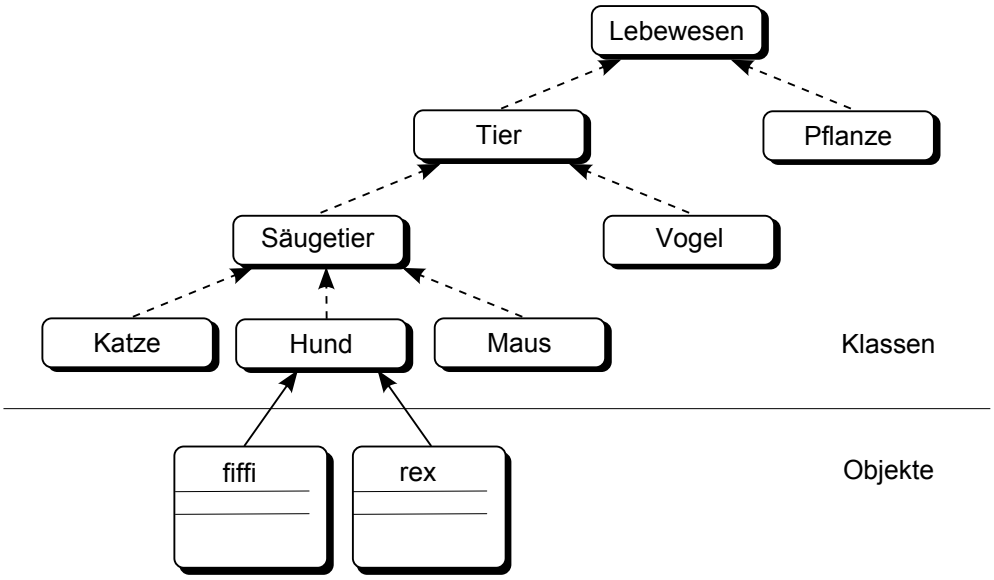

Abbildung 2.17: Eine Vererbungshierarchie (Auszug) nach [Pepper, 2007]

Doch auch die objektorientierte Programmierung hat ihre Grenzen. So existieren Programmieranforderungen, die weder durch die prozedurale noch durch die objektorientierte Programmierung gelöst werden können. Viele Programmiertechniken, die in der Vergangenheit entwickelt wurden, haben die Aufteilung des Quellcodes in Belange (engl. *Separation of Concerns*) zwar stetig verbessert, unterliegen jedoch jeweils spezifischen Einschränkungen [Tarr et al., 1999; Elrad et al., 2001].

Separation of Concerns

Ein Belang (engl. *Concern*) ist eine modulare Codeeinheit, die ein bestimmtes Ziel beziehungsweise einen bestimmten Sinn und Zweck erfüllt. Durch die Separation dieser Belange wird sowohl die Flexibilität als auch die Konfigurierbarkeit von Software begünstigt [Colyer et al., 2004]. Die prozedurale Programmierung setzt die Separation einer eingeschränkten Menge von Belangen zum Beispiel um, indem sie diese Belange in Form von Prozeduren und Funktionen abspaltet [Newton, 2007]. In der objektorientierten Programmierung hingegen definieren sich Belange durch Klassen [Ossher und Tarr, 2000]. Abbildung 2.18 zeigt ein Beispiel für eine *Queue*-Umsetzung durch die objektorientierte Programmierung.

```
1  namespace util {
2    class Item {
3      friend class Queue;
4      Item* next;
5      public: Item() : next(0) {}
6    };
7
8    class Queue {
9      Item* first;
10     Item* last;
11     public: Queue() : first(0), last(0) {}
12
13     void enqueue (Item* item) {
14       if (last) {last->next = item; last = item;}
15       else{ last = first = item;  }
16     }
17
18     Item* dequeue () {
19       Item* res = first;
20       if (first == last) {first = last = 0;}
21       else{ first = first->next; }
22       return res;
23     }
24   };
25 }
```

Abbildung 2.18: Beispiel: Objektorientierte Implementierung der *Queue*-Klasse nach [Spinczyk und Lohmann, 2005]

Durch diese Realisierung einer *Queue* lassen sich die einzelnen Belange leicht identifizieren. Die Unterteilung in die Methoden *enqueue* (Zeilen 13 bis 16) und *dequeue* (Zeilen 18 bis 23) zum Hinzufügen und zum Entfernen von Elementen stellen hierbei die separierten Belange dar, aus denen eine *Queue* besteht. Ein zusätzlicher Belang wird durch die Klasse *Item* (Zeilen 2 bis 6) dargestellt, welche die Reihenfolge der jeweiligen Elemente regelt, indem jedes Element auf das nachfolgende Element verweist.

Problematisch für diese Programmierparadigmen werden Belange, welche sich vereinzelt durch den gesamten Quellcode ziehen, wodurch sie mit Hilfe von Konstrukten, wie Methoden und Klassen, nicht mehr gebunden werden können [Newton, 2007]. Das liegt daran, dass die Aufteilung der Belange entlang nur einer dominanten Dimension, laut Tarr et al., zur Tyrannei der dominanten Zerlegung führt [Tarr et al., 1999]. Ein Beispiel für einen solchen nicht fassbaren Belang sind Fehlerbehandlungen. Viele Anwendungen benötigen die Eigenschaft, Fehler zu erkennen und entsprechend zu handeln, sollte ein solcher auftreten. Diese Fehlerabfrage muss jedoch in jeder Methode wiederholt ausgeführt werden und das an einer Vielzahl von Stellen im gesamten Quellcode, wobei der auszuführende Code für diese Abfrage oft identisch ist [Bruntink et al., 2004]. Weitere Beispiele für diese querschneidenden Belange (engl. *Crosscutting Concerns*) sind Logging-Mechanismen, Statistiken, Autorisierungen, Transaktionen und Caching [Pessemier et al., 2006; Kästner et al., 2008]. Besonders problematisch ist das Verflechten der eigentlichen Logik der Methode mit dem querschneidenden Belang, wodurch es schwerer wird, den Code zu lesen und zu pflegen. Somit wäre es wünschenswert und für das Gewährleisten von Konfigurierbarkeit sogar notwendig, wenn diese Methode nur ihren spezifischen Code beinhalten würde [Hürsch und Lopes, 1995].

```
1  class Queue {
2    Item* first; Item* last;
3    int counter;
4    os::Mutex lock;
5    public: Queue() : first(0), last(0) { counter = 0; }
6
7    void enqueue (Item* item) {
8      lock.enter();
9      try {
10       if (item == 0) { throw QueueInvalidItemError(); }
11       if (last) {last->next = item; last = item;}
12       else {last = first = item;}
13       ++counter;
14     } catch (...) { lock.leave(); throw; }
15     lock.leave();
16   }
17
18   Item* dequeue () {
19     Item* res = first;
20     lock.enter();
21     try {
22       res = first;
23       if (first == last)
24          {first = last = 0;}
25       else {first = first->next;}
26       if (counter > 0) { -counter; }
27       if (res == 0) { throw QueueEmptyError(); }
28     } catch (...) { lock.leave(); throw; }
29     lock.leave();
30     return res;
31   }
32
33   int count() { return counter };
34 };
```

Abbildung 2.19: Beispiel: Erweiterung der *Queue*-Klasse nach [Spinczyk und Lohmann, 2005]

Abbildung 2.19 zeigt eine Erweiterung der *Queue*-Klasse. Ein Elementzähler (Zeilen 3, 5, 13, 26 und 33), Fehlerbehandlungen (Zeilen 9, 10, 14, 21, 27 und 28) und

Threadsicherheit (Zeilen 4, 8, 14, 15, 20, 28 und 29) machen die ursprüngliche *Queue*-Klasse unübersichtlicher, da sich durch den gesamten Quellcode Stücke der Erweiterungen verteilen [Spinczyk und Lohmann, 2005].

Klassifikation der querschneidenden Belange

Querschneidende Belange lassen sich in zwei verschiedene Klassifikationen unterteilen. So existiert zum einen die Unterteilung in homogene und heterogene querschneidende Belange und zum anderen die Unterscheidung zwischen statischen und dynamischen querschneidenden Belangen [Apel et al., 2006a]. Zusätzlich zu diesen zwei Klassifikationen kann die Differenzierung auch in zerstreute (engl. *scattered*) und verflochtene (engl. *tangled*) Belange erfolgen [Kiczales et al., 1997].

- **Homogen und heterogen**

  Homogene querschneidende Belange zeichnen sich dadurch aus, dass an verschiedenen Stellen im Programm derselbe Quellcode eingebunden wird. Wohingegen bei den heterogenen querschneidenden Belangen an verschiedenen Stellen im Programm unterschiedlicher Quelltext eingewebt wird [Colyer et al., 2004].

- **Statisch und dynamisch**

  Bei der Trennung zwischen homogenen und heterogenen querschneidenden Belangen wird angegeben, an welchen Stellen im Quellcode, welche Erweiterung eingebracht wird. Im Gegensatz dazu ist es bei der Unterscheidung von statischen und dynamischen querschneidenden Belangen von Bedeutung, zu welcher Zeit das Programm durch die Belange beeinflusst wird [Masuhara und Kiczales, 2003]. Statische querschneidende Belange beeinflussen die feste Struktur des Programms, indem zum Beispiel neue Funktionen und neue Superklassen hinzugefügt werden. Dahingegen beeinflussen dynamische querschneidende Belange die Ausführung der Anwendung in Form von Ereignissen und Aktionen [Mezini und Ostermann, 2004; Wand et al., 2004]. Dazu gehören zum Beispiel der Aufruf von Funktionen oder das Ausgeben von *Exceptions* [Apel et al., 2007a].

- **Zerstreut und verflochten**

  Ein weiterer Unterschied zwischen Belangen ergibt sich daraus, ob ein Belang über mehrere Module zerstreut ist oder sich mehrere Belange in einem Modul verflechten [Kiczales et al., 1997; Lieberherr et al., 2003]. In der *Queue*-Klasse aus Abbildung 2.19 ist beispielsweise der Belang zur Gewährleistung der Threadsicherheit über die ganze Klasse zerstreut und zusammen mit dem Belang zur Fehlerbehandlung verflochten.

2.4.1 Aspektorientierte Programmierung

Die aspektorientierte Programmierung setzt an dem Punkt an, an dem die objektorientierte Programmierung es nicht mehr möglich macht, jeden Belang zu modularisieren, da die Belange sich durch das gesamte System zerstreuen [Kiczales et al., 1997]. Zu diesem Zweck erweitert die aspektorientierte Programmierung die objektorientierte Programmierung und kapselt jeden querschneidenden Belang in einem separaten Modul (Aspekt). Außerdem stellt sie funktionelle Verknüpfungen zwischen den Aspekten

und dem Zielquellcode (engl. *Componentcode*), in dem die im Aspekt-Modul befindlichen Belange benötigt werden, bereit. Das Ermitteln dieser Schnittpunkte (*Joinpoints*) und das Einweben der Funktionalität des Aspektes (*Advicecode*) an diesen Stellen im Quellcode erfolgt durch den Aspektweber [Gal et al., 2002] (siehe Abbildung 2.20).

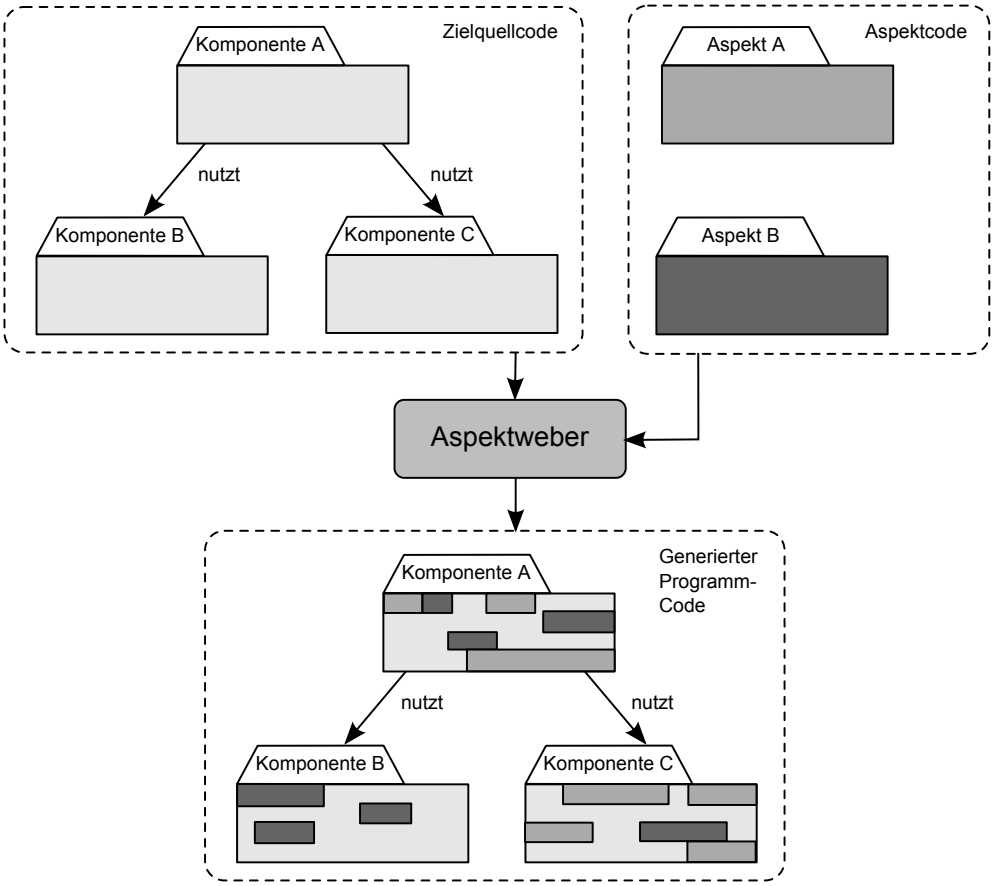

Abbildung 2.20: Aspektcode weben [Mahrenholz et al., 2002b]

- **Joinpoints**

  Joinpoints markieren die Stellen im Quellcode, an denen die Modifikationen des Quellcodes eingebunden werden [Mahrenholz et al., 2002a]. Die Art des *Joinpoints* variiert je nach Umsetzung der aspektorientierten Programmierung. Zum einen gibt es *Joinpoints*, die einen Punkt im Kontrollfluss eines laufenden Programms markieren (dynamische *Joinpoints*), wie Funktionsausführungen und Funktionsaufrufe [Wand et al., 2004]. Zum anderen können *Joinpoints* auch Schlüsselworte einer Programmiersprache sein (statische *Joinpoints*) [Eichberg et al., 2004]. Dazu gehören zum Beispiel Identifikatoren wie *class*, *function* und *type*.

- **Pointcuts**

  Pointcuts ermöglichen die Adressierung der *Joinpoints* im Quellcode, indem sie die *Joinpoints* beschreiben, an denen der Aspekt im Quellcode eingebunden werden soll [Masuhara et al., 2003; Apel et al., 2007a]. Genauso wie bei den *Joinpoints* gibt es statische und dynamische *Pointcuts*, abhängig von der Art der *Joinpoints*.

- **Advice**

  Ein *Advice* ist der Teil eines Aspektes, der die Art der Modifikation des Quellcodes beinhaltet [Spinczyk et al., 2005]. Der *Advice* wird dabei an den Stellen im Zielquellcode eingewebt, die vorher durch die *Pointcuts* definiert wurden. Für dynamische *Joinpoints* gilt, dass der *Advice*, abhängig von zusätzlichen Schlüsselworten wie *before*, *after* und *around* vor, um oder nach einem *Joinpoint* ausgeführt wird [Mahrenholz et al., 2002a]. Die Verwendung von *before* ist dann sinnvoll, wenn das Verändern von Übergabewerten von Funktionen vor dessen Aufruf gewünscht wird [Masuhara et al., 2003]. Ist das Ziel jedoch die Beeinflussung von Rückgabewerten von Funktionen beziehungsweise deren Auswertung, dann wird die Ausführung des *Advicecodes* nach dem *Joinpoint* durch den Einsatz des Wortes *after* ermöglicht [Spinczyk et al., 2005]. Das Ausführen des *Advicecodes* durch *around* erfolgt wie bei *before* vor dem *Joinpoint*, aber innerhalb des *Advicecodes* kann der ursprüngliche Code aufgerufen werden. Geschieht dies nicht, erfolgt sozusagen eine vollständige Ersetzung des Originalcodes mit dem *Advicecode* [Wand et al., 2004]. Im Gegenzug dazu können statische querschneidende Belange mit der Hilfe von *Introductions*, die eine spezielle Art von *Advices* darstellen, umgesetzt werden. So können *Introductions* beispielsweise Funktionen und Attribute zu *Joinpoints* des Typs *class*, die in einem *Pointcut* enthalten sind, hinzufügen [Mahrenholz et al., 2002a].

2.4.2 Merkmalsorientierte Programmierung

Im Gegensatz zur aspektorientierten Programmierung, durch welche die Wartbarkeit und Verständlichkeit eines Programms erhöht wird, indem querschneidende Belange in Aspekte gekapselt werden [Czarnecki und Eisenecker, 2000], bietet die merkmalsorientierte Programmierung die Möglichkeit, die Erweiterbarkeit einer Software durch schrittweise Verfeinerung (engl. *Step-wise Refinement*) zu verbessern [Batory et al., 2003].

- **Step-wise Refinement**

  Step-wise Refinement ermöglicht das Entwickeln von komplexeren Programmen auf Grundlage eines einfachen Programms, indem zu dem einfachen Programm Merkmale (engl. *Features*) schrittweise hinzugefügt werden [Batory et al., 2003]. Dabei können wie bei der Vererbung Methoden der Superklasse überschrieben und zusätzlich Funktionen eines Merkmals in den Inhalt eines weiteren Merkmals integriert werden [Prehofer, 1997]. Zu den möglichen Verfeinerungen gehören Klassenverfeinerungen, bei denen neue Elemente zu einer Klasse hinzugefügt werden, und Verfeinerungen von bestehenden Elementen wie zum Beispiel Methoden [Apel et al., 2008]. Durch verschiedene Kombinationen einer Vielzahl von Merkmalen, wird eine hohe Konfigurierbarkeit der Anwendung ermöglicht. Dabei muss ein Merkmal nicht nur aus der Verfeinerung einer Klasse bestehen, sondern kann ein Zusammenschluss aus mehreren Verfeinerungen sowie aus zusätzlichen neuen Klassen sein (*Collaborations*) [Smaragdakis und Batory, 2002; Batory et al., 2003]. Abbildung 2.21 zeigt einen *Collaborationsstack*, bestehend aus vertikal arrangierten Klassen ($c_1 - c_3$) sowie horizontal aufgestellten *Collaborations* ($f_1 - f_3$), die mehrere Klassen überspannen [Leich et al., 2005b].

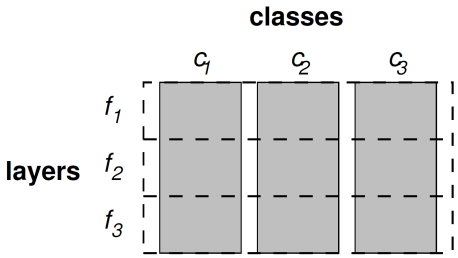

Abbildung 2.21: *Collaborationstack* [Leich et al., 2005b]

- **Beispiel: Merkmalsorientierte Programmierung**

  Ein Beispiel für eine merkmalsorientierte Erweiterung eines Graphen zu einem gewichteten Graphen zeigt Abbildung 2.22. Dabei wird der Basisgraph durch die Klassen *Graph*, *Node* und *Edge* dargestellt. Die Basisklassen *Graph* und *Edge* erfahren eine Verfeinerung und eine Erweiterung durch die zusätzlich hinzugefügte Klasse *Weight* [Apel et al., 2008].

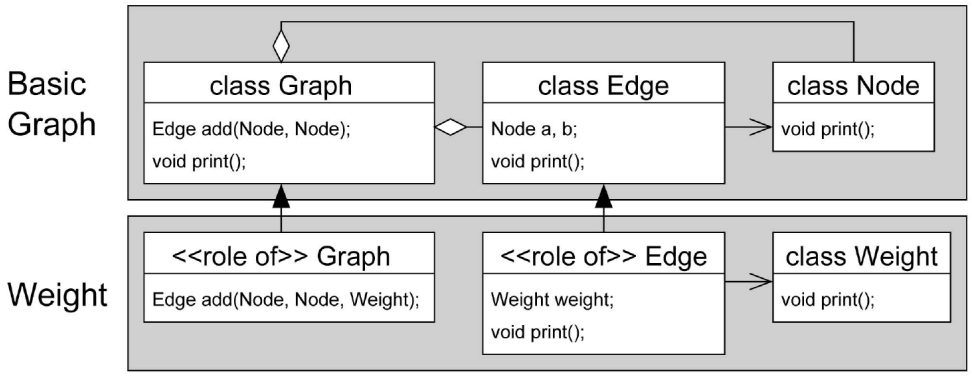

Abbildung 2.22: Erweiterung eines Basisgraphen zu einem gewichteten Graph mit FOP [Apel et al., 2008]

2.4.3 Aspectual Feature Modules

Die aspektorientierte und die merkmalsorientierte Programmierung zeichnen sich sowohl durch Vor- als auch Nachteile aus, wodurch sie für konkrete Einsatzgebiete nützlich werden. Die aspektorientierte Programmierung zeigt ihre Stärke beispielsweise im Modularisieren von homogenen querschneidenden Belangen, wohingegen die merkmalsorientierte Programmierung bei der Modularisierung von heterogenen querschneidenden Belangen von Vorteil ist [Apel et al., 2005a]. Des Weiteren unterstützt die merkmalsorientierte Programmierung vor allem statische querschneidende Belange sowie den Zusammenhalt aller zum Merkmal gehörenden Artefakte und die aspektorientierte Programmierung dynamische querschneidende Belange sowie das Verbinden von Merkmalen mit unterschiedlichem strukturellen Aufbau. Wie in Abbildung 2.23 zu sehen ist, gleichen beide Techniken die Schwachpunkte der jeweils anderen Technik aus, wodurch eine Symbiose beider Techniken wünschenswert wird [Apel et al., 2006b].

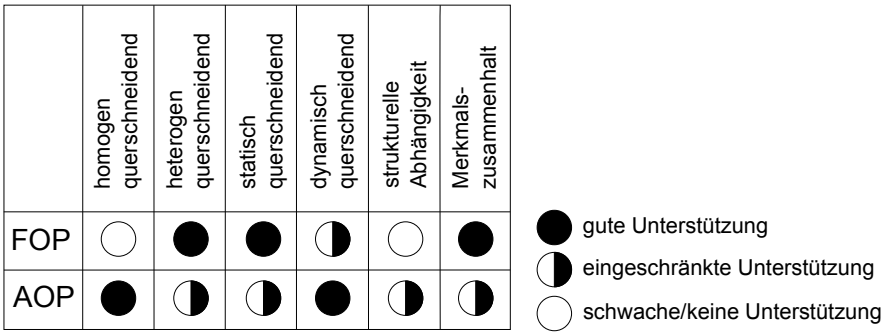

Abbildung 2.23: Vor- und Nachteile von AOP und FOP nach [Apel et al., 2006b]

Die *Aspectual Feature Modules* stellen eine Umsetzung dieser Verknüpfung von aspektorientierter und merkmalsorientierter Programmierung dar. Dabei erweitern die *Aspectual Feature Modules* ein Programm zum einen durch merkmalsorientierte Verfeinerungen und zum anderen durch aspektorientierte Mechanismen. Je nach Art des bestehenden Problems kann der Programmierer die passende Technik wählen und beide Techniken in geeigneter Weise kombinieren [Apel et al., 2005b]. So ist es nun beispielsweise auch möglich, wie in Abbildung 2.24 zu sehen ist, Aspekte zu verfeinern (*Aspect Refinement*) [Apel et al., 2007b].

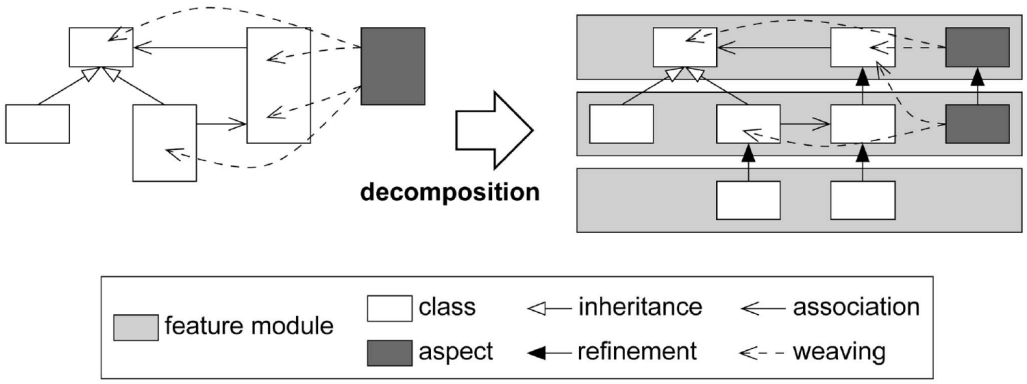

Abbildung 2.24: Eine merkmalsorientierte Dekomposition eines aspektorientierten Designs [Apel et al., 2008]

Basierend auf der merkmalsorientierten Implementierung des gewichteten Graphen aus Abbildung 2.22 zeigt Abbildung 2.25 eine zusätzliche Erweiterung durch das Merkmal *Color*, welches mit Hilfe eines Aspektes und einer Klasse implementiert und durch *Aspectual Feature Modules* abgekapselt wurde. Die Nutzung eines Aspektes an dieser Stelle ist vorteilhaft für die Vermeidung von Codewiederholungen und durch das Abkapseln des Aspektes und der Klasse *Color* wird der Zusammenhang der Bestandteile des Merkmals erreicht [Apel et al., 2008].

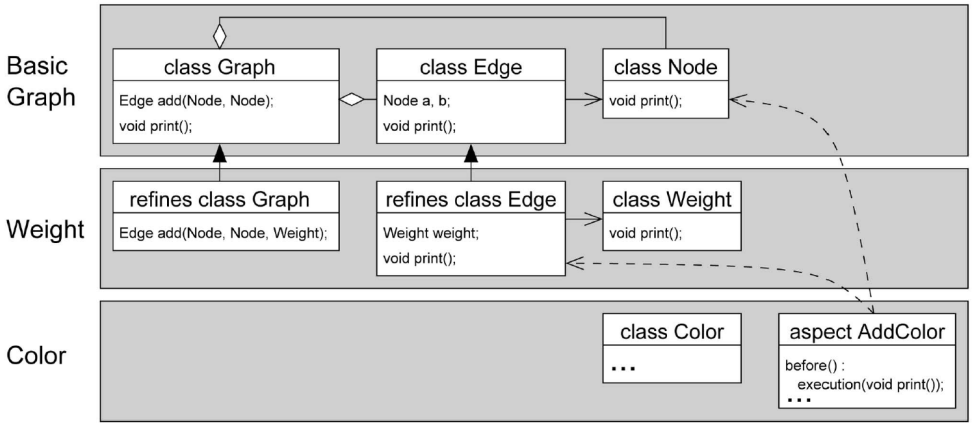

Abbildung 2.25: Erweiterung des gewichteten Graphen durch das Merkmal *Color* mit Hilfe von AFM [Apel et al., 2008]

Apel et al. zeigen, dass *Aspectual Feature Modules* die separat angewendete merkmalsorientierte und die aspektorientierte Programmierung übertreffen, indem ihre Stärken genutzt und ihre Schwächen umgangen werden [Apel et al., 2008]. Dabei bietet sich die Nutzung von Aspekten für homogene und dynamische querschneidende Belange an, wohingegen *Collaborations* und Verfeinerungen für heterogene querschneidende Belange und Funktionserweiterungen von Vorteil sind [Apel et al., 2005b, 2008].

2.5 Zusammenfassung

In den Grundlagen wurde aufgezeigt, wodurch sich *Cloud Computing* definiert, welche Besonderheiten *Cloud Computing* ausmachen und inwiefern sich die verschiedenen Dienst- und Verwendungsmodelle unterscheiden. Des Weiteren erfolgte eine Übersicht über die Vor- und Nachteile von *Cloud*-Diensten sowie über die Eigenheiten der unterschiedlichen *Cloud*-Anbieter, insbesondere dem *Amazon Web Service S3*.

Ebenso wurde auf den Aspekt der Transaktionsverwaltung eingegangen, indem sowohl die ACID Eigenschaften definiert, ihre Einschränkungen in Bezug auf das CAP-Theorem aufgezeigt und die Verfahren zur Synchronisationsverwaltung erläutert wurden. Dabei wurden sowohl die sperrenden als auch die nichtsperrenden Verfahren sowie die *Multiversion Concurrency Control* näher betrachtet.

Der dritte Abschnitt befasste sich anschließend mit den Erweiterungen der objektorientierten Programmierung, welche in Bezug auf die *Separation of Concerns* an ihre Grenzen stößt und durch die aspektorientierte und merkmalsorientierte Programmierung beziehungsweise durch deren Kombination, den *Aspectual Feature Modules* weiterentwickelt wurde.

Das weitere Vorgehen baut im Folgenden darauf auf, diese drei großen Fachgebiete miteinander zu verbinden und ein System zu entwickeln, durch welches Entwicklern eine konfigurierbare Transaktionsverwaltung zur Verfügung gestellt werden kann. Diese soll vor allem Entwickler unterstützen, die Anwendungen schreiben, welche mit *Cloud*-Speichersystemen kommunizieren und Mehrbenutzerzugriffe verwalten müssen.

Kapitel 3

3 Merkmalsanalyse der Transaktionsunterstützung für Cloud Computing

Um Anwendungen zu entwickeln, die den Bedürfnissen verschiedener Nutzergruppen genügen, gibt es verschiedene Ansätze. So ist es beispielsweise möglich, eine Anwendung für jede Nutzergruppe individuell zu entwickeln, was zu einem hohen Entwicklungsaufwand und einer geringen Wiederverwendbarkeit der Anwendung oder Anwendungsteilen führt. Eine Alternative ist die Entwicklung einer Standardanwendung mit einer breiten, aber nicht auf spezifische Nutzerkreise spezialisierten, Funktionsvielfalt, wodurch die Anwendung für einen größeren Anwendungsbereich nutzbar wird [Pohl et al., 2005]. Aufbauend auf diesen beiden Ansätzen, wurden Softwareproduktlinien entwickelt, um einen großen Nutzerkreis anzusprechen und gleichzeitig den Bedürfnissen von speziellen Anwendergruppen gerecht zu werden. Clements und Northrop definieren Softwareproduktlinien wie folgt:

> *„A software product line is a set of softwareintensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.“ [Clements und Northrop, 2001]*

Dabei bilden die Merkmale, welche alle Produkte gemeinsam haben, den Ausgangspunkt, zu dem nutzerkreisspezifische Merkmale je nach Anforderung hinzugewählt werden können. Um solche Softwareproduktlinien entwickeln zu können, muss die gesamte Zieldomäne[1] auf allgemeine Eigenschaften und Variabilitätsmerkmale untersucht und darauf aufbauend ein Merkmalsdiagramm modelliert werden [Czarnecki und Eisenecker, 1999].

Im Folgenden wird zum Zwecke der Domänenanalyse zunächst der Aufbau des Zielsystems beschrieben, wobei eine Drei-Schichten-Architektur[2] das Grundkonzept darstellt. Anschließend erfolgt eine detaillierte Merkmalsanalyse bezogen auf die benötigten Komponenten zur Transaktionsverwaltung und *Recovery*-Unterstützung sowie eine Aufschlüsselung und mögliche Integration verschiedener Synchonisationsverfahren.

[1] Eine Domäne ist ein Wissens- oder Aktivitätsbereich, welcher durch eine Menge von Konzepten und Begriffsdefinitionen charakterisiert wird [Bass et al., 2000].

[2] Eine Drei-Schichten-Architektur beschreibt die Strukturierung von Softwaresystemen in drei Schichten. Dazu gehören die Schicht für die graphische Oberfläche, die Schicht für die Anwendungslogik sowie die Schicht für die persistente Speicherung der Daten [Abts, 2010].

Drei-Schichten-Architektur

Die Entwicklung einer Transaktionsunterstützung für Anwendungen auf Basis von *Amazon S3* erfordert eine Übersicht der benötigten Komponenten. Hierfür geeignet ist eine Einteilung in eine Drei-Schichten-Architektur (siehe Abbildung 3.1), wobei *Amazon S3* die Schicht für die Datenhaltung darstellt und die Präsentationsschicht für die Clients durch den Endentwickler definiert wird.

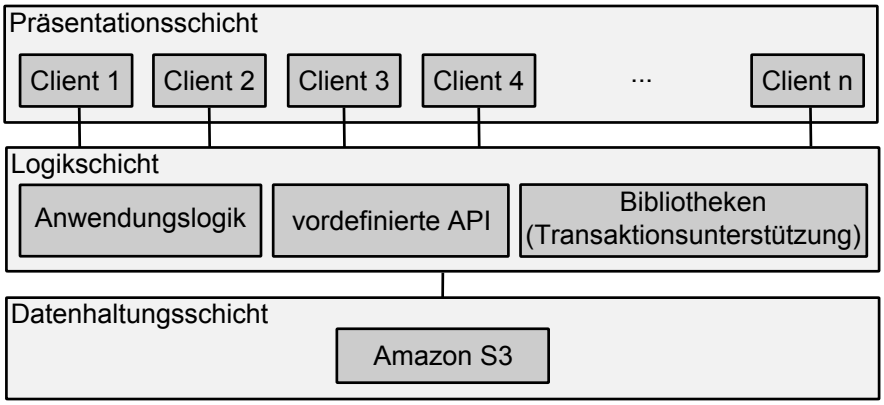

Abbildung 3.1: Aufbau auf Basis einer Drei-Schichten-Architektur

Zum Zwecke der Merkmalsanalyse wird jedoch insbesondere die Logikschicht beziehungsweise der Server, auf dem die eigentliche Anwendung ausgeführt werden soll und welche die Zugriffe durch die *Clients* koordiniert, betrachtet. So soll die Logikschicht über folgende Komponenten verfügen:

- **Anwendungslogik**

  Die Anwendungslogik wird vom Endentwickler geschrieben. Sie muss die eingehenden Anfragen der Clients koordinieren und Anfragen an den *S3*-Dienst stellen. Für die Unterstützung des Entwicklers soll es eine vordefinierte Programmierschnittstelle geben, so dass zum Beispiel *Update*, *Delete* und *Insert*-Befehle leichter aufzurufen sind und das Einweben der Transaktionskomponenten erleichtert werden kann.

- **Bibliotheken für die Transaktionslogik**

  Die Bibliotheken, welche die Transaktionslogik beinhalten, werden entsprechend in das Hauptprogramm eingebunden und sollen für die notwendige Integration dieser Logik in den Anwendungscode sorgen. Zu dieser Transaktionslogik gehören der Transaktionsmananger, *Recovery*-Mechanismen und Puffer-Möglichkeiten sowie die Implementierungen verschiedener Synchronisationsverfahren. Um Teile der Transaktionslogik in die Anwendungslogik integrieren zu können, müssen sowohl notwendige Verfeinerungen als auch Aspekte entsprechend in den Bibliotheken vorhanden sein.

3.1 Notation des Merkmalsdiagramms

Um die Transaktionslogik so umfassend wie möglich bezüglich ihrer Komponenten beziehungsweise ihrer Merkmale aufschlüsseln zu können, erfolgt in den folgenden Ab-

schnitten eine Analyse aller Merkmale sowie die schrittweise Erfassung dieser Merkmale in einem Merkmalsdiagramm, so dass eine Übersicht über die Merkmale und deren Abhängigkeiten entsteht. Zu diesem Zweck wird die Eltern-Kind-Notation der Beziehungen zwischen den Merkmalen, welche im Folgenden an einem Beispiel näher erläutert werden, von Kang et al. und dessen Erweiterungen von Czarnecki übernommen (siehe Abbildung 3.2) sowie um *cross-tree*-Beschränkungen (*Contraints*) ergänzt [Kang et al., 1990; Czarnecki, 1998; Batory, 2005].

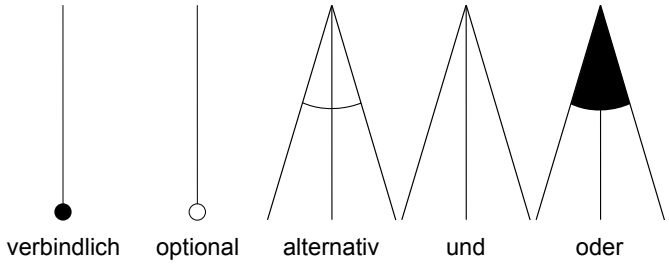

Abbildung 3.2: Beziehungsnotation im Merkmalsdiagramm nach [Batory, 2005]

Abbildung 3.3 zeigt ein Merkmalsdiagramm für die Zusammenstellung eines Kraftfahrzeugs. Durch die Und-Notation zwischen der Anzahl der Reifen, dem Kraftfahrzeugtyp und der möglichen Ausstattung wird angegeben, dass alle drei Merkmale gleichzeitig ausgewählt werden können, wenn ihr Elternmerkmal *Kraftfahrzeug* ausgewählt wurde. Dabei müssen die Merkmale *Anzahl Reifen* und *Typ* aufgrund ihrer Verbindlich-Notation aktiviert werden, das Merkmal *Ausstattung* hingegen kann aufgrund der Optional-Notation abgewählt werden. Hinsichtlich der Anzahl der Reifen muss die Wahl zwischen *vier*, *sechs* und *acht* erfolgen, da die Alternativ-Notation bedingt, dass nur eine Merkmal nur gleichen Zeit aktiv sein darf. Im Gegensatz dazu kann bei der Entscheidung zwischen den Merkmalen *Navigationsgerät*, *Ledersitze* und *Klimaanlage* mehr als eins beziehungsweise alle Merkmale ausgewählt werden. Bedingung dafür ist jedoch, dass das Merkmal *Ausstattung* aktiv ist und mindestens eins dieser drei Merkmale gewählt wird. Des Weiteren ist das Modell mit zwei *cross-tree*-Beschränkungen versehen. So bedingt das Merkmal *PKW*, dass das Kraftfahrzeug *vier* Reifen haben muss. Außerdem dürfen die Merkmale *LKW* und *Ledersitze* nicht gleichzeitig ausgewählt sein, was bedeutet, dass LKWs nicht mit Ledersitzen versehen werden dürfen.

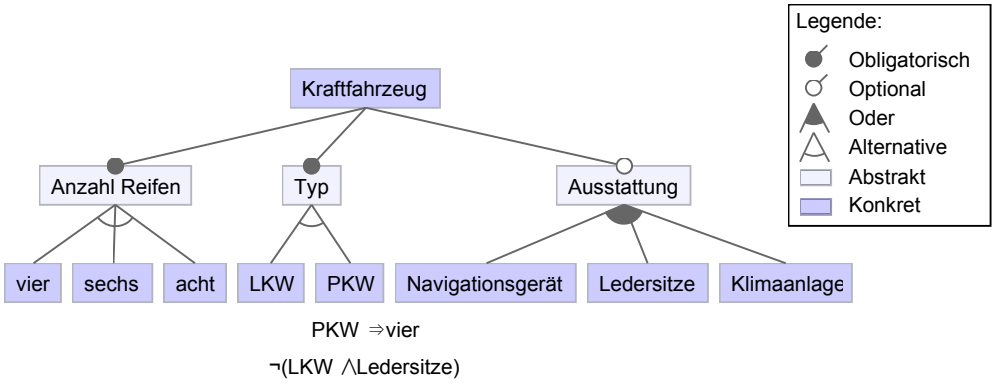

Abbildung 3.3: Beziehungsnotation am Beispiel: Kraftfahrzeug

3.2 Transaktionsmanager

Von großem Interesse für den Aufbau einer Transaktionsverwaltung für *Cloud*-Systeme ist die Aufschlüsselung der Komponenten, die ein Transaktionsmanager beinhalten muss (siehe Abbildung 3.4). So benötigt der Transaktionsmanager zum einen die Möglichkeit mehrere *Transaktionen* in einer *Transaktionsliste* zu verwalten und mit der Hilfe eines *Schedulers* eine Abarbeitungsreihenfolge der Transaktionen und ihrer Operationen zu ermitteln.

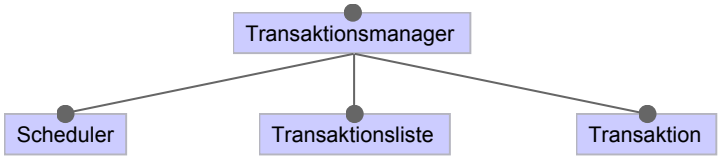

Abbildung 3.4: Ausschnitt Merkmalsdiagramm: Transaktionsmanager

3.2.1 Transaktion

Benötigt wird ein Merkmal *Transaktion*, das eine Liste der Operationen (*Trans-Op-Liste*) einer Transaktion sowie weitere relevante Informationen halten kann (siehe Abbildung 3.5). Dabei muss die Reihenfolge, in der die Operationen eintreffen, für jede Transaktion erhalten bleiben. Weiterhin ist für jede Operation die Information von Interesse, ob die jeweilige Operation schon ausgeführt wurde oder nicht, was dazu führt, dass für das Merkmal *Operation* eine Erweiterung um das Merkmal *Op-Zustand* eingeführt wird. Außerdem muss der Zustand der gesamten Transaktion (*Trans-Zustand*) vermerkt sein. Zu den möglichen Zuständen, die eine Transaktion einnehmen kann, gehören *Laufend*, *Verzögert*, *Committed*, *Abgebrochen* und *Gestoppt* [Saake et al., 2005].

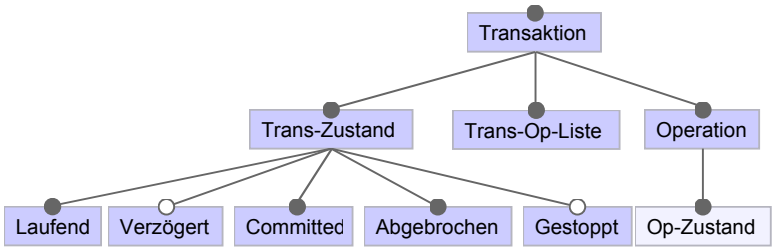

Abbildung 3.5: Ausschnitt Merkmalsdiagramm: Transaktion

- **Laufend**

  Eine Transaktion ist im Status *Laufend*, wenn ihre Operationen der Reihe nach ausgeführt werden.

- **Verzögert**

  Kann eine Operation einer Transaktion zum aktuellen Zeitpunkt nicht ausgeführt werden, weil diese Operation den fehlerfreien Ablauf stören würde, kann die Transaktion auf *Verzögert* gesetzt werden. Ausgehend von diesem Zustand ist es der Transaktion möglich, die Ausführung der Operation zu einem späteren

Zeitpunkt erneut zu starten. Dieser Zustand kann nicht bei jedem Synchronisationsverfahren auftreten, da einige Verfahren die Transaktion abbrechen, sobald ein Konflikt festgestellt wird. Aufgrund dessen ist dieses Merkmal optional.

- **Committed**

  Den Zustand *Committed* nimmt eine Transaktion dann an, wenn alle ihre Operationen erfolgreich ausgeführt wurden und das Ende der Transaktion erreicht wurde.

- **Abgebrochen**

  Eine Transaktion kann zum einen abgebrochen werden, weil eine ihrer Operationen abgewiesen wird, wenn diese die Serialisierbarkeit stört (*Durch-Fehler*) und zum anderen, wenn der Entwickler den Abbruchbefehl manuell auslöst (siehe Abbildung 3.6). Der Abbruch durch den Entwickler ist jedoch nur dann möglich, wenn das Merkmal *Stopp-durch-Nutzer-Abbruch* aktiviert wurde, da der manuelle Abbruchbefehl ein optionales Zusatzmerkmal darstellt. Der Abbruch durch einen Fehler ist jedoch notwendig und erlaubt zwei Folgeaktionen. So kann die Transaktion nach einem Abbruch in den Zustand *Gestoppt* versetzt und folglich nicht mehr ausgeführt werden (*Stopp-nach-Fehler*) oder aber neustarten (*Neustart-nach-Fehler*). Im Falle eines Neustarts muss des Weiteren angegeben werden, was nach wiederholtem Scheitern der Transaktion geschehen soll. So ist beispielsweise optional zuwählbar, dass die Transaktion nach einer bestimmten Anzahl von Ausführungsversuchen letztendlich doch in den Status *Gestoppt* versetzt wird (*Stopp-nach-zu-vielen-Versuchen*). Diese Wahlmöglichkeit besteht jedoch nicht, wenn der Abbruch durch den Entwickler manuell aufgerufen wird, da die Transaktion bei Wiederholungen immer wieder abgebrochen werden würde. Der wiederholte Abbruch wird dadurch ausgelöst, dass die auszuführenden Operationen, inklusive des Abbruchbefehls, gespeichert und somit identisch wiederholt ausgeführt werden. Aufgrund dieses Verhaltens wird die Transaktion nach dem manuellen Abbruch immer automatisch in den Zustand *Gestoppt* versetzt.

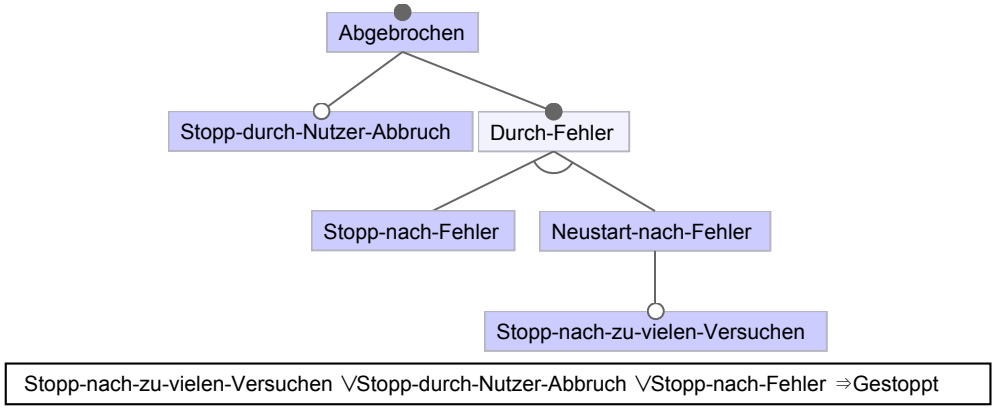

Abbildung 3.6: Ausschnitt Merkmalsdiagramm: Transaktionszustand *Abgebrochen*

- **Gestoppt**

  Eine abgebrochene Transaktion, welche nicht mehr neu gestartet wird, erhält den Status *Gestoppt*. Dieser Zustand ist optional, da er nur ausgewählt werden muss, wenn eines der drei Merkmale *Stopp-durch-Nutzer-Abbruch*, *Stopp-nach-Fehler* beziehungsweise *Stopp-nach-zu-vielen-Versuchen* ausgewählt wird.

Je nach Synchronisationsverfahren müssen dem Merkmal *Transaktion* weitere Informationen zugeführt werden, wie beispielsweise der Zeitstempel für diese Transaktion oder Informationen für bestimmte *Recovery*-Verfahren.

3.2.2 Kaskadierende Abbrüche und Wartegraph

Zur Feststellung der involvierten Transaktionen im Falle des Auftretens von *kaskadierenden Abbrüchen* wird eine Datenstruktur benötigt, die für jedes Objekt, auf welches zugegriffen wurde, festhält, welche Transaktion welche Operation auf diesem Objekt ausgeführt hat. Im Speziellen sind die Liest-von-Beziehungen für jedes Objekt von Interesse. Infolge eines Abbruchs einer Transaktion kann mit dieser Datenstruktur festgestellt werden, ob die, durch diese Transaktion geschriebenen Objekte, anschließend von anderen Transaktionen gelesen wurden. Ist dies der Fall müssten die lesenden Transaktionen folglich auch abgebrochen und die Datenobjekte neu geprüft werden (siehe Abbildung 3.7).

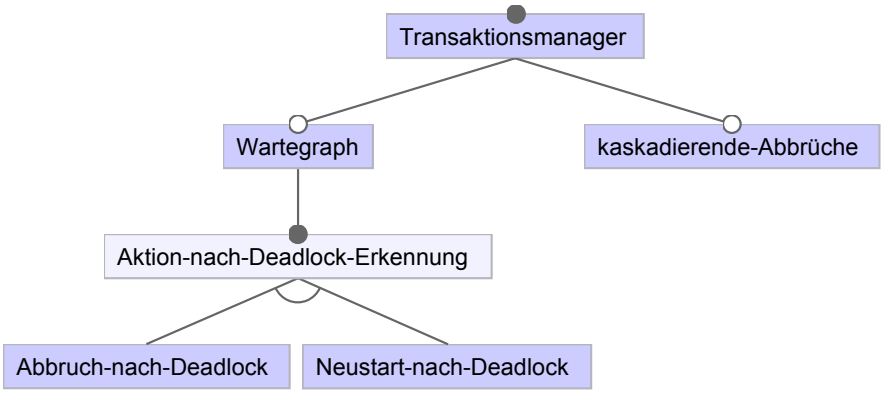

Abbildung 3.7: Ausschnitt Merkmalsdiagramm: Kaskadierende Abbrüche und *Deadlock*-Erkennung durch Wartegraph

Zusätzlich zu dem Merkmal für die kaskadierenden Abbrüche wird ein Merkmal für einen *Wartegraphen* benötigt. Seine Funktion besteht in dem Erfassen von Wartebeziehungen zwischen Transaktionen. Jedes Mal, wenn festgestellt wird, dass eine Transaktion auf eine andere warten muss, wird dies in den Graphen eingetragen und dieser auf Zyklen kontrolliert. So kann beim Auftreten eines Zyklus das Abbrechen einer oder mehrerer Transaktionen ausgelöst werden. Dadurch muss auch geklärt werden, ob diese anschließend neu gestartet (*Neustart-nach-Deadlock*) oder vollständig angehalten (*Abbruch-nach-Deadlock*) werden. Im Falle eines Abbruchs ist zu entscheiden, ob *alle Transaktionen* abgebrochen werden sollen oder nur jeweils *eine Transaktion*, die anhand eines *Auswahlkriteriums* ausgewählt wird. Mögliche Kriterien sind die *Anzahl der aufgebrochenen Zyklen*, die *Transaktionslänge* und der *Rücksetzaufwand* (siehe Abbildung 3.8). Dabei muss es möglich sein, mehr als ein Kriterium zu prüfen, für den

Fall, dass zum Beispiel mehrere Transaktionen dieselbe Anzahl an Zyklen aufbrechen oder sich nicht in der Transaktionslänge unterscheiden. Alternativ kann auch nur die Transaktion abgebrochen werden, die den Zyklus letztendlich ausgelöst hat.

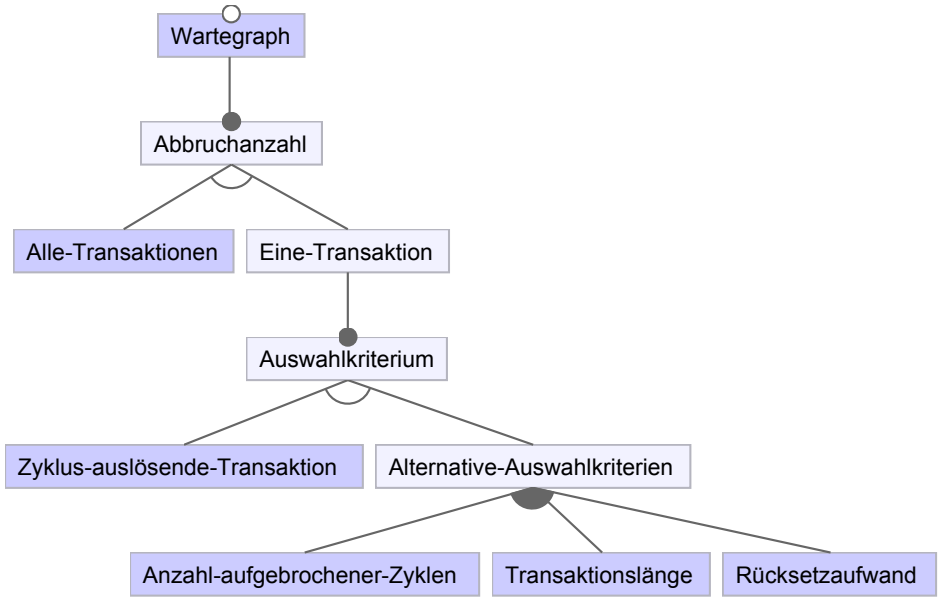

Abbildung 3.8: Ausschnitt Merkmalsdiagramm: *Deadlock*-Auflösung

Sowohl das Merkmal *Wartegraph* als auch das Merkmal *kaskadierende Abbrüche* sind optionale Erweiterungen des Transaktionsmanagers, da diese Merkmale nur für ausgewählte Synchronisationsverfahren notwendig sind. So schließen einige Synchronisationsverfahren zum Beispiel das Auftreten von kaskadierenden Abbrüchen beziehungsweise *Deadlocks*, für die der Wartegraph benötigt wird, aus.

3.2.3 Scheduler

Mit Hilfe des Merkmals *Scheduler* erfolgt die Auswahl, welches *Synchonisationsverfahren* bevorzugt wird. Die Grundunterscheidung (siehe Abbildung 3.9) besteht dabei zwischen den *Annotationsverfahren* und den *annotationsfreien Verfahren*. Zu den Annotationsverfahren zählen beispielsweise die *Sperrverfahren*, welche voraussetzen, dass die Datenobjekte mit Sperrinformationen versehen werden und die auf *Zeitmarken* basierenden Verfahren wie zum Beispiel die *Multiversion Concurrency Control* (*MVCC*) und das *Zeitmarkenverfahren*. Dahingegen sind die *serielle Abarbeitung*, der *Serialisierbarkeitsgraphentester* (*SGT*) sowie die *optimistische Synchronisation* zu den annotationsfreien Verfahren zu zählen.

Für den optionalen Transaktionszustand *Verzögert* ergibt sich bezüglich dieser Verfahren, dass nur die Sperrverfahren diesen Zustand erzeugen können, da eine Transaktion darauf warten kann, dass Sperren wieder freigegeben werden. Bei den annotationsfreien Verfahren und dem Zeitmarkenverfahren werden Transaktionen jedoch abgebrochen, sobald ein Konflikt festgestellt wird. Des Weiteren ergeben sich zusätzliche *cross-tree*-Beschränkungen, um sicherzustellen, dass keine zwei Verfahren gleichzeitig aktiv sind, wobei MVCC eine Ausnahme bildet, da sie nur für Leseoperationen gilt und ein weiteres Synchronisationsverfahren benötigt, um die Schreiboperationen zu koordinieren.

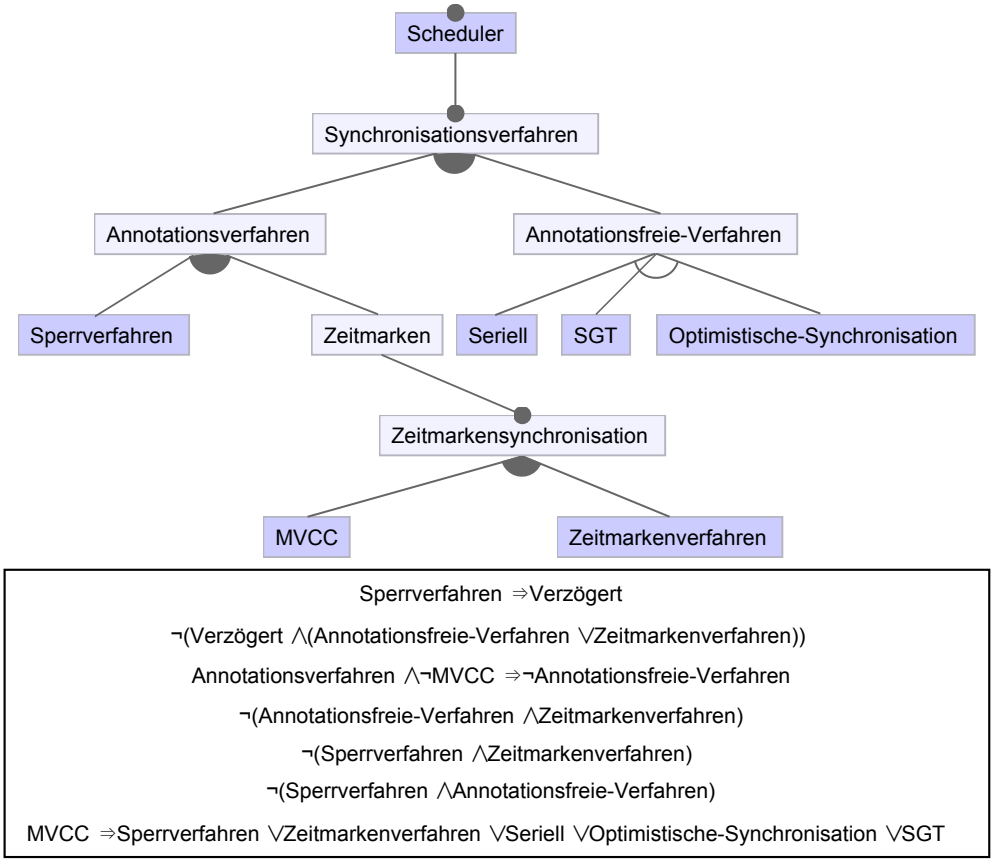

Abbildung 3.9: Ausschnitt Merkmalsdiagramm: Scheduler

- **Annotationsverfahren**

  Annotationsverfahren zeichnen sich dadurch aus, dass sie mit Markierungen arbeiten, an denen abgelesen werden kann, ob ein Objekt gelesen oder geändert werden darf. Die Sperrverfahren verwenden zu diesem Zweck Sperren auf den Objekten und die Zeitmarkenverfahren sowie die *Multiversion Concurrency Control* Zeitstempel.

  1. **Sperrverfahren**

     Zu den sperrenden Verfahren gehören das *Zwei-Phasen-Sperr-Protokoll* (*2-PL*) und seine Varianten. Mit der Hilfe der optionalen Merkmale *strikt* und *konservativ* können das strikte, das konservative beziehungsweise das konservative strikte Zwei-Phasen-Sperr-Protokoll erzeugt werden (siehe Abbildung 3.10). Beim strikten konservativen Zweiphasensperrprotokoll können keine *Deadlocks* oder kaskadierenden Abbrüche auftreten, wodurch der Wartegraph und die Abarbeitung von kaskadierenden Abbrüchen in einer ausschließenden Beziehung zu diesem Merkmal gesetzt werden. Dahingegen erfordert das strikte Zwei-Phasen-Sperr-Protokoll einen Wartegraphen zur *Deadlock*-Erkennung, das konservative Zwei-Phasen-Sperr-Protokoll die Behandlung von kaskadierenden Abbrüchen und das normale Zwei-Phasen-Sperr-Protokoll sogar beides. Als optionales Merkmal kann zu den Zwei-Phasen-Sperr-Protokollen das *multigranulare Sperren* (*MGL*) hinzugewählt

werden, wodurch intentionale Sperren auf den Ebenen, die über dem eigentlichen gesperrten Objekt liegen, gesetzt und abgefragt werden. Im Falle von *Amazon S3* sind die übergeordneten Objekte, welche mit intentionalen Sperren versehen werden müssten, die einzelnen *Buckets*.

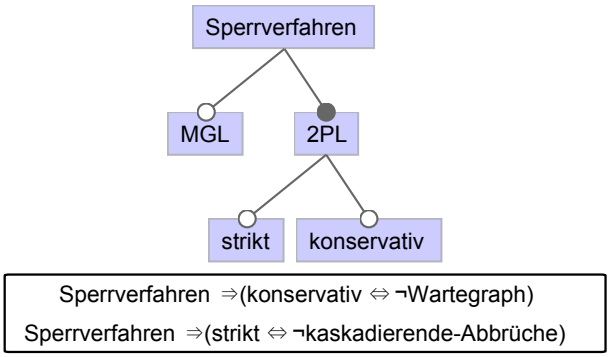

Abbildung 3.10: Ausschnitt Merkmalsdiagramm: Sperrverfahren

Das Sperren und Entsperren kann automatisch nach den Regeln des gewählten Sperrprotokolls erfolgen (*Auto-Sperren* und *Auto-Entsperren*) oder durch den Entwickler mit der Hilfe von Markierungen angegeben werden (*Manuell-Sperren* und *Manuelles-Entsperren*). Eine zusätzliche Wahlmöglichkeit bietet der Ort, an dem gesperrt werden soll (*Sperr-Ort*). So kann jedem Objekt die Sperrinformation als Metainformation mitgegeben (*Sperren-auf-S3-Objekte*) oder eine *lokale Sperrliste* geführt werden (siehe Abbildungen 3.11). Im Falle der manuell gesetzten Sperren ist ein Test erforderlich, der prüft, ob die Sperren dem jeweils aktiven Sperrprotokoll genügen (*Test-auf-Korrektheit*).

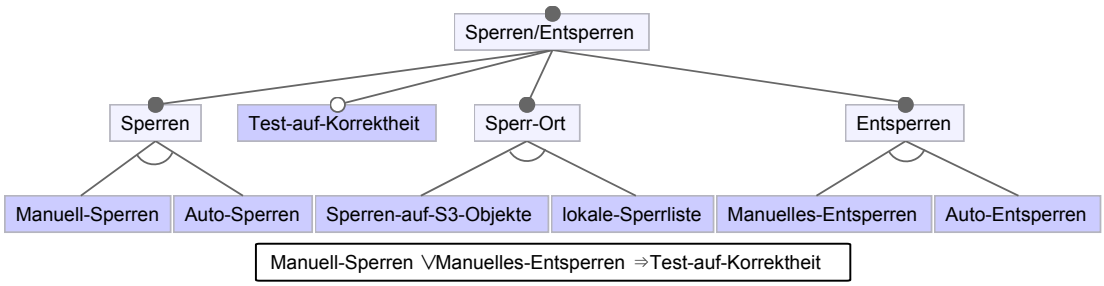

Abbildung 3.11: Ausschnitt Merkmalsdiagramm: Art und Überprüfung der Sperren und Freigaben

2. **Zeitmarkenverfahren**

   Durch das Zeitmarkenverfahren wird ein Algorithmus bereitgestellt, der mit der Hilfe von Zeitmarken ermittelt, wann eine Operation einer Transaktion ausgeführt werden darf und wann sie abgewiesen werden muss. Dieser Algorithmus benötigt zu diesem Zweck die Möglichkeit, den Datenspeicherobjekten Zeitstempel (*Zeitmarken-auf-Objekte*) zu geben. Zum einen muss nach jedem erfolgreichen Schreibvorgang der Schreibzeitstempel und nach jedem erfolgreichen Lesevorgang der Lesezeitstempel übergeben werden (*rts/wts*), die sich durch die Transaktion definieren, welche die Operation ausgeführt hat. Woraus folgt, dass auch jede Transaktion Zeitmarkeninforma-

tionen (*TransZeitmarke*), wie zum Beispiel eine *BOT-Zeitmarke*, benötigt (siehe Abbildung 3.12). Sowohl für die Transaktionszeitmarken als auch die Objektzeitmarken muss entschieden werden (*ts-Art*) und (*Trans-ts-Art*), ob sie als Zähler (*counter* und *counterZM*) oder als Echtzeitzeitmarken (*real* und *realZM*) generiert werden sollen. Damit die Zeitmarken ein einheitliches Format haben, werden die Zählervarianten und die Echtzeitzeitmarken mit der Hilfe von *cross-tree*-Beschränkungen und Alternativ-Notationen gegenseitig ausgeschlossen. Weitere *cross-tree*-Beschränkungen geben an, dass zusätzlich zum Zeitmarkenverfahren auch die unter den annotationsfreien Verfahren aufgeführte optimistische Synchronisation Transaktionszeitmarken benötigen. Im Gegensatz dazu besteht für die Transaktionszeitmarken bei der seriellen Abarbeitung, dem Serialisierbarkeitsgraphentester und den Sperrverfahren keine Notwendigkeit, solange nicht gleichzeitig die *Multiversion Concurrency Control* aktiv ist.

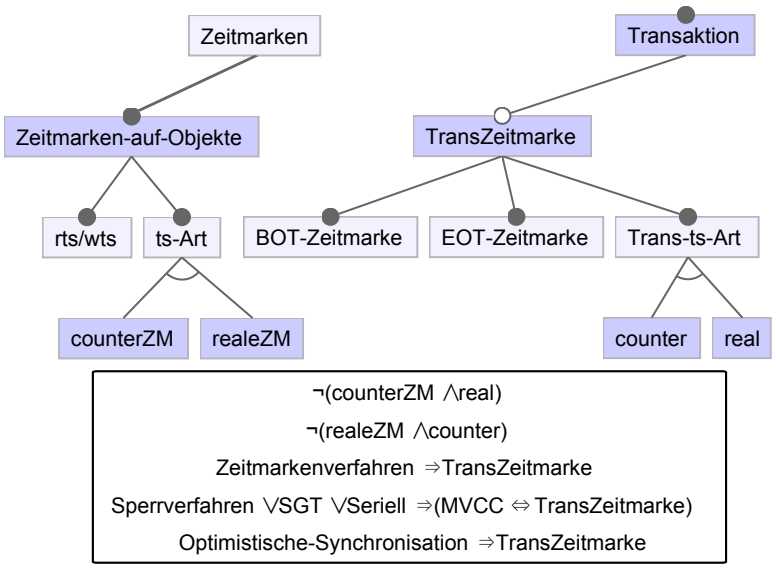

Abbildung 3.12: Ausschnitt Merkmalsdiagramm: Benötigte Elemente des Zeitmarkenverfahrens

Wie das Zwei-Phasen-Sperr-Protokoll kann auch das Zeitmarkenverfahren zur Vermeidung von kaskadierenden Abbrüchen modifiziert werden. So werden beim strikten Zeitmarkenverfahren (*striktZ*) nur Objekte gelesen, die von Transaktionen, die bereits ein *Commit* ausgeführt haben, geschrieben wurden (siehe Abbildung 3.13). Infolgedessen ergibt sich, dass unter Verwendung des Zeitmarkenverfahrens eine Erkennung und Behandlung von kaskadierenden Abbrüchen erforderlich ist, aber nicht unter Verwendung des strikten Zeitmarkenverfahrens. Einen Wartegraphen hingegen benötigt das Zeitmarkenverfahren nicht, da Transaktionen abgebrochen werden und nicht auf andere Transaktionen warten. Eine weitere optionale Modifikation des Zeitmarkenverfahrens erfolgt durch das Merkmal *optimiert*, wodurch der Algorithmus des Zeitmarkenverfahrens insofern geändert wird, dass der Algorithmus eine größere Menge an *Schedules* erlaubt, indem er für blindes Schreiben optimiert wird.

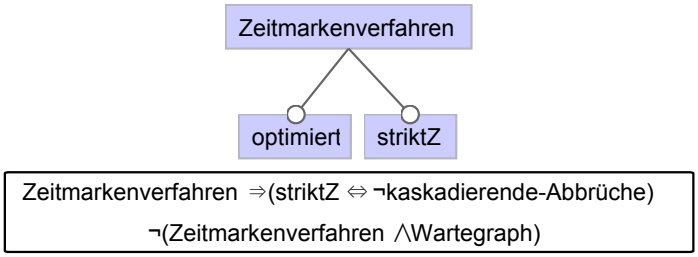

Abbildung 3.13: Ausschnitt Merkmalsdiagramm: Varianten des Zeitmarkenverfahrens

3. **Multiversion Concurrency Control**

   Die Aufgaben der *Multiversion Concurrency Control* sind das Erstellen und Verwalten von mehreren *Versionen* eines Datenobjektes (siehe Abbildung 3.14). Dabei ist es möglich, diese Versionen durch *Amazon S3* verwalten zu lassen (*S3-Versionen*), da *Amazon* dies als zusätzliche Funktion anbietet. Voraussetzung dafür ist, dass der Entwickler diese Funktion für *S3* aktiviert hat. Beachtet werden muss dabei, dass sich zum Beispiel der Speicherbedarf bezüglich des *S3*-Speichers erhöht. Von daher ist es angedacht, als Alternative eine lokale Speicherung der Versionen anzubieten (*lokale-Versionen*).

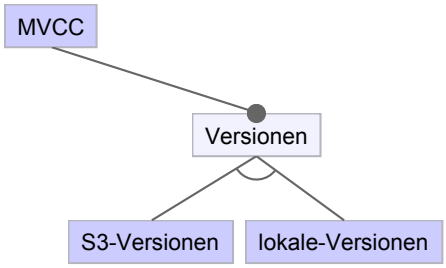

Abbildung 3.14: Ausschnitt Merkmalsdiagramm: Versionierung durch die *Multiversion Concurrency Control*

   Als optionales Merkmal ergibt sich das *Löschen der alten Versionen*, um den Speicherbedarf zu minimieren, da unter bestimmten Bedingungen ältere Versionen nicht mehr von Nutzen sind. Eine Möglichkeit ist es, diese *nach jeder beendeten Transaktion* zu löschen und eine andere, den Löschzeitpunkt dem Entwickler selbst zu überlassen (*Manuelles-Löschen*) (siehe Abbildung 3.15).

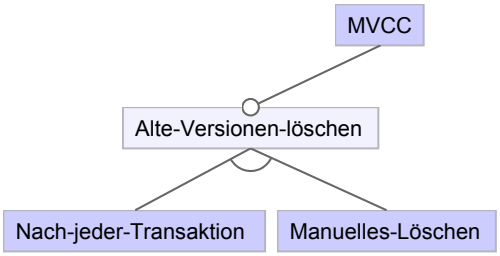

Abbildung 3.15: Ausschnitt Merkmalsdiagramm: Löschen alter Versionen der *Multiversion Concurrency Control*

- **Annotationsfreie Verfahren**

  Die annotationsfreien Verfahren synchronisieren Transaktionen ohne Markierungen wie Sperren oder Zeitmarken, aber dafür mit anderen Mechanismen. So benötigt zum Beispiel der Serialisierbarkeitsgraphentester einen Konfliktgraphen und die optimistische Synchronisation *Read-* und *Write-Sets* sowie Kopien der zu bearbeitenden Objekte. Eine weitere Eigenheit ist, dass annotationsfreie Verfahren keinen Wartegraphen benötigen, da sie wie das Zeitmarkenverfahren keine *Deadlocks* auslösen, sondern Transaktionen im Konfliktfall sofort abbrechen (siehe Abbildung 3.16).

Abbildung 3.16: Ausschnitt Merkmalsdiagramm: *Constraint* durch annotationsfreie Verfahren

1. **Serielle Abarbeitung**

   Bei der seriellen Abarbeitung wird die Transaktionsliste der Reihe nach abgearbeitet. Erst wenn alle Operationen einer Transaktion ausgeführt wurden und die Transaktion ein *Commit* ausgeführt hat, wird mit der nächsten Transaktion in der Liste fortgesetzt. Da bei einer seriellen Abarbeitung keine kaskadierenden Abbrüche auftreten können, wird bei diesem Verfahren das Merkmal zur Behandlung der kaskadierenden Abbrüche nicht benötigt, was im Merkmalsdiagramm durch eine *cross-tree*-Beschränkung aufgezeigt wird (siehe Abbildung 3.17).

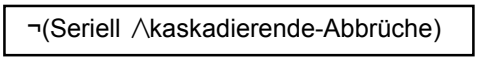

Abbildung 3.17: Ausschnitt Merkmalsdiagramm: *Constraint* durch seriellen Scheduler

2. **Serialisierbarkeitsgraphentester**

   Für die Umsetzung des Serialisierbarkeitsgraphentesters werden die Merkmale, die in Abbildung 3.18 zu sehen sind, benötigt. Zum einen muss es möglich sein, Knoten und Kanten zum Konfliktgraphen hinzuzufügen (*Knoten/Kanten-hinzufügen*), wenn ein Konflikt zweier transaktionsfremder Operationen erkannt wird. Zum anderen muss nach jedem Hinzufügen getestet werden, ob durch die neue Kante ein Kreis im Graphen verursacht wurde (*Test-auf-Zyklen*), was zum Abbruch der auslösenden Operation beziehungsweise Transaktion führt. Der Test auf Zyklen im Graphen kann durch verschiedene Algorithmen durchgeführt werden, wie beispielsweise durch eine

Tiefensuche[3] oder durch das *topologische Sortieren*[4]. Das dritte Merkmal betrifft das Entfernen von abgeschlossenen Transaktionen aus dem Konfliktgraphen (*Knoten/Kanten-entfernen*). So dürfen nur Knoten entfernt werden, die keine eingehenden Kanten haben und demnach keine Kreise mehr verursachen können. Auch dieses Verfahren benötigt das Merkmal zur Erkennung und Behandlung von kaskadierenden Abbrüchen.

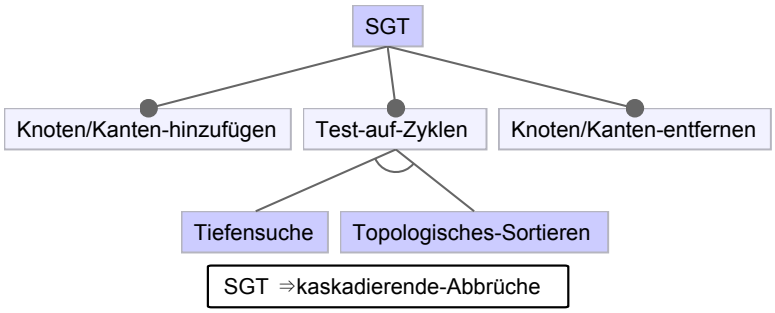

Abbildung 3.18: Ausschnitt Merkmalsdiagramm: Serialisierbarkeitsgraphentester

3. **Optimistische Synchronisation**

   Essentielle Merkmale der optimistischen Synchronisation sind das Arbeiten der Transaktionen auf, nur ihnen verfügbaren, lokalen Kopien der Datenobjekte und die Aufteilung der Abarbeitungsphasen der Transaktionen in *Lese-*, *Validierungs-* und *Schreibphase* (siehe Abbildung 3.19). Jede Transaktion muss somit wissen, in welcher Phase sie sich befindet und gleichzeitig muss gewährleistet werden, dass immer nur eine Transaktion zur selben Zeit in der Validierungsphase ist. Im Speziellen wird eine unteilbare Validierungsschreibphase bevorzugt, wodurch beide Phasen in diesem kritischen Abschnitt ausgeführt werden. Da zwei Möglichkeiten existieren, eine Transaktion zu validieren, ergibt sich zusätzlich die Auswahl zwischen der *Rückwärts-* und der *Vorwärtsvalidierung*, wobei auch beide Validierungsmöglichkeiten zusammen ausgeführt werden können. Da bei der optimistischen Synchronisation auf Datenkopien gearbeitet wird und am Ende jeder Transaktion eine Validierung stattfindet, können kaskadierende Abbrüche nicht auftreten, was bedeutet, dass dieses Merkmal bei Verwendung der optimistischen Synchronisation ausgeschlossen werden sollte.

[3] Bei der Tiefensuche werden die Knoten eines gerichteten Graphen der Reihe nach abgearbeitet, wobei ausgehend von einem Startknoten ein Pfad weiter in die Tiefe abgesucht wird, bis ein Ende erreicht wurde oder nur bereits besuchte Knoten folgen würden. Von diesem Punkt wird durch schrittweises Zurückgehen im Graphen jeder noch nicht besuchte Teilbaum betrachtet, bis alle Knoten abgearbeitet wurden oder der gesuchte Knoten gefunden wurde [Cormen et al., 2001]. Für die Zyklensuche kann der Basisalgorithmus erweitert werden, so dass ein Zyklus genau dann gefunden wurde, wenn der Nachfolgeknoten ein bereits besuchter, aber noch nicht vollständig abgearbeiteter Knoten ist.

[4] Mit der Hilfe der topologischen Sortierung kann ein gerichteter Graph in eine lineare Ordnung seiner Knoten überführt werden. Zu diesem Zweck werden solange Knoten ohne eingehende Kanten aus dem Graphen entfernt, bis dieser leer ist. Dies ist jedoch nur möglich, wenn der Graph keinen Kreis beinhaltet [Cormen et al., 2001].

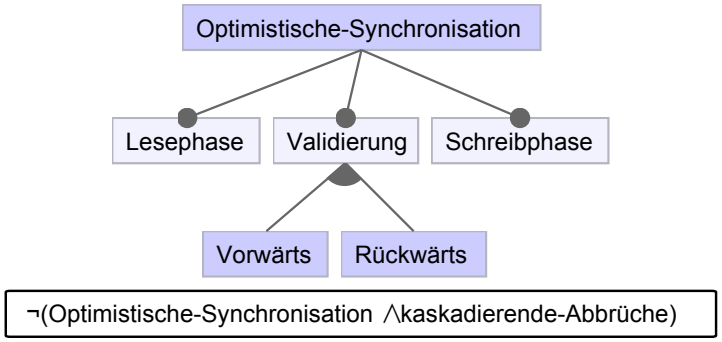

Abbildung 3.19: Ausschnitt Merkmalsdiagramm: Optimistische Synchronisation

Ein weiteres Merkmal, welches die optimistische Synchronisation benötigt, ist der *Transaktionspuffer*, der dem *Puffermanager* zugeordnet wird. Zu diesem Transaktionspuffer gehören die Merkmale *Read-Set*, *Write-Set* und *Objektkopien* (siehe Abbildung 3.20). Das *Read-* und das *Write-Set* sind insbesondere für den Abgleich der Transaktionen in der Validierungsphase von Nutzen und die Objektkopien stellen die Kopien der Datenobjekte da, auf denen die Transaktionen lokal arbeiten.

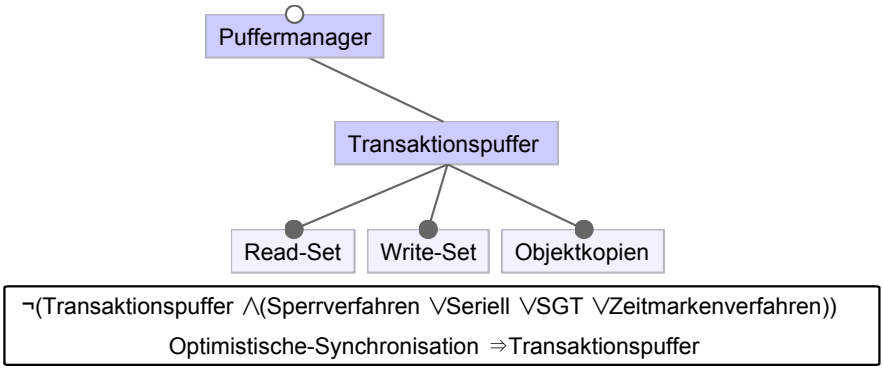

Abbildung 3.20: Ausschnitt Merkmalsdiagramm: Transaktionspuffer

Da diese Merkmale nicht für Annotationsverfahren, den Serialisierbarkeitsgraphentester und die serielle Abarbeitung erforderlich sind, sollten sie für diese ausgeschlossen und für die optimistische Synchronisation bedingt werden.

3.2.4 Transaktionsmodell

Zusätzlich zu *einfachen* Transaktionen, welche den ACID-Eigenschaften genügen, können auch die *erweiterten Transaktionsmodelle* ausgewählt werden (siehe Abbildung 3.21), bei denen diese Eigenschaften aufgeweicht sind. Im Gegensatz zu einfachen Transaktionen eignen sie sich vor allem für langlebige Transaktionen. Unter Verwendung der einfachen Transaktionen kann es zum Beispiel oft vorkommen, dass diese langlebigen Transaktionen abgebrochen werden müssen, die Ausführung von kurzen Transaktionen lange blockiert wird oder häufig *Deadlocks* ausgelöst werden [Gray et al., 1981a; Elmagarmid, 1992]. Zu den erweiterten Transaktionsmodellen gehören zum einen die *ge-*

schachtelten Transaktionen und zum anderen die *entkoppelten Subtransaktionen*. Wenn erweiterte Transaktionsmodelle ausgewählt werden, ist es notwendig, sich für mindestens eins dieser beiden Modelle zu entscheiden, wobei es auch möglich ist, geschachtelte Transaktionen und entkoppelte Subtransaktionen zusammen zu verwenden. Im Gegensatz dazu muss eine Ausprägung der geschachtelten Transaktionen gewählt werden, da es nur sinnvoll ist, entweder *offen* geschachtelte, *geschlossen* geschachtelte, *flexible* Transaktionen oder *Sagas* einzusetzen. Die flexiblen Transaktionen stellen dabei schon eine Kombination aus offen und geschlossen geschachtelten Transaktionen dar. Da bei der Verwendung der seriellen Abarbeitung von Transaktionen keine Vorteile in Sachen Parallelität durch die geschachtelten Transaktionen entstehen, können sich diese beiden Merkmale infolgedessen gegenseitig ausschließen.

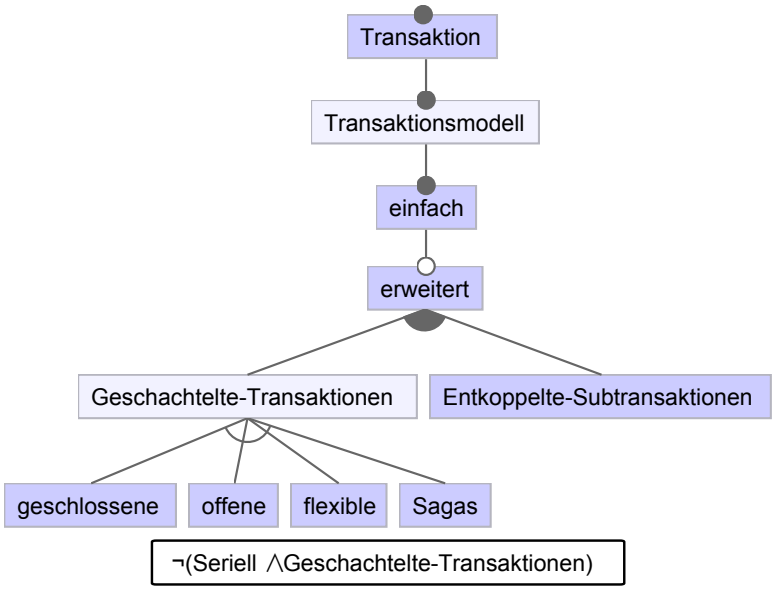

Abbildung 3.21: Ausschnitt Merkmalsdiagramm: Transaktionsmodelle

- **Geschachtelte Transaktionen**

  Mit der Hilfe von geschachtelten Transaktionen werden komplexe und vor allem lange Transaktionen in Subtransaktionen, die einzelne Arbeitsschritte der Transaktionen darstellen, aufgeteilt. Diese können wiederum in Subtransaktionen gegliedert werden, wodurch eine Baumstruktur aus der Wurzeltransaktion und ihren Subtransaktionen entsteht. [Moss, 1981] Die Synchronisation von Geschwisterknoten, welche denselben Elternknoten haben, kann bei geschachtelten Transaktionen wie bei vollkommen separaten Transaktionen erfolgen [Elmagarmid, 1992].

  1. **Geschlossen geschachtelt**

     Bei der Variante der geschlossen geschachtelten Transaktionen gilt, dass die Effekte durch ein *Commit* der Subtransaktionen nur für dessen Elternknoten sichtbar sind, bis die Wurzeltransaktion ein *Commit* ausgeführt hat, wodurch die Effekte auch für andere Wurzeltransaktionen sichtbar werden [Agrawal et al., 2006].

2. **Offen geschachtelt**

   Im Gegensatz zu geschlossen geschachtelten Transaktionen erlauben offen geschachtelte Transaktionen, dass Zwischenergebnisse der Subtransaktionen nach außen hin sichtbar werden dürfen [Elmagarmid, 1992; Agrawal et al., 2006]. Für das Merkmalsdiagramm (siehe Abbildung 3.22) ergeben sich für offen geschachtelte Transaktionen zusätzliche Besonderheiten. So kann es vitale und nicht-vitale Subtransaktionen geben, wobei die nicht-vitalen Subtransaktionen, wenn sie abbrechen, ignoriert werden können (*ignoriere-nicht-vitale*). Um aber nicht-vitale Subtransaktionen ignorieren zu können, muss das Merkmal *nicht-vitale Subtransaktionen* mit Hilfe einer *cross-tree*-Beschränkung ebenfalls ausgewählt werden, damit es nicht-vitale Subtransaktionen gibt.

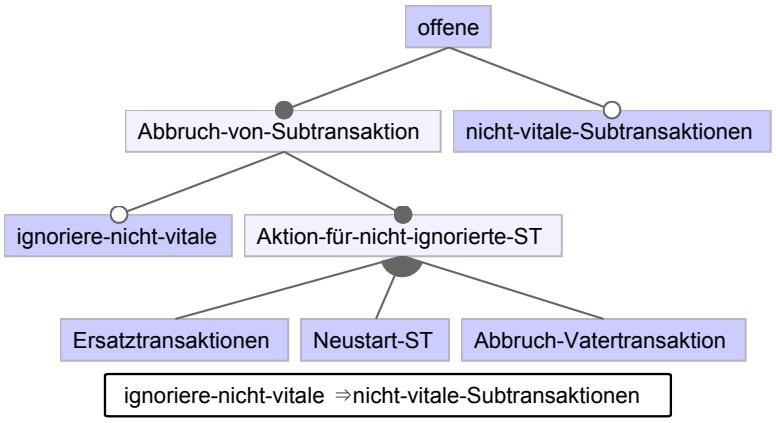

Abbildung 3.22: Ausschnitt Merkmalsdiagramm: Abbrüche von Subtransaktionen bei offen geschachtelten Transaktionen

   Für die *vitalen* Subtransaktionen gibt es des Weiteren alternative Reaktionen auf ihren Abbruch. So kann beispielsweise eine *Ersatztransaktion* durch den Entwickler aufgerufen, die betreffende Subtransaktion wiederholt ausgeführt (*retry-ST*) oder die *Vatertransaktion abgebrochen* werden [Elmagarmid, 1992].

3. **Flexibel**

   Die flexiblen Transaktionen stellen eine Kombination aus offen und geschlossen geschachtelten Transaktionen dar, indem zwischen kompensierbaren und nichtkompensierbaren Subtransaktionen unterschieden wird. Subtransaktionen, die kompensiert werden können, dürfen ein *Commit* ausführen, bevor die Wurzeltransaktion ein *Commit* ausgeführt hat, da ihre Effekte rückgängig gemacht werden können. Im Gegensatz dazu müssen nichtkompensierbare Subtransaktionen auf das *Commit* der gesamten Transaktion warten [Elmagarmid et al., 1990].

4. **Sagas**

   Sagas sind eine weitere Spezialform von geschachtelten Transaktionen. Ihre wichtigste Eigenschaft besteht darin, dass jede Subtransaktion zusammen mit ihrer kompensierenden Transaktion angegeben wird und die Saga einer bestimmten Schachtelungsregelung unterliegt. So bestehen Sagas immer aus

einer Wurzeltransaktion und nur einer weiteren Ebene an einfachen Subtransaktionen [Garcia-Molina und Salem, 1987].

- **Entkoppelte Subtransaktionen**

  Durch entkoppelte Subtransaktionen können innerhalb von Transaktionen eigenständige Wurzeltransaktionen aufgerufen werden, welche wiederum einen Transaktionsbaum mit Subtransaktionen darstellen und außerhalb des Geltungsbereiches der aufrufenden Transaktion laufen. Sie werden entweder innerhalb einer Transaktion aktiviert oder in Abhängigkeit von einem *Abort* oder *Commit* gestartet [Dayal et al., 1991].

3.2.5 Kompensierende Transaktionen

Kompensierende Transaktionen können zum einen durch die Nutzung der im Log gesicherten *undo*-Informationen ausgeführt (*Durch-Log*) oder durch den Entwickler angegeben werden (*Durch-Nutzer*) (siehe Abbildung 3.23). Im letzteren Fall muss der Entwickler für jede Transaktion angeben, welche Transaktion als kompensierend aufgerufen werden soll, wenn die ursprüngliche Transaktion abbricht. Sagas beispielsweise setzen voraus, dass die kompensierenden Transaktionen für jede Subtransaktion vom Entwickler festgelegt werden, da das Zurücksetzen der Subtransaktionen nicht durch den *Recoverymanager* ausgeführt wird [Saake et al., 2005].

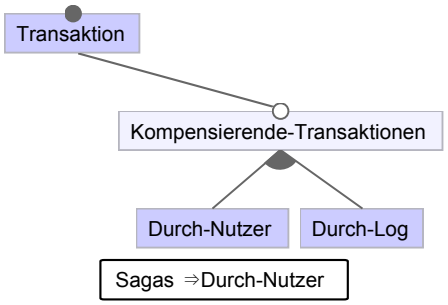

Abbildung 3.23: Ausschnitt Merkmalsdiagramm: Kompensierende Transaktionen

Das Merkmal für die kompensierenden Transaktionen ist optional, da zum Beispiel die optimistische Synchronisation auf eigenen Kopien arbeitet und demgemäß Änderungen durch die Transaktionen erst nach dem *Commit* beziehungsweise nach einer Validierung in den persistenten Speicher gelangen, was ein Zurücksetzen der Änderungen durch diese Transaktionen nicht mehr bedingt (siehe Abbildung 3.24). Verfahren jedoch, die direkt auf dem persistenten Speicher beziehungsweise auf einem gemeinsamen Datenbestand arbeiten, benötigen die Möglichkeit der Kompensation ihrer Änderungen, wenn Transaktionen abgebrochen werden. Auch im Falle der Nutzung von offen geschachtelten und flexiblen Transaktionen werden kompensierende Transaktionen für die optimistische Synchronisation wieder erforderlich.

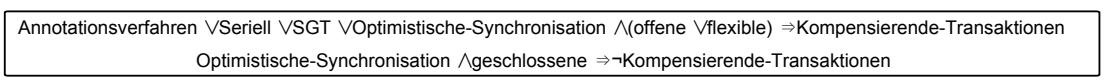

Abbildung 3.24: Ausschnitt Merkmalsdiagramm: *Constraints* für die kompensierenden Transaktionen

3.3 Speichermanager

Neben dem Transaktionsmanager benötigt eine Transaktionsunterstützung die Möglichkeit Daten im Fehlerfall oder nach einem Ausfall des Systems wieder herzustellen und Objekte zwischen dem persistenten und dem flüchtigen Speicher zu transferieren. Zu diesem Zweck werden ein Puffer- und ein *Recoverymanager*, die teilweise zusammenarbeiten, angeboten (siehe Abbildung 3.25).

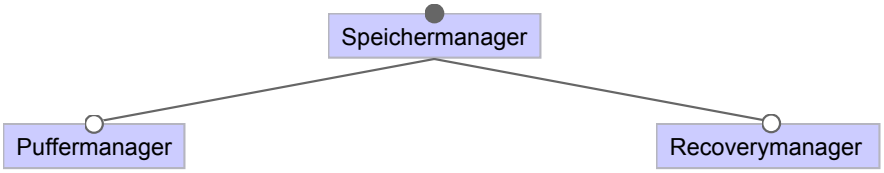

Abbildung 3.25: Ausschnitt Merkmalsdiagramm: Speichermanager

3.3.1 Puffermanager

Der Puffermanager besteht aus mehreren Komponenten, welche abhängig von den zu puffernden Daten sind. So können beispielsweise Daten aus dem persistenten Speicher in den Puffer geladen werden, so dass alle Transaktionen auf diesem gepufferten Datenbestand arbeiten können (*DB-Puffer*). Außerdem benötigt das Log, wenn es verwendet wird, eine Möglichkeit, kurzfristig noch nicht geschriebene Logeinträge in einem separaten Puffer zu halten (*Log-Puffer*). Für die optimistische Synchronisation wird sogar, wie bereits erläutert, jeder Transaktion eine isolierte Kopie der von dieser Transaktion bearbeiteten Daten zugeordnet (*Transaktionspuffer*) (siehe Abbildung 3.26).

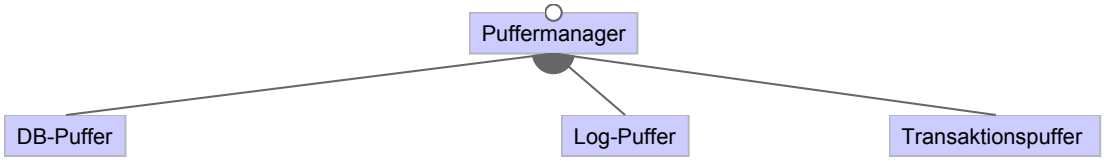

Abbildung 3.26: Ausschnitt Merkmalsdiagramm: Puffermanager

- **Datenpuffer**

  Für den gepufferten Datenbestand werden Strategien benötigt, welche angeben, wann Daten spätestens aus dem Puffer in den persistenten Speicher zurückgeschrieben werden müssen. Außerdem ist es wichtig zu entscheiden, was geschehen soll, wenn der Puffer voll ist, aber neue Daten aus dem persistenten Speicher in den Puffer gelesen werden müssen und wie das Einbringen der Änderungen in den persistenten Speicher umgesetzt wird (siehe Abbildung 3.27). Zu diesem Zweck gibt es die Propagierungsstrategien *REDO* und *NO-REDO*, die Seitenersetzungsstrategien *UNDO* und *NO-UNDO* sowie die Einbringstrategien *direkte* und *indirekte Zuordnung*.

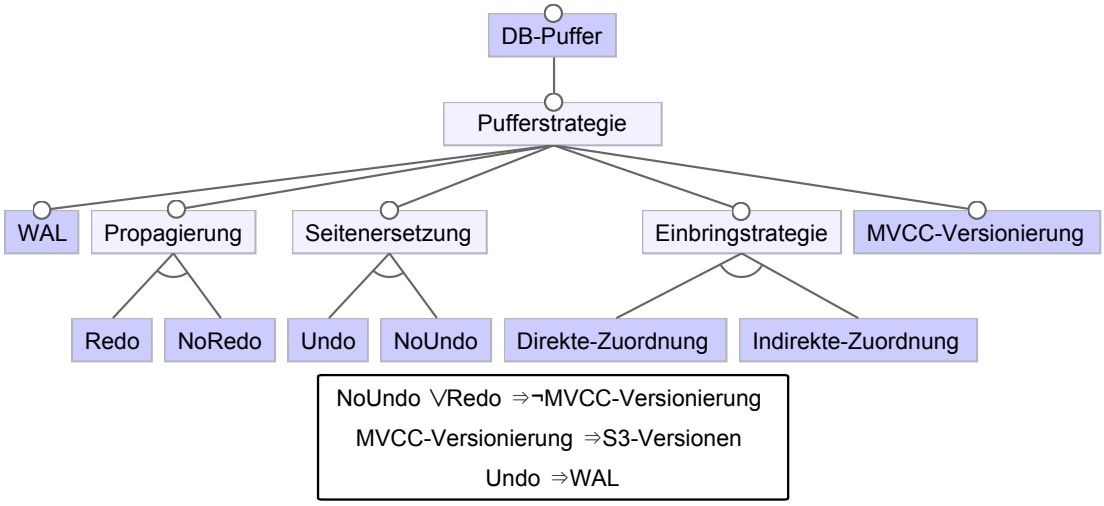

Abbildung 3.27: Ausschnitt Merkmalsdiagramm: Datenpuffer

Erweitert werden können diese Strategien mit dem *WAL-Prinzip*, welches festlegt, wann Log-Einträge spätestens ausgeschrieben werden müssen, was aber auch das Führen eines Logs voraussetzt. Benötigt wird das WAL-Prinzip vor allem unter Verwendung des UNDO, da so sichergestellt werden kann, dass im Falle eines Fehlers alle Log-Einträge für das Zurücksetzen rechtzeitig im persistenten Speicher sind. Ein zusätzlicher Punkt wird für die UNDO/NO-REDO-Konfiguration angeboten, da im Falle der Verwendung der MVCC deren Objektkopien als mögliche *Before-Images* genutzt werden können (*MVCC-Versionierung*). Voraussetzung ist, dass diese Kopien nicht lokal angelegt wurden, sondern auf dem persistenten Speicher (*S3-Versionen*).

- **Log-Puffer**

  Wird ein Log geführt, ist es wichtig, dass es möglichst umgehend in den persistenten Speicher geschrieben wird. Aus diesem Grund wird ein separater *Log-Puffer* angelegt, der zum Beispiel durch eine *Warteschlange* oder einen *Ringpuffer* umgesetzt werden kann und kontinuierlich ausgelesen werden muss (siehe Abbildung 3.28).

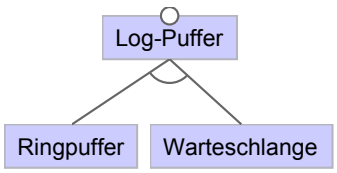

Abbildung 3.28: Ausschnitt Merkmalsdiagramm: Log-Puffer

3.3.2 Recoverymanager

Mittelpunkt des *Recoverymanagers* bilden die *Recovery-Verfahren*, zu denen das *Schattenspeicherverfahren*, *ARIES* sowie die *Recovery*-Strategien *UNDO/REDO*, *UNDO/NO-REDO*, *NO-UNDO/REDO* und *NO-UNDO/NO-REDO* gehören und von denen

genau eins aktiviert werden muss, sobald das Merkmal Pufferstrategie ausgewählt wurde. Außer dem Schattenspeicherverfahren benötigen diese Verfahren auch noch ein *Log* und teilweise auch *Sicherungspunkte* (siehe Abbildung 3.29).

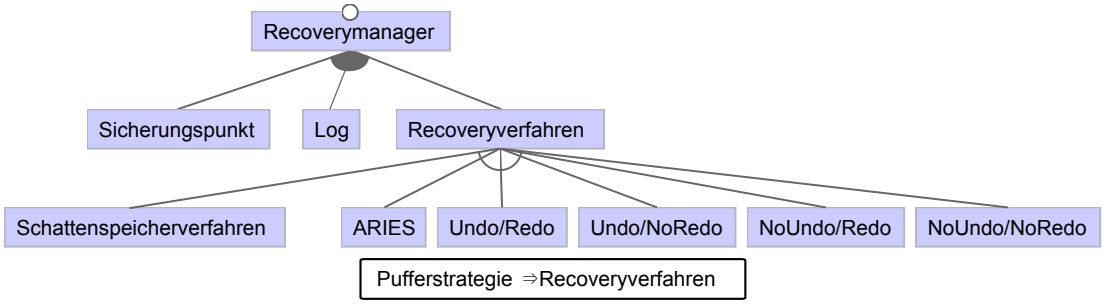

Abbildung 3.29: Ausschnitt Merkmalsdiagramm: *Recoverymanager*

- **Log**

  Ein *Log*-Eintrag hat eine fest definierte Struktur. So besteht er aus einer eindeutigen ID (*LSN*), der Kennung der Transaktion, die für diesen Log-Eintrag verantwortlich ist (*TransID*) sowie einem Verweis auf das Objekt, das bearbeitet wurde (*PageID*). Dazu kommen die *undo* und *redo*-Informationen (*Redo/Undo*), wenn sie benötigt werden und ein Hinweis auf den vorhergehenden, von dieser Transaktion ausgelösten Log-Eintrag (*PrevLSN*). Für die Verweise auf den Beginn (*BOT*) und den Abschluss (*commit/abort*) einer Transaktion beziehungsweise auf einen angelegten Sicherungspunkt (*Checkpoint*) werden spezielle, auf diese Markierungen angepasste Einträge angelegt (siehe Abbildung 3.30). Für den Fall, dass es nicht notwendig oder gewünscht ist, das Log immer größer werden zu lassen, wird auch eine Option angeboten, alte, nicht mehr benötigte Logeinträge zu löschen (*Alte-Log-Einträge-löschen*). Dies könnten zum Beispiel Einträge sein, die vor Sicherungspunkten liegen. Zusätzlich zu der *cross-tree*-Beschränkung, dass das Log einen Log-Puffer bedingt, muss das Log genau dann aktiviert werden, wenn das WAL-Prinzip ausgewählt wurde.

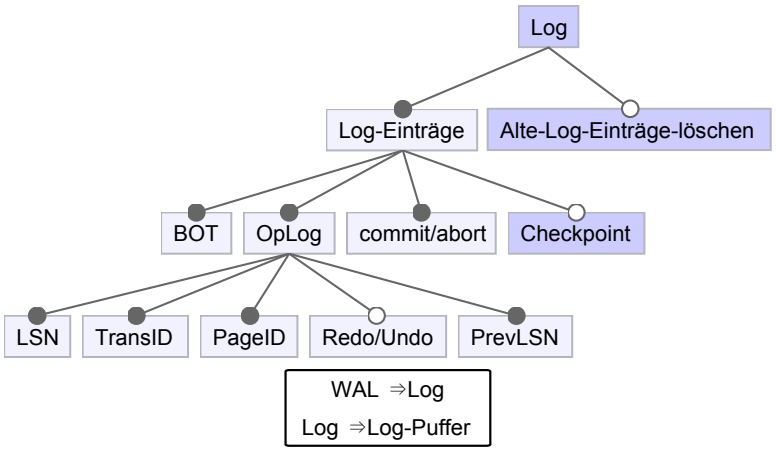

Abbildung 3.30: Ausschnitt Merkmalsdiagramm: Log

Ob *undo*- beziehungsweise *redo*-Informationen angelegt werden und welcher Art diese sind, hängt von dem verwendeten *Recovery*-Verfahren ab. So gibt es zum einen die physische Protokollierung mit *After*- und *Before-Images* und zum anderen die logische Protokollierung mit dem Festhalten der ausgeführten Operationen und ihren Kompensationen (*Redo-Info* und *Undo-Info*) (siehe Abbildung 3.31). Die direkte Zuordnung zum Beispiel benötigt physische Protokollinformationen, sofern das ARIES-Verfahren nicht gleichzeitig in Verwendung ist. Allgemein gilt, dass, abhängig von der gewählten Propagierungs- beziehungsweise Seitenersetzungsstrategie, auch die jeweiligen *redo*- beziehungsweise *undo*-Log-Informationen benötigt werden. Eine weitere *cross-tree*-Beschränkung ergibt sich durch die kompensierenden Transaktionen, da diese ein Undo-Log benötigen, wenn sie mit der Hilfe eines Logs ausgeführt werden sollen.

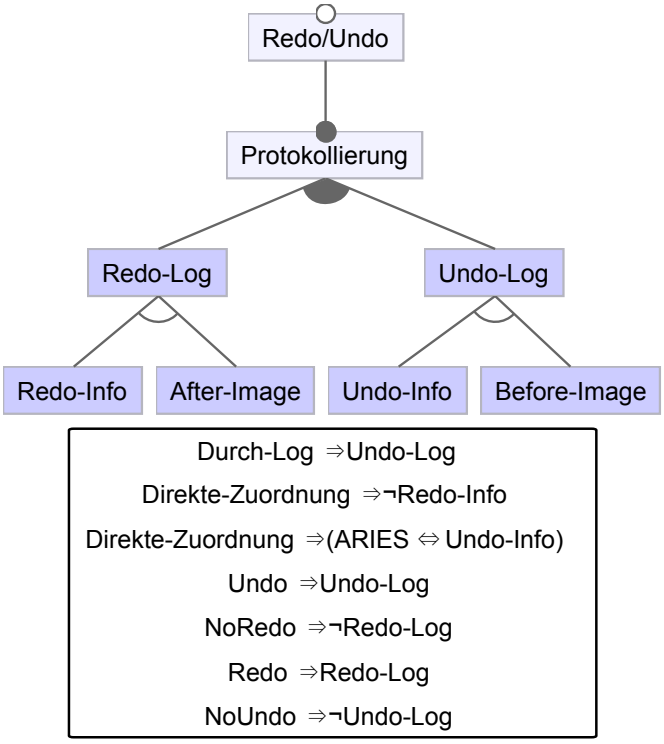

Abbildung 3.31: Ausschnitt Merkmalsdiagramm: *redo*- und *undo*-Informationen

- **Sicherungspunkte**

  Werden *Sicherheitspunkte* benötigt, lässt sich zwischen drei verschiedenen Ansätzen wählen. So können entweder *transaktionskonsistente*, *aktionskonsistente* oder *unscharfe* Sicherungspunkte angelegt werden (siehe Abbildung 3.32). Gleichzeitig bedingt die Nutzung von Sicherungspunkten einen Eintrag im Log und umgekehrt.

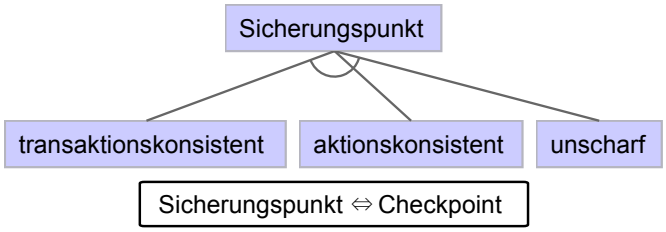

Abbildung 3.32: Ausschnitt Merkmalsdiagramm: Sicherungspunkte

- **Recovery-Verfahren**

  Zu den *Recovery*-Verfahren gehören UNDO/REDO, UNDO/NO-REDO, NO-UNDO/REDO und NO-UNDO/NO-REDO sowie das ARIES-Verfahren, welche alle ein Log benötigen und das Schattenspeicherverfahren, welches zusätzlich oder anstatt eines Logs eingesetzt werden kann. Je nach Verfahren werden bestimmte Merkmale vorausgesetzt oder ausgeschlossen.

  1. **UNDO/REDO, UNDO/NO-REDO, NO-UNDO/REDO und NO-UNDO/NO-REDO**

     Zusätzlich zu den jeweilig benötigten Seitenersetzungs- und Propagierungsstrategien, die sich äquivalent zum Namen der Verfahren ergeben, müssen diese Verfahren um weitere Eigenschaften und Merkmale ergänzt werden (siehe Abbildung 3.33). So sollte zum UNDO/REDO-Verfahren die direkte Zuordnung gewählt werden sowie ein Verfahren zum Setzen von Sicherungspunkten. Das UNDO/NO-REDO-Verfahren hingegen verlangt nach jedem *Commit* das Setzen eines transitionskonsistenten Sicherungspunktes. Dafür lässt sich das UNDO/NO-REDO-Verfahren jedoch mit der MVCC-Versionierung kombinieren, um deren Kopien als *Before-Images* verwenden zu können. Da die Einbringstrategie ein optionales Merkmal ist, muss es beim NO-UNDO/REDO-Verfahren hinzugewählt werden, wobei die Wahl zwischen der direkten und der indirekten Zuordnung erhalten bleibt. Im Gegensatz dazu muss beim NO-UNDO/NO-REDO-Verfahren die indirekte Zuordnung gewählt werden, um Änderungen beim *Commit* atomar in den persistenten Speicher überführen zu können. Durch die indirekte Zuordnung können Änderungen zunächst auf Kopien ausgeführt werden und bei Bedarf zu Originalen umgeschaltet werden [Saake et al., 2005].

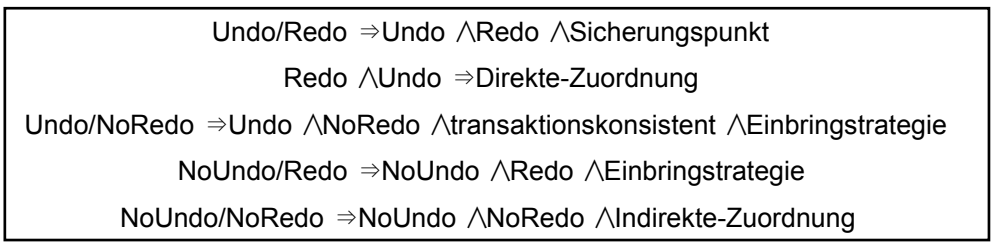

Abbildung 3.33: Ausschnitt Merkmalsdiagramm: *Constraints* der *Recovery*-Strategien

2. **ARIES**

   Das ARIES-Verfahren verwendet das WAL-Prinzip, unscharfe Sicherungspunkte und die UNDO- und REDO-Pufferstrategien, wobei es jedoch sowohl physische *redo*- als auch logische *undo*-Log-Informationen kombiniert (siehe Abbildung 3.34). Die überdies notwendigen Datenstrukturen beziehungsweise erforderlichen Referenzen sind die *Dirty-Pages-Liste*, der Verweis auf den Logeintrag der letzten Änderung eines Datensatzes (*pageLSN*) und der Verweis auf den Logeintrag, welcher angelegt wurde, als der Datensatz in den *dirty*-Zustand versetzt wurde (*recoveryLSN*). Die *Dirty-Page*-Liste und der Verweis auf den Logeintrag der letzten Änderung einer Transaktion (*lastLSN*) müssen entsprechend auch für ARIES bedingt beziehungsweise andernfalls ausgeschlossen werden [Mohan et al., 1992].

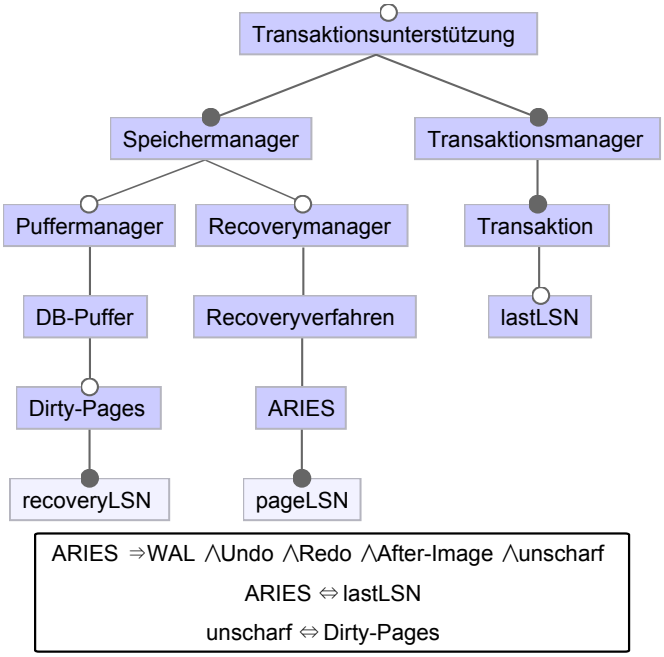

Abbildung 3.34: Ausschnitt Merkmalsdiagramm: ARIES

3. **Schattenspeicherverfahren**

   Essentieller Bestandteil des Schattenspeicherverfahrens sind die virtuellen *Seitentabellen* zum Umschalten zwischen den Originaldaten und der Kopie. Erweitert werden kann dieses Verfahren dadurch, dass eine zu starke Zerstreuung der Daten vermieden wird. Dieses *Twin-Block-Schattenspeicher*-Verfahren speichert infolgedessen die Arbeits- und die Schattenkopie immer in benachbarten Blöcken, was zum Beispiel mit Hilfe eines *Buckets* für die Kopie und eines *Buckets* für das Original in Bezug auf *Amazon S3* umgesetzt werden könnte, und verwendet statt der Seitentabellen *Bit-Listen* (siehe Abbildung 3.35).

   Weitere *cross-tree*-Beschränkungen ergeben sich, da für das Schattenspeicherverfahren weder Propagierungs- noch Seitenersetzungsstrategien benötigt werden. Auch die Einbringstrategien fallen unter der Voraussetzung, dass nicht das *Twin-Block*-Verfahren ausgewählt wurde, weg.

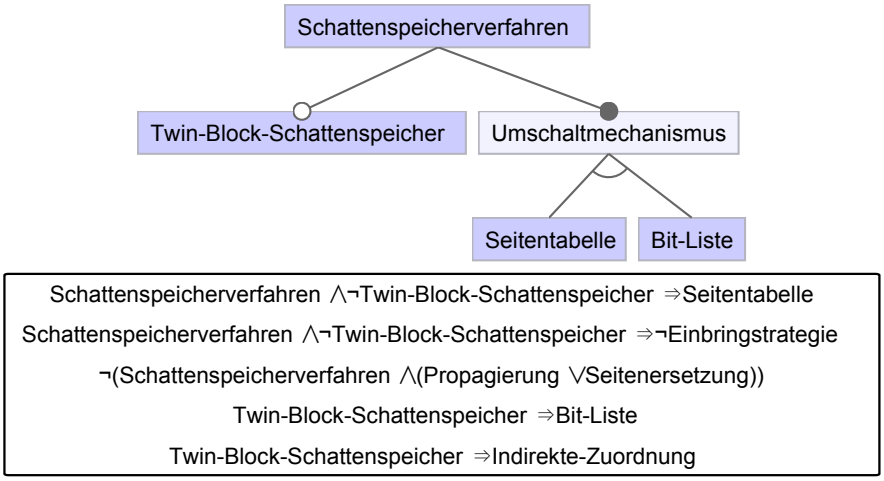

Abbildung 3.35: Ausschnitt Merkmalsdiagramm: Schattenspeicherverfahren

3.4 Zusammenfassung

In diesem Kapitel wurde zunächst ein Einblick in die Architektur des Zielsystems gegeben, welches durch eine Drei-Schichten-Architektur mit *Amazon S3* als Datenhaltungsschicht umgesetzt werden soll. Besonderer Fokus wurde auf die Logikschicht gelegt, welche sowohl die Anwendungslogik als auch die vordefinierten Funktionen zur Kommunikation mit *S3* enthalten soll und durch Bibliotheken zur Transaktionsunterstützung erweitert werden muss. Um diese Bibliotheken umsetzen zu können, wurde ein Merkmalsdiagramm entworfen, welches die notwendigen und variablen Merkmale der Domäne visualisiert.

Zu den notwendigen Merkmalen gehören der Transaktionsmanager und der Speichermanager, wobei der Transaktionsmanager zur Umsetzung Transaktionen benötigt, welche durch ihre Operationen und Zustände definiert und um Zusatzmerkmale wie zum Beispiel Transaktionszeitmarken erweitert werden können. Des Weiteren muss ein Synchronisationsverfahren ausgewählt werden, mit dem der Transaktionsmanager nebenläufige Transaktionen, die auf demselben Speicherbereich arbeiten, koordinieren kann, um einen fehlerfreien Ablauf zu gewährleisten. Zu den möglichen Synchronisationsverfahren gehören zum Beispiel die Sperrverfahren, die Zeitmarkenverfahren, die serielle Abarbeitung von Operationen, der Serialisierbarkeitsgraphentester und die optimistische Synchronisation. Erweitert werden können diese Verfahren zusätzlich noch durch die *Multiversion Concurreny Control* zur Koordination von Lesezugriffen. Um Probleme wie *Deadlocks* oder kaskadierende Abbrüche behandeln zu können, wurden außerdem Merkmale integriert, die zur Umsetzung von Hilfsstrukturen wie dem Wartegraphen dienen. Da es nicht nur das einfache Transaktionsmodell gibt, sondern auch erweiterte Transaktionsmodelle wie die geschachtelten Transaktionen und die entkoppelten Subtransaktionen, wurden auch für diese Transaktionsmodelle Merkmale im Merkmalsdiagramm aufgenommen.

Hinsichtlich des Speichermanagers wurden Merkmale für die Umsetzung des Puffermanagers und des *Recoverymanagers* entworfen. So beinhaltet der Puffermanager den Datenpuffer, den Log-Puffer und den Transaktionspuffer, wobei der Datenpuffer die Propagierungs-, Einbring- und Seitenersetzungsstrategien umsetzen muss und durch das WAL-Prinzip und die MVCC-Versionierung erweitert werden kann. Der Transak-

tionspuffer hingegen ist nur für die Hilfsstrukturen der optimitischen Synchronisation wie den Objektkopien, den *Read-* und *Write-Sets* zuständig und der Log-Puffer für das Übergeben von Log-Informationen in den persistenten Speicher. Die Merkmale für das Log, die *Recovery*-Verfahren und Sicherungspunkte befinden sich hingegen im Speichermanager. Dieser koordiniert und garantiert den fehlerfreien Transfer von Daten zwischen dem Puffer und dem persistenten Speicher und gewährleistet eine Wiederherstellung der Daten nach einem Transaktions- beziehungsweise Systemfehler. Zu diesem Zweck bietet er eine Wahl zwischen verschiedenen *Recovery*-Verfahren wie dem Schattenspeicherverfahren, ARIES und den Verfahren, die sich aus den Kombinationen der Seitenersetzungs- und Propagierungsstrategien ergeben. Ergänzt wird das Merkmalsdiagramm dabei durch eine Liste von *cross-tree*-Beschränkungen, die Merkmale wie Zeitmarken, Sperren oder Hilfsstrukturen, je nach aktiviertem Verfahren, ausschließen oder bedingen.

Aufbauend auf diesem Merkmalsdiagramm, welches vollständig in Abbildung 3.37, Abbildung 3.38 und Abbildung 3.39 zu sehen ist, soll im folgenden Kapitel ein Kollaborationsdiagramm entworfen werden, welches die benötigten Aspekte und Verfeinerungen der Merkmale visualisiert und angibt, welche Klassen und Funktionen durch die Aspekte und Verfeinerungen verändert beziehungsweise erweitert werden müssen. In beiden Modellen wird dabei die Farblegende aus Abbildung 3.36 verwendet, um durch die farbliche Gruppierung der Merkmale den Zusammenhang der einzelnen Merkmale zu unterstreichen und die Verbindung zwischen den beiden Modellen zu verdeutlichen.

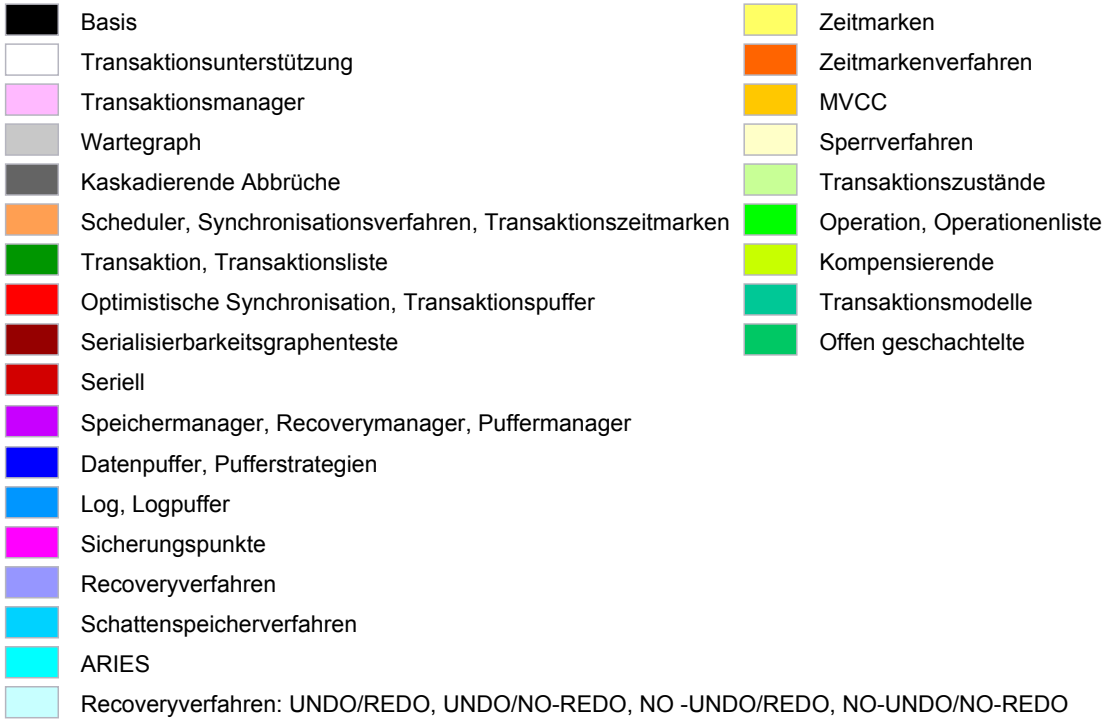

Abbildung 3.36: Vollständiges Merkmalsdiagramm: Farblegende

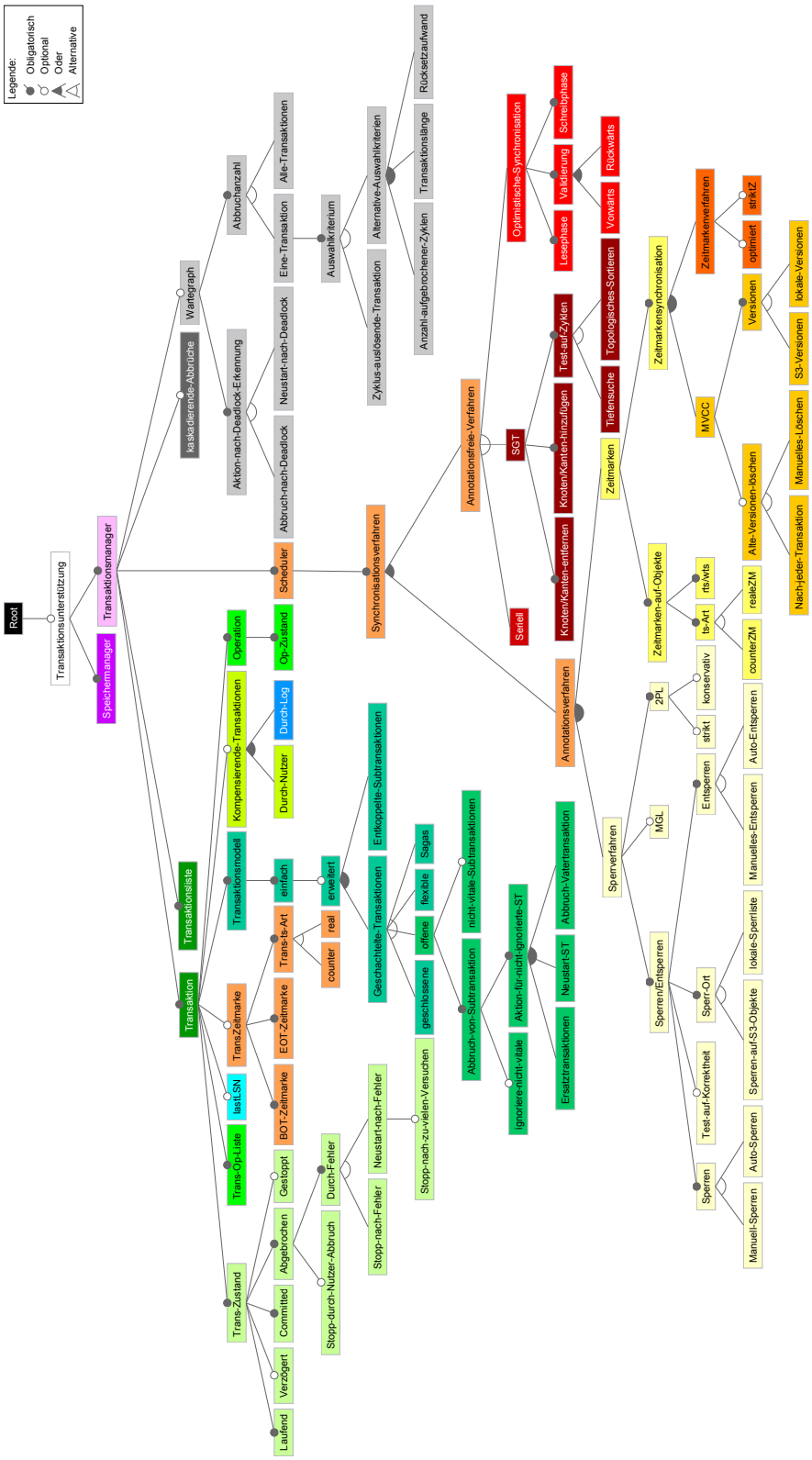

Abbildung 3.37: Vollständiges Merkmalsdiagramm: Transaktionsmanager

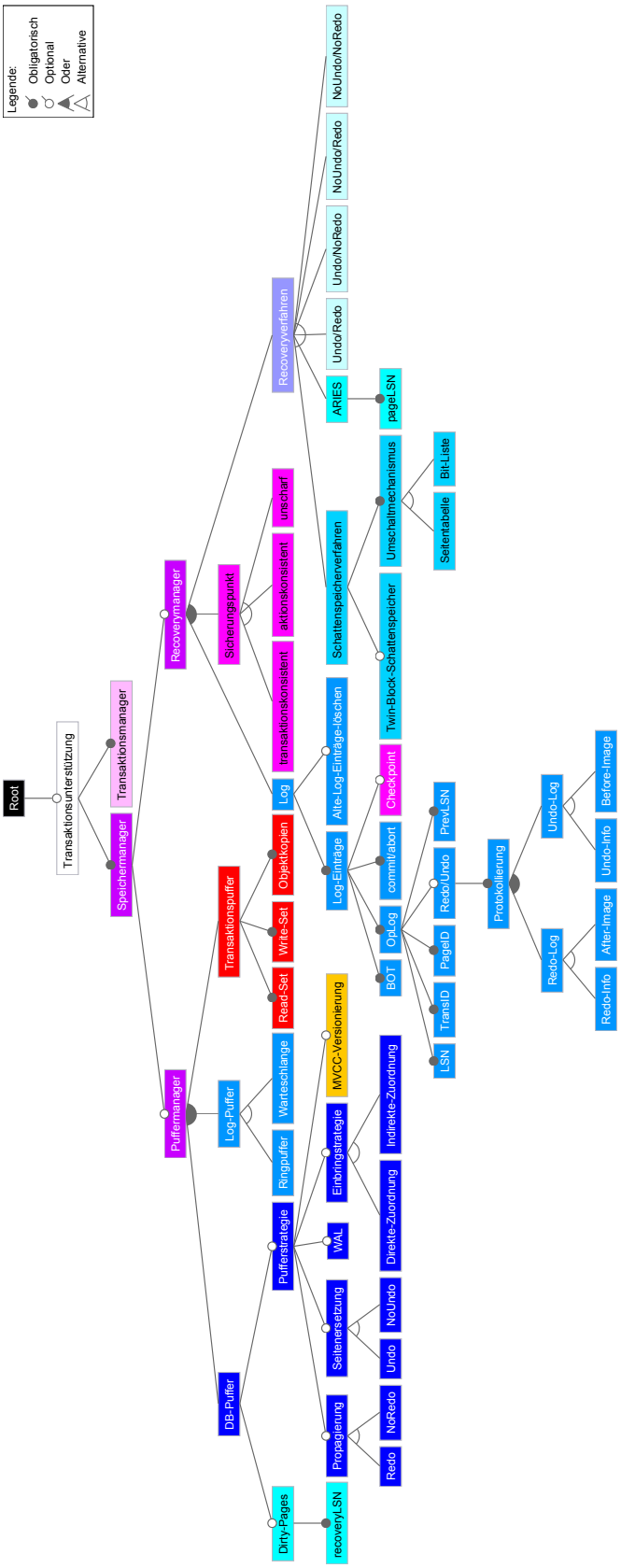

Abbildung 3.38: Vollständiges Merkmalsdiagramm: Speichermanager

Abbildung 3.39: Vollständiges Merkmalsdiagramm: *Constraints*

Kapitel 4

4 Merkmalsbasierter Entwurf anhand von Aspekten und Verfeinerungen

In diesem Kapitel wird ein Entwurf für eine mögliche Implementierung der konfigurierbaren Transaktionsunterstützung in Form eines Kollaborationsdiagramms vorgestellt. Mit Hilfe des Merkmalsdiagramms aus dem vorhergehenden Kapitel, welches die Variationsmöglichkeiten der Transaktionsunterstützung definiert, lassen sich zunächst die Kernmerkmale und optionalen Merkmale bestimmen, so dass zu diesen die benötigten Klassen, Aspekte und Verfeinerungen ermittelt werden können. Dabei werden Schichten im Kollaborationsdiagramm nur für die konkreten Merkmale des Merkmalsdiagramms erstellt, da die abstrakten Merkmale nicht umgesetzt werden. Der Aufbau des Kollaborationsdiagramms orientiert sich des Weiteren am *Collaborationstack* der merkmalsorientierten Programmierung (siehe Abschnitt 2.4.2), wobei eine Schicht alle Klassenerweiterungen und neuen Klassen beinhaltet, durch die sich ein bestimmtes Merkmal definiert. Da jede Schicht neben Verfeinerungen auch Aspekte enthalten kann, wurde das Kollaborationsdiagramm um die Möglichkeit, Aspekte zu visualisieren, erweitert.

4.1 Notation des Kollaborationsdiagramms

Abbildung 4.1 zeigt die verwendete Notation des Kollaborationsdiagramms zur Unterscheidung von Basisklassen, Kern- und Variationsmerkmalen sowie zum Hervorheben von neuen Klassen, Verfeinerungen und Aspekten am Beispiel eines Kraftfahrzeugs. Karosserie, Fahrwerk und Elektronik stellen beispielsweise Basisklassen dar, welche unabhängig von aktivierten Merkmalen ein Basiskraftfahrzeug definieren. Erst durch das Hinzufügen der Kernmerkmale *Motor* und *Motorkontrolle* beziehungsweise des variablen Merkmals *Erweiterte Kontrolle* werden neue Klassen, Aspekte und Verfeinerungen zum Basismodell hinzugefügt. Die beiden Kernmerkmale werden dabei immer aktiviert, sobald eine Erweiterung des Basismodells gewünscht wird. Dahingegen kann das variable Merkmal *Erweiterte Kontrolle* beliebig aktiviert beziehungsweise deaktiviert werden. Durch das Kernmerkmal *Motor* wird eine neue Klasse hinzugefügt, die den Motor definiert. Durch das Kernmerkmal *Motorkontrolle*, welches automatisch auch aktiv wird, sobald das Merkmal *Motor* aktiviert wird, wird die Klasse Motor zum Beispiel um ein Gütesiegel erweitert und ein Aspekt für eine Kontrolle des Motors hinzugefügt. Das variable Merkmal *Erweiterte Kontrolle* verfeinert die Kontrolle des Motors um zusätzliche Kontrollkriterien und den Motor um ein erweitertes Gütesiegel.

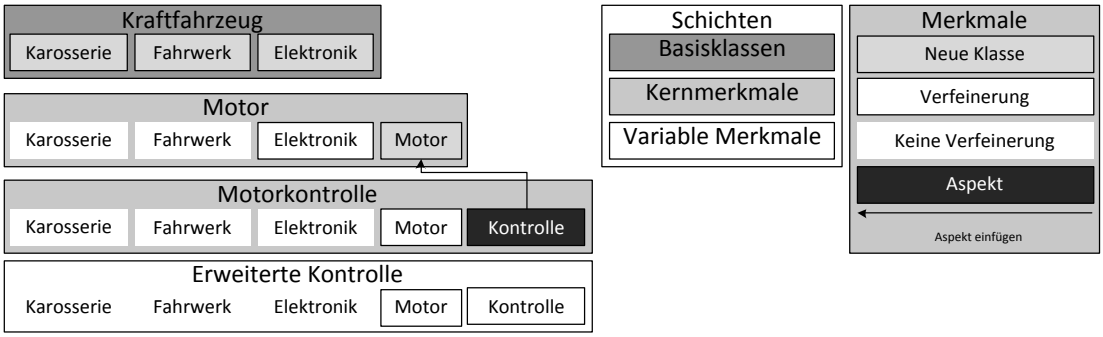

Abbildung 4.1: Kollaborationsdiagramm: Legende

4.2 Basis- und Kernmerkmale

Im Folgenden werden die Klassen und Verfeinerungen vorgestellt, welche durch die Basis- und die Kernmerkmale eingeführt werden. Die Funktionalität der Basisklassen umfasst beispielsweise die grundlegenden Elemente für den Start des Programms, für welches der Entwickler die konfigurierbare Transaktionsunterstützung nutzen möchte, und für die Kommunikation mit *Amazon S3*. Diese Elemente werden auch dann benötigt, wenn keine Transaktionsunterstützung aktiv ist. Durch das Hinzufügen der *Transaktionsunterstützung* wird die Basis um Kernmerkmale erweitert, die nicht variabel sind.

4.2.1 Basisklassen

Das Basismerkmal enthält die Klasse *Main*, die den Einstiegspunkt für die Ausführung der Anwendung liefert, welche eine Transaktionsunterstützung erhalten soll (siehe Abbildung 4.2). Solch ein Einstiegspunkt besteht zum Beispiel aus dem Vorhandensein einer Funktion *main*.

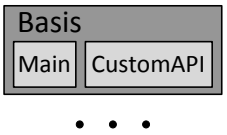

Abbildung 4.2: Kollaborationsdiagramm: Basis

Des Weiteren wird die Klasse *CustomAPI* zur Verfügung gestellt, die in der Klasse *Main* beim Programmstart initialisiert werden muss, so dass durch den Aufruf der *CustomAPI* das Hinzufügen (*put*), Löschen (*delete*) und Auslesen (*get*) von *S3*-Objekten möglich wird. Dazu muss jede der drei Funktionen *put*, *get* und *delete* den *Bucket*-Namen und den individuellen Schlüssel als Übergabewerte erhalten. Die Funktion zum Hinzufügen benötigt zusätzlich noch die Datei, welche eingefügt werden soll. Innerhalb dieser Funktionen wird mit Hilfe des Zugriffsschlüssels und des privaten Schlüssels, die durch den Entwickler angegeben werden müssen, eine Verbindung zum *S3-Client* hergestellt und auf die *S3*-Objekte entsprechend der auszuführenden Operation zugegriffen.

4.2.2 Transaktionsunterstützung

Durch die Aktivierung der *Transaktionsunterstützung* wird die Klasse *CustomAPI* um Instanzen der Klassen *Transaktionsmanager* und *Speichermanager* erweitert (siehe Abbildung 4.3), so dass diese ab der Initialisierung der *CustomAPI* und somit für die vollständige Laufzeit der Anwendung zur Verfügung stehen.

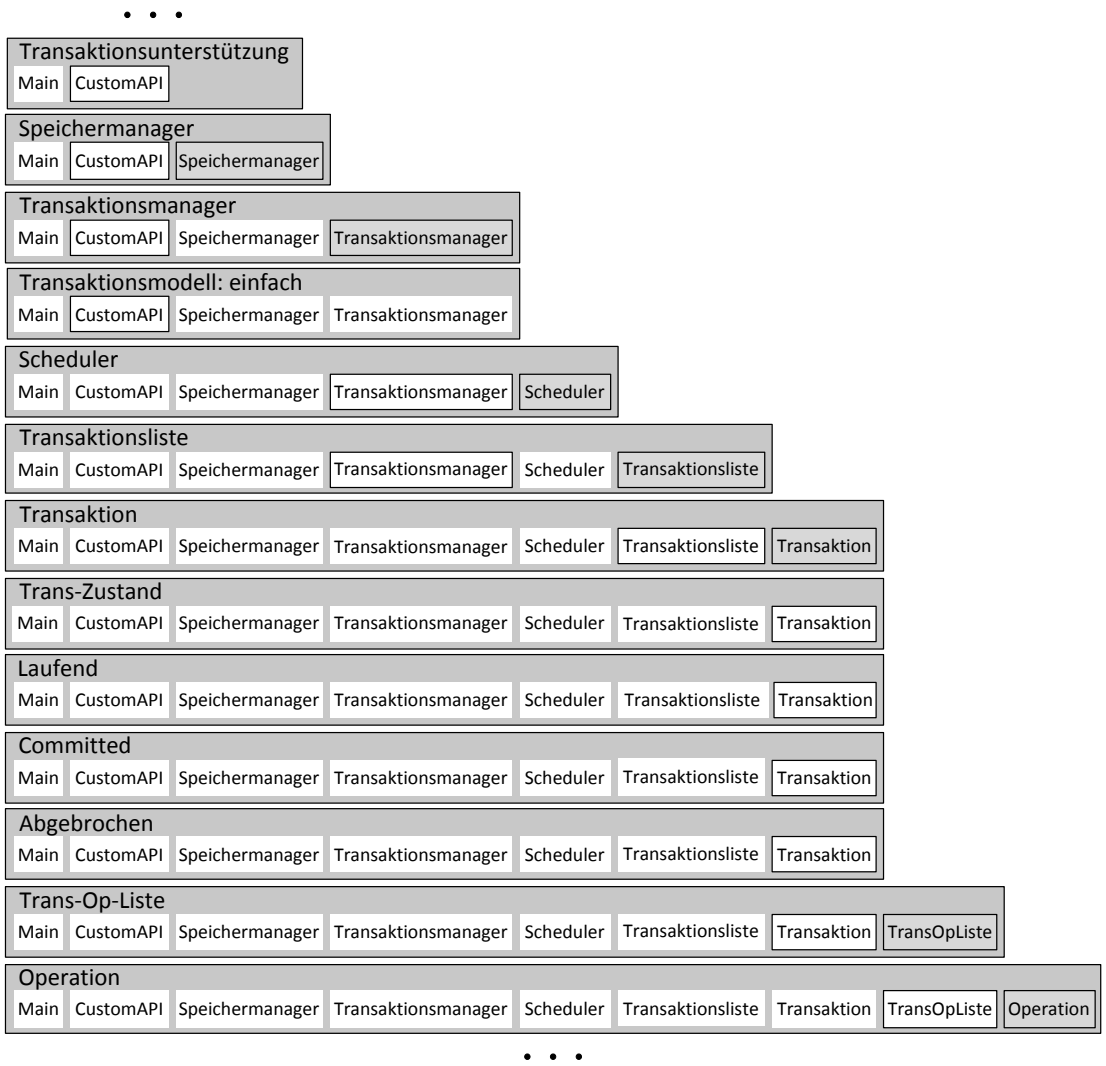

Abbildung 4.3: Kollaborationsdiagramm: Kernmerkmale

Zusätzlich erfolgt durch die Auswahl des Merkmals *Transaktionsunterstützung* die Aktivierung weiterer Kernmerkmale, welche durch den Transaktionsmanager beziehungsweise den Speichermanager angesprochen werden können. Dazu gehören zum Beispiel Erweiterungen der *CustomAPI* sowie das Hinzufügen neuer Klassen wie dem *Scheduler*, der *Transaktionsliste*, der *Transaktion* und der *Operation*. Im Folgenden werden die Erweiterungen durch die Kernmerkmale und ihr Nutzen für die Transaktionsunterstützung kurz erläutert.

Erweiterungen durch die Kernmerkmale

Durch das als Standard verwendete *einfache Transaktionsmodell* wird die *CustomAPI* um die Funktionen *beginTransaktion* zum Starten einer Transaktion und *commitTransaktion* zum Beenden einer Transaktion erweitert. Zusätzlich müssen die Funktionen *put*, *get* and *delete* überschrieben werden, damit keine Operation auf *S3* direkt ausgeführt wird. Stattdessen werden die jeweiligen Operationen zu der *Operationsliste* ihrer Transaktion hinzugefügt.

Durch den Aufruf der Funktion *beginTransaktion* wird eine neue Transaktion initialisiert, welche eine eindeutige Transaktions-ID generiert und eine Operationenliste erhält, zu der eine Operation der Art *BOT* hinzugefügt wird (siehe Abbildung 4.4). Eine neue Operation generiert automatisch eine eindeutige Operations-ID und speichert die Transaktions-ID der Transaktion, durch welche die Operation angelegt wurde. Jede neue Operation erhält den Zustand *nicht ausgeführt* und jede neue Transaktion wird mit dem Zustand *BOT* initialisiert.

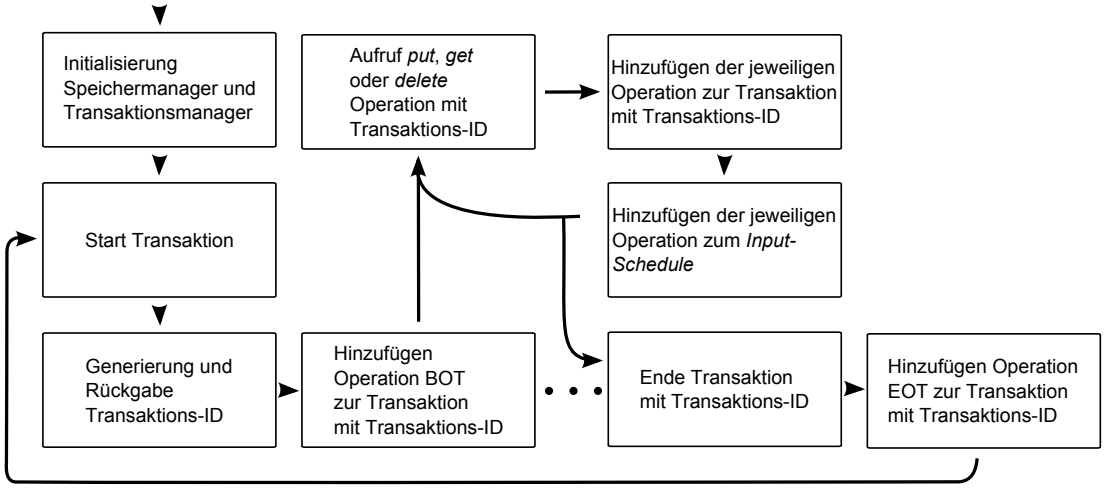

Abbildung 4.4: Ablauf: Hinzufügen einer Transaktion

Den Operationen, die beim Aufruf von *put*, *get* und *delete* erstellt werden, muss die Transaktions-ID derjenigen Transaktion übergeben werden, zu der die Operationen gehören. Zu diesem Zweck wird bei jedem Aufruf von *beginTransaktion* die Transaktions-ID zurückgegeben, die beim Erstellen der neuen Transaktion ermittelt wurde. So kann der Entwickler jede Operation einer spezifischen Transaktion zuordnen. Dies gilt auch für den Aufruf von *commitTransaktion*, da bekannt sein muss, welche Transaktion beendet werden soll. Wie die Funktion *beginTransaktion* erzeugt auch die Funktion *commitTransaktion* eine Operation, welcher jedoch die Art *EOT* zugewiesen wird. Um die Operationen ausführen zu können, werden sie nicht nur ihren Transaktionen zugewiesen, sondern zusätzlich an den *Input-Schedule* des *Schedulers* gehängt. Der *Scheduler* wird durch den Transaktionsmanager initialisiert und prüft regelmäßig, welche Operationen aus dem *Input-Schedule* ausgeführt werden dürfen beziehungsweise abgelehnt werden müssen (siehe Abbildung 4.5).

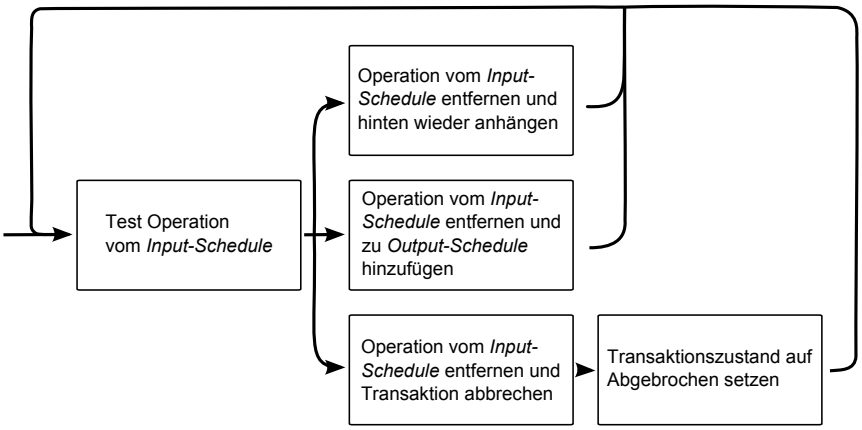

Abbildung 4.5: Ablauf: *Scheduler*

Nicht erlaubte Operationen werden, abhängig vom Synchronisationsverfahren, vom Anfang des *Input-Schedules* entfernt und hinten wieder angehängt oder verworfen, da das negative Ergebnis des Tests vom *Scheduler* zu einem Abbruch der Transaktion führt. Bei einem Abbruch wird die Transaktion in den Zustand *Abgebrochen* versetzt. Erlaubte Operationen hingegen werden vom *Input-Schedule* genommen und ans Ende des *Output-Schedules* hinzugefügt. Dieser *Output-Schedule* wird wiederum regelmäßig vom Speichermanager ausgelesen, indem der Speichermanager die jeweils vorderste Operation aus dem *Output-Schedule* entfernt und die Operation ausführt (siehe Abbildung 4.6). Ist diese Operation ein *BOT*, wird die zugehörige Transaktion in den Zustand *Laufend* und bei einem *EOT* in den Zustand *Committed* versetzt. Die Operationen *put*, *get* und *delete* verändern den Transaktionszustand nicht, aber setzen den Operationszustand von *nicht ausgeführt* auf *ausgeführt*.

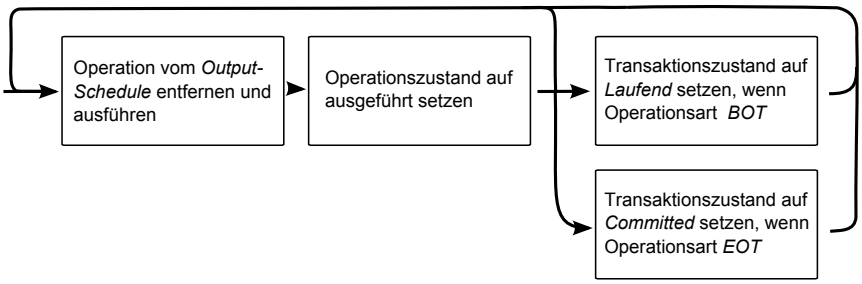

Abbildung 4.6: Ablauf: Speichermanager

4.3 Variable Merkmale des Transaktionsmanagers

Variable Merkmale zeichnen sich dadurch aus, dass sie entweder vollständig weggelassen werden können, wie zum Beispiel das Merkmal *Wartegraph*, oder eine Auswahl zwischen gleichwertigen, aber alternativen Merkmalen bieten. So muss sich beispielsweise für ein Synchronisationsverfahren entschieden werden, um zusammen mit den Kernmerkmalen eine Transaktionsunterstützung gewährleisten zu können. Im Folgenden werden die Klassen, Aspekte und Verfeinerungen der Merkmale vorgestellt, welche vorrangig dem Transaktionsmanager zuzuordnen sind.

4.3.1 Transaktionszustände

Zusätzlich zu den Zuständen *Laufend*, *Committed* und *Abgebrochen* können optional die Zustände *Verzögert* und *Gestoppt* hinzugewählt werden, was dazu führt, dass die Klasse *Transaktion* um diese Zustände erweitert werden muss (siehe Abbildung 4.7). Der Zustand *Verzögert* muss verwendet werden, wenn ein Synchronisationsverfahren wie zum Beispiel die Sperrverfahren durch das Abweisen einer Operation nicht den Abbruch der Transaktion auslösen, sondern die Operation solange verzögern, bis eine Ausführung möglich ist. Wird hingegen durch das Abweisen einer Operation die dazugehörige Transaktion abgebrochen, sind verschiedene Folgeaktionen möglich.

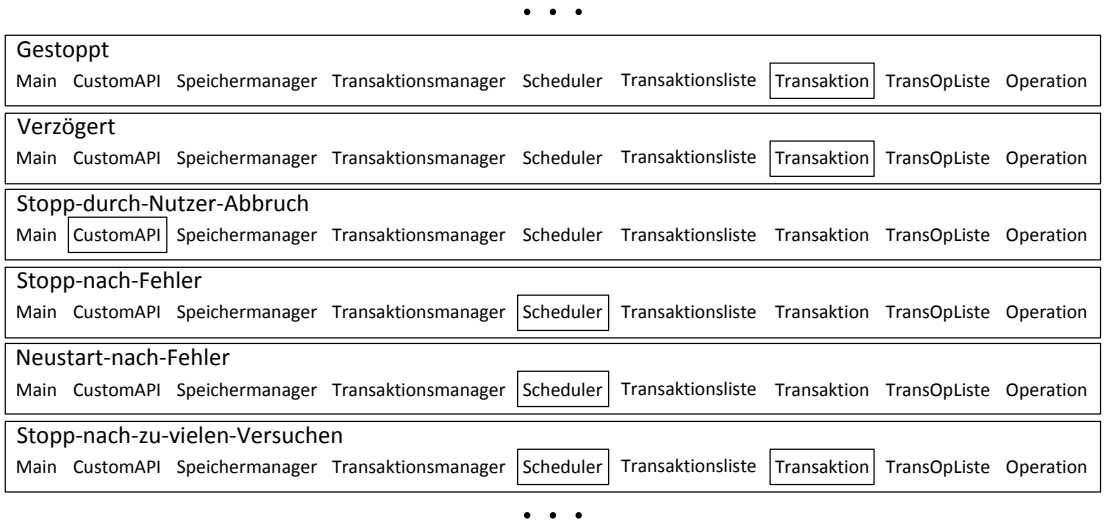

Abbildung 4.7: Kollaborationsdiagramm: Optionale Transaktionszustände

Soll die Transaktion durch den Abbruch nicht mehr ausgeführt werden (*Stopp-nach-Fehler*), muss sie in den Zustand *Gestoppt* überführt werden. Damit dies möglich wird, muss der *Scheduler* nach dem Abweisen der jeweiligen Operation die Transaktion in den Zustand *Gestoppt* setzen und alle Operationen, die noch zu dieser Transaktion gehören, aus dem *Input-* und *Output-Schedule* entfernen. Auch in dem Fall, dass die Transaktion stattdessen neu gestartet werden soll (*Neustart-nach-Fehler*), müssen die Operationen aus dem *Input-* und *Output-Schedule* entfernt werden, aber alle Operationen werden wieder in den Zustand *nicht ausgeführt* überführt und die Operationen der Transaktion neu in den *Input-Schedule* übernommen. Der Zustand der Transaktion muss wieder in den Startzustand *BOT* versetzt werden, welcher erst wieder in *Laufend* abgeändert werden kann, wenn die *BOT*-Operation der Transaktion aufgerufen wird. Da es jedoch möglich ist, dass der Neustart einer Transaktion wiederholt zum Abbruch der Transaktion führt, kann mit Hilfe des Merkmals *Stopp-nach-zu-vielen-Versuchen* ein endgültiges Anhalten der Transaktion erreicht werden, wenn sie zu oft neu gestartet wurde. Dazu wird die Transaktion um einen Zähler erweitert, der festhält, wie oft die Transaktion schon neu gestartet wurde. Wird die Transaktion wiederholt durch den *Scheduler* abgebrochen und die maximale Abbruchanzahl ist erreicht, wird die Transaktion nicht noch einmal neugestartet, sondern in den Zustand *Gestoppt* überführt.

Eine weitere Ergänzung stellt das Hinzufügen einer Abbruchfunktion für den Entwickler dar (*Stopp-durch-Nutzer-Abbruch*), so dass dieser nicht nur die Funktion *commitTransaktion* aufrufen kann, sondern auch eine Funktion *abortTransaktion*, um eine

Transaktion zu beenden. Durch diese Funktion wird die Transaktion automatisch in den Zustand *Gestoppt* überführt und alle ihre Operationen aus dem *Input-* und *Output-Schedule* entfernt.

4.3.2 Kompensierende Transaktionen

Beim Abbruch von Transaktionen ist es wichtig, die durchgeführten Änderungen durch die Transaktionen wieder rückgängig machen zu können, sofern die Änderungen direkt auf den persistenten Speicher und nicht wie bei der optimistischen Synchronisation auf Objektkopien geschrieben werden (siehe Abschnitt 2.2). Sollen Änderungen von Transaktionen mit der Hilfe eines Logs zurückgesetzt werden (*Kompensierende-Transaktionen: Durch-Log*), muss dieses Log ausgelesen werden, sobald ein manueller Abbruch durch den Entwickler oder ein automatischer Abbruch durch den *Scheduler* ausgelöst wird (siehe Abbildung 4.8).

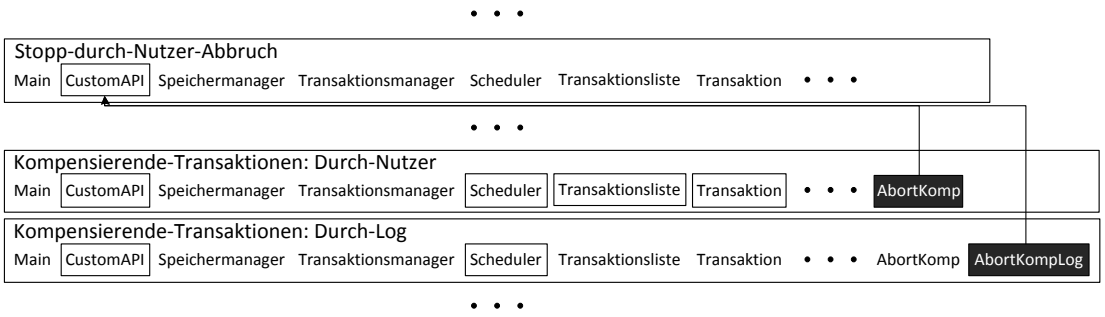

Abbildung 4.8: Kollaborationsdiagramm: Kompensierende Transaktionen

Die Funktion *abortTransaktion* kommt durch ein optionales Merkmal hinzu, so dass an dieser Stelle keine Verfeinerung der Klasse *CustomAPI* für die Funktion *abortTransaktion* angewendet werden kann, da Verfeinerungen, im Gegensatz zu Aspekten, das Vorhandensein der zu ändernden Elemente voraussetzen. Aus diesem Grund wird an dieser Stelle und mit Hilfe des optionalen Webens ein Aspekt verwendet [Leich et al., 2005a; Kästner, 2007], um die Funktion *abortTransaktion* mit zusätzlicher Funktionalität zu erweitern. Zu beachten ist, dass für folgende Erweiterungen der Funktion *abortTransaktion* jeweils ein neuer Aspekt angelegt werden muss, sofern nicht ein bereits vorhandener Aspekt zum Erweitern der Funktion *abortTransaktion* verfeinert werden kann. Dies ist genau dann der Fall, wenn die Erweiterungen für die Funktion *abortTransaktion* voneinander unabhängig sind. So muss für die Erweiterung durch das Merkmal *Kompensierende-Transaktionen: Durch-Nutzer* ein weiterer Aspekt angelegt werden, da auch dieses Merkmal bei einem manuellen Abbruch der Transaktion das Rücksetzen der Änderungen einleiten muss.

Durch das Merkmal *Kompensierende-Transaktionen: Durch-Nutzer* werden die Änderungen nicht aus dem Log gelesen und müssen folglich auf andere Art und Weise festgehalten werden. Dazu wird beim Anlegen einer Transaktion durch *beginTransaktion* gleichzeitig eine kompensierende Transaktion angelegt, welche als solche markiert wird. Durch das Anlegen der kompensierenden Transaktion muss die Transaktions-ID der ursprünglichen Transaktion übergeben werden, damit diese Transaktion beim Abbruch weiß, welche Transaktion sie als kompensierende aufrufen muss. Bei jeder anschließend aufgerufenen Operation muss des Weiteren angegeben werden, ob sie zur ursprünglichen oder kompensierenden Transaktion gehört, damit kompensierende Operationen

nicht durch den *Scheduler* getestet, sondern direkt in den *Output-Schedule* übernommen werden. Auch beim Auslesen der nächsten auszuführenden Operation werden die kompensierenden Operationen nicht direkt ausgeführt, sondern erhalten eine Markierung, welche angibt, dass die jeweilige Operation ausgeführt werden muss, sobald die kompensierende Transaktion ausgelöst wird. Wenn der Entwickler oder der *Scheduler* eine Transaktion abbricht, wird bei der Operation von der kompensierenden Transaktion gestartet, welche als letzte die Markierung zum Ausführen erhalten hat und die kompensierende Transaktion rückwärts abgearbeitet.

4.3.3 Erweiterte Transaktionsmodelle

Durch das Merkmal der *erweiterten Transaktionsmodelle* wird die Klasse *Transaktion* verfeinert (siehe Abbildung 4.9), so dass für jede Transaktion die Transaktions-ID der Vatertransaktion und die Transaktions-IDs aller Kindtransaktionen festgehalten werden kann. Diese IDs werden benötigt, um beim Abbruch einer Transaktion zum Beispiel auch deren Vatertransaktion oder Kindtransaktionen abbrechen zu können.

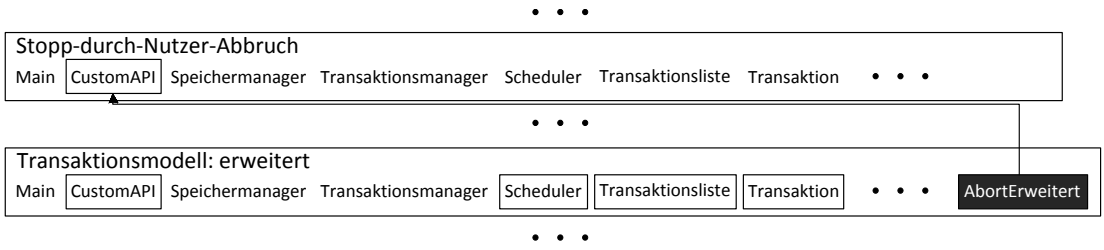

Abbildung 4.9: Kollaborationsdiagramm: Erweiterte Transaktionsmodelle Teil 1

Um den Transaktionen diese IDs zuweisen zu können, muss beim Aufruf der Funktion *beginTransaktion* jeder Subtransaktion die ID der Vatertransaktion übergeben werden, so dass die Vatertransaktions-ID der Subtransaktion gesetzt werden kann und gleichzeitig auch die Vatertransaktion die neu generierte Transaktions-ID der Subtransaktion erhält. Diese Informationen werden benötigt, um bei einem manuellen Abbruch einer Transaktion durch den Entwickler oder durch einen automatischen Abbruch durch den *Scheduler* gegebenenfalls die zur Transaktion gehörenden Vatertransaktion beziehungsweise Subtransaktionen abbrechen zu können. Ebenso ist es beispielsweise erforderlich, beim *Commit* der Vatertransaktion zu testen, ob auch alle Subtransaktionen erfolgreich ein *Commit* ausgeführt haben.

Je nach Art des verwendeten Transaktionsmodells werden die Folgeaktionen, die durch einen Abbruch oder durch das erfolgreiche Beenden von Transaktionen und Subtransaktionen ausgelöst werden, angepasst (siehe Abbildung 4.10). So gilt bei den *geschlossen geschachtelten Transaktionen*, dass beim Abbruch der Vatertransaktion oder einer Subtransaktion auch alle anderen beteiligten Transaktionen beziehungsweise Subtransaktionen abbrechen müssen. Außerdem dürfen bei einem *Commit* von Subtransaktionen noch keine Ergebnisse an andere Transaktionen freigegeben werden. Im Gegensatz dazu dürfen abgeschlossene, *offen geschachtelte Transaktionen* beim *Commit* ihrer Subtransaktionen schon Ergebnisse freigeben.

Offen geschachtelte Transaktionen haben des Weiteren noch die Möglichkeit, *nicht-vitale Subtransaktionen* zugewiesen zu bekommen. Dazu muss jede Subtransaktion beim Aufruf der Funktion *beginTransaktion* durch den Entwickler als vital oder nicht-vital eingestuft werden. Damit dies möglich wird, muss die Klasse *Transaktion* um diese

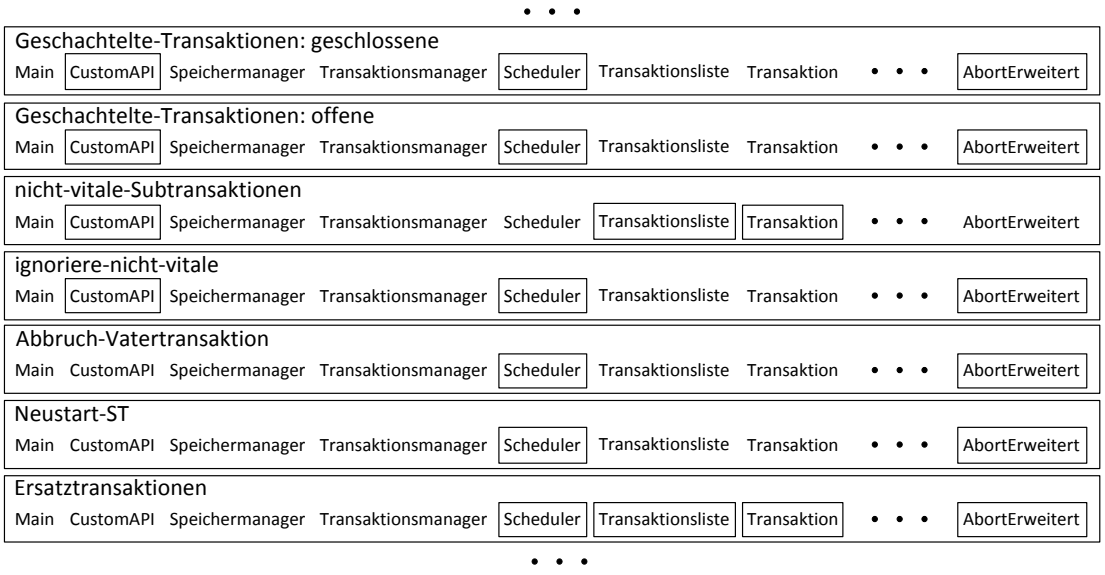

Abbildung 4.10: Kollaborationsdiagramm: Erweiterte Transaktionsmodelle Teil 2

zusätzliche Eigenschaft erweitert und die Übergabewerte der Funktion *beginTransaktion* angepasst werden. Können nicht-vitale Subtransaktionen ignoriert werden (*ignoriere-nicht-vitale*), so wird mit Hilfe dieser Eigenschaft bei einem Abbruch der Subtransaktion festgestellt, dass ihr Abbrechen nicht den Abbruch der Vater- und Kindtransaktionen auslöst. Ebenso ist es bei einem *Commit* der Vatertransaktion nicht notwendig, dass nicht-vitale Subtransaktionen auf ein erfolgreiches *Commit* getestet werden. Für die vitalen Transaktionen gelten die Abbruch- und *Commit*-Regeln jedoch weiterhin.

Zusätzlich ist es möglich, aus verschiedenen Folgeaktionen beim Abbruch einer vitalen Subtransaktion zu wählen, so dass, abhängig von der Auswahl, der *Scheduler* und die Funktion *abortTransaktion* der *CustomAPI* angepasst werden müssen. So kann der Entwickler für jede Subtransaktion angeben, was geschehen soll, wenn diese Subtransaktion abbricht. Zusätzlich zu der Möglichkeit, den *Abbruch der Vatertransaktion* auszulösen, kann auch der *Neustart der Subtransaktion* angegeben werden. In dem Fall werden, wie beim Neustart einer einfachen Transaktion, die Operationen aus dem *Input-* und *Output-Schedule* entfernt, der Status aller Operationen dieser Subtransaktion auf *nicht ausgeführt* gesetzt und anschließend wieder in den *Input-Schedule* geladen. Die dritte Möglichkeit besteht im Angeben einer *Ersatztransaktion*, die ausgeführt wird, wenn die Subtransaktion scheitert. Dazu muss der Entwickler den Funktionsnamen der Ersatztransaktion angeben. Der Funktionsname wird dann der Transaktion als weitere Eigenschaft zugeteilt und kann beim Abbruch die entsprechende Funktion ausführen.

Weitere Transaktionsmodelle sind die *flexiblen Subtransaktionen* und die *Sagas* (siehe Abbildung 4.11). Die flexiblen Subtransaktionen dürfen nur dann Ergebnisse beim *Commit* freigeben, wenn sie als kompensierbar markiert wurden. Dazu wird den Subtransaktionen diese Information beim Aufruf der Funktion *beginTransaktion* übergeben. Bei den Sagas hingegen müssen alle Subtransaktionen kompensierbar sein und da durch die Aktivierung des Merkmals *Sagas* automatisch das Merkmal *Kompensierende-Transaktionen: Durch-Nutzer* aktiviert wird, muss beim Hinzufügen von Subtransaktionen auch jeweils die kompensierende Transaktion mit aufgebaut werden. Aufgrund der Eigenschaft von Sagas, dass sie nur eine Ebene Subtransaktionen haben können, muss

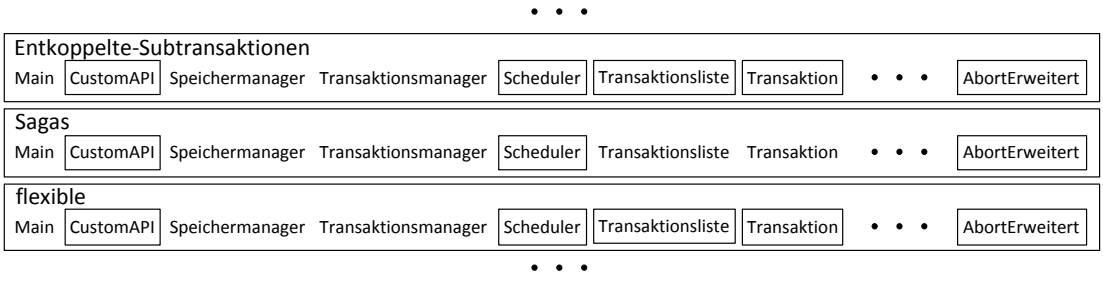

Abbildung 4.11: Kollaborationsdiagramm: Erweiterte Transaktionsmodelle Teil 3

des Weiteren bei jedem Aufruf von *beginTransaktion* getestet werden, ob die Vorgängersubtransaktion schon ein *Commit* erhalten hat. Ein ähnlicher Test ist für alle Varianten der geschachtelten Transaktionen notwendig, da gilt, dass Subtransaktionen nur innerhalb des zeitlichen Rahmens ihrer Vatertransaktionen laufen dürfen. Folglich muss bei jedem *Commit* einer Subtransaktion oder der Wurzeltransaktion überprüft werden, ob ihre Kindtransaktionen schon den *Commit*-Befehl erhalten haben. Im Gegensatz zu den geschachtelten Transaktionen gilt diese Regel für die *entkoppelten Subtransaktionen* nicht. Subtransaktionen können bei diesem Modell zu jeder Zeit aufgerufen und beendet werden. Zusätzlich bedeutet auch ein Abbruch einer Subtransaktion nicht, dass die Vatertransaktion abbrechen muss. Da entkoppelte Subtransaktionen aber die Möglichkeit bieten, beim Abbruch beziehungsweise beim erfolgreichen *Commit* einer Transaktion eine andere Transaktion auszulösen, müssen beim Aufruf der Funktion *beginTransaktion* die Folgetransaktionen angegeben werden, die durch den Abbruch oder das *Commit* der Transaktion ausgelöst werden sollen. Dazu wird auch die Klasse *Transaktion* um diese beiden Informationen erweitert, so dass diese bei Bedarf ausgelesen werden können.

4.3.4 Transaktionszeitmarken

Transaktionszeitmarken (*TransZeitmarke*) geben an, wann eine Transaktion die Funktionen *beginTransaktion* und *commitTransaktion* aufgerufen hat. Dazu müssen beim Aufruf dieser beiden Funktionen die Zeitmarken der jeweiligen Transaktion bestimmt und gesetzt werden (siehe Abbildung 4.12).

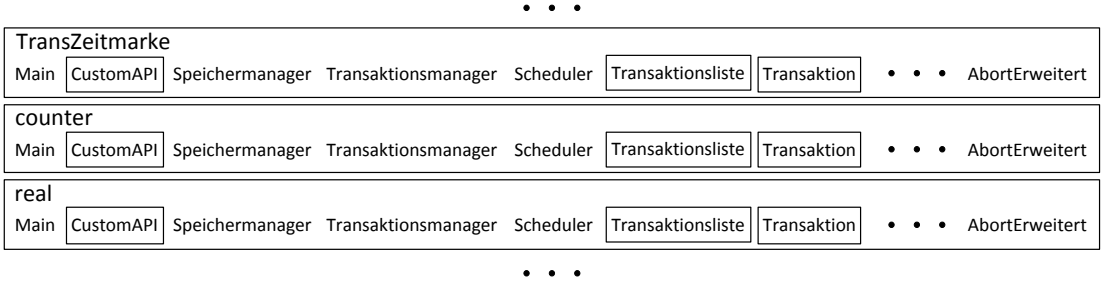

Abbildung 4.12: Kollaborationsdiagramm: Transaktionszeitmarken

Um das Setzen der Transaktionszeitmarken zu gewährleisten, wird die Klasse *Transaktion* um die Eigenschaften *BOT*-Zeitstempel und *EOT*-Zeitstempel erweitert. Da das Merkmal für die Zeitmarken (*TransZeitmarke*) jedoch entweder mit Hilfe eines Zählers (*counter*) oder mit Hilfe realer Zeitstempel (*real*) umgesetzt werden kann, müssen die beiden Merkmale die Klasse *Transaktion* entsprechend anpassen. Erfolgt die Umset-

zung der Zeitmarken durch das Zählermerkmal, so müssen außerdem die aktuell höchsten Werte des *BOT*- und *EOT*-Zählers zusätzlich festgehalten und jedes Mal, wenn ein *beginTransaktion* oder *commitTransaktion* aufgerufen wird, inkrementiert werden.

4.3.5 Kaskadierende Abbrüche und Wartegraph

Die Erkennung und Behandlung von *kaskadierenden Abbrüchen* und der Aufbau und die Verwendung des *Wartegraphen* sind durch optionale Merkmale auswählbar. Dies führt dazu, dass der Transaktionsmanager um diese beiden Hilfsstrukturen erweitert wird, wenn eines oder beide Merkmale aktiviert werden (siehe Abbildung 4.13). Bezogen auf die kaskadierenden Abbrüche, bedeutet dies, dass der Transaktionsmanager um eine Hilfsstruktur zum Hinzufügen und Auslesen von Liest-von-Beziehungen erweitert wird. Wird eine Schreiboperation ausgeführt, wird diese und die Transaktions-ID der schreibenden Transaktion notiert und für jede folgende Leseoperation anderer Transaktionen eine Liest-von-Beziehung angelegt, bis eine neue Schreiboperation ausgeführt wird. Durch die Liest-von-Beziehungen wird somit festgehalten, welche Transaktion geschriebene Werte von einer anderen Transaktion liest und folglich auch abgebrochen werden müsste, wenn die Transaktion abbricht, durch die der Wert geschrieben wurde. Da kaskadierende Abbrüche sowohl beim automatischen Abbruch durch den *Scheduler* als auch durch den manuellen Abbruch durch den Entwickler ausgelöst werden können, müssen sowohl die Funktion *abortTransaktion* als auch die Klasse *Scheduler* angepasst werden.

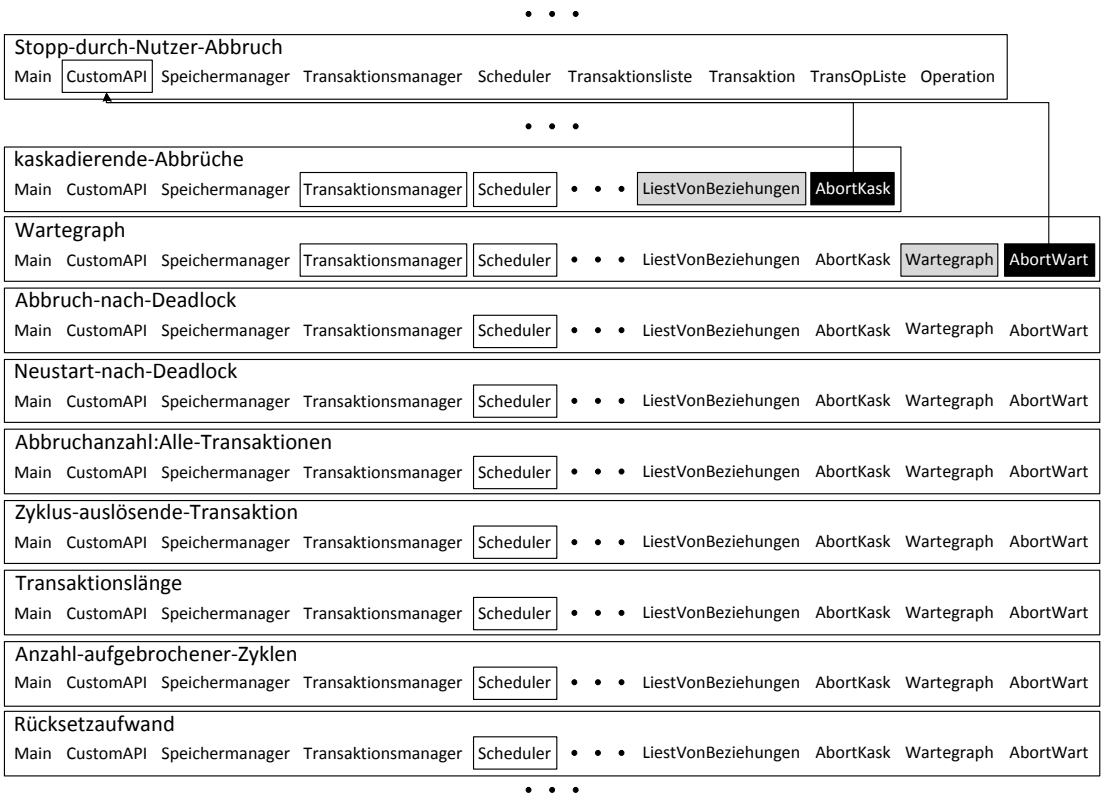

Abbildung 4.13: Kollaborationsdiagramm: Kaskadierende Abbrüche und Wartegraph

Der Wartegraph hingegen wird für die Erkennung und Auflösung von *Deadlocks* benötigt, was bedeutet, dass durch den Wartegraphen Wartebeziehungen zwischen Transaktionen festgehalten werden müssen. Wartebeziehungen treten genau dann auf, wenn Operationen zum Beispiel durch Sperren verzögert werden, indem sie mit ihrer Ausführung warten müssen, bis die Sperre aufgehoben wurde. Immer dann wenn eine Operation mit Hilfe des *Schedulers* abgewiesen wurde, wird im Wartegraphen festgehalten, welche Transaktion auf die sperrende Transaktion warten muss. Nach dem Eintragen dieser Wartebeziehung wird getestet, ob dadurch ein Zyklus im Graphen entstanden ist. Wenn ein Zyklus erkannt wurde, werden entweder eine Transaktion oder *alle Transaktionen* abgebrochen oder neugestartet (*Abbruch-nach-Deadlock*, *Neustart-nach-Deadlock*). Falls genau eine Transaktion abgebrochen werden soll, wird, je nach gewähltem Abbruchkriterium (*Zyklus-auslösende-Transaktion*, *Transaktionslänge*, *Anzahl-aufgebrochener-Zyklen*, *Rücksetzaufwand*), der Algorithmus zum Bestimmen der abzubrechenden Transaktion angepasst und der Abbruch durch den *Scheduler* eingeleitet. Da mehrere Abbruchkriterien gleichzeitig aktiv sein können, muss eine Abarbeitungsreihenfolge der Kriterien durch den Entwickler der konfigurierbaren Transaktionsverwaltung festgelegt werden, so dass das jeweils nächste Kriterium getestet werden kann, wenn durch ein Kriterium mehrere Transaktionen für einen Abbruch in Frage kommen würden. Wartebeziehungen werden aber nicht nur beim Aufteten eines *Deadlocks* aus dem Wartegraphen entfernt, sondern auch, wenn zum Beispiel eine Sperre aufgehoben wurde oder Transaktionen durch den Aufruf von *abortTransaktion* beziehungsweise durch den *Scheduler* abgebrochen werden.

4.3.6 Synchronisationsverfahren

Durch das Hinzufügen von Synchronisationsverfahren wird vor allem der *Scheduler* erweitert und der Test, ob eine Operation in den *Output-Schedule* übernommen werden darf, angepasst. Zusätzlich bedingen einige Verfahren Hilfsstrukturen wie einen *Konfliktgraphen*, *Read-* und *Write-Sets* beziehungsweise Erweiterungen der Informationen für Objekte, Operationen und Transaktionen wie beispielsweise *Sperren* und *Zeitmarken*.

- **Serielle Synchronisation**

  Da Transaktionen bei der *seriellen Synchronisation* nacheinander abgearbeitet werden müssen, werden zunächst nur Operationen einer Transaktion in den *Input-Schedule* überführt. Jedes Mal, wenn eine andere Transaktion als die gerade aktive eine Operation zum *Input-Schedule* hinzufügen möchte, wird deren Transaktions-ID durch den *Scheduler* in einer Liste gespeichert, so dass beim *Commit* der aktuell abgearbeiteten Transaktion die ID der nächsten auszuführenden Transaktion ermittelt werden kann (siehe Abbildung 4.14). Mit Hilfe dieser Transaktions-ID lässt sich die jeweilige Transaktion auslesen und ihre Operationen können nacheinander in den *Input-Schedule* überführt werden.

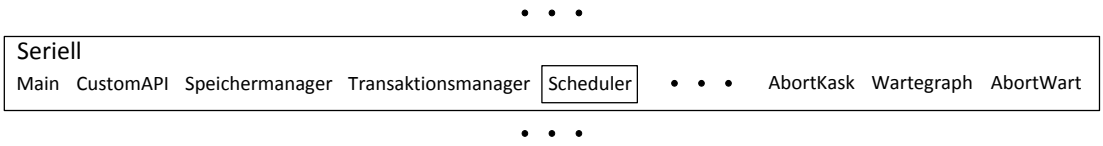

Abbildung 4.14: Kollaborationsdiagramm: Serielle Synchronisation

- **Serialisierbarkeitsgraphentester**

  Der Serialisierbarkeitsgraphentester (*SGT*) benötigt als neue Hilfsstruktur einen Konfliktgraphen, welcher durch den *Scheduler* angelegt wird (siehe Abbildung 4.15). Zu diesem Konfliktgraphen können Konfliktbeziehungen zwischen Transaktionen hinzugefügt und entfernt werden. Außerdem lässt sich der Konfliktgraph entweder mit Hilfe der *topologischen Sortierung* oder einer *Tiefensuche* auf Zyklen untersuchen.

  Durch die Verwendung des Serialisierbarkeitsgraphentesters wird vor dem Überführen von Operationen aus dem *Input-Schedule* in den *Output-Schedule* überprüft, ob im Konfliktgraphen ein Zyklus entstehen würde, wenn die, durch diese Operation hervorgerufenen Konflikte in den Konfliktgraphen aufgenommen werden. Für den Fall, dass die jeweilige Operation einen Zyklus auslösen würde, muss diese Operation zurückgewiesen und deren Transaktion abgebrochen werden. Wenn kein Zyklus entsteht, verbleiben die neuen Konfliktbeziehungen im Konfliktgraphen und die Operation wird in den *Output-Schedule* überführt.

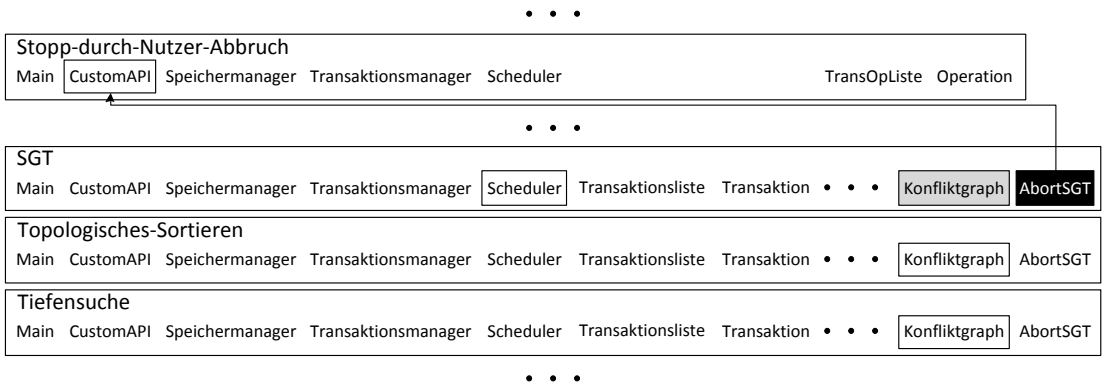

Abbildung 4.15: Kollaborationsdiagramm: Serialisierbarkeitsgraphentester

Wenn Transaktionen erfolgreich abgeschlossen werden, können sie aus dem Konfliktgraphen entfernt werden, sofern für sie keine eingehenden Konfliktbeziehungen mehr bestehen. Zusätzlich müssen Transaktionen auch dann aus dem Konfliktgraphen entfernt werden, wenn sie manuell durch den Entwickler abgebrochen werden, was dazu führt, dass die Funktion *abortTransaktion* der *CustomAPI* ebenfalls erweitert werden muss.

- **Optimistische Synchronisation**

  Da Transaktionen durch die *optimistische Synchronisation* drei verschiedene Phasen haben können, wird die Klasse *Transaktion* um die Möglichkeit erweitert, die jeweilige Phase setzen und auslesen zu können (siehe Abbildung 4.16). Sobald eine Transaktion beginnt, wird sie in die Lesephase versetzt, was bedeutet, dass ihre Operationen auf lokalen Kopien, die der *Transaktionspuffer* des *Puffermanagers* anlegt, ausgeführt werden. Außerdem wird für jede Transaktion durch den Transaktionspuffer mit Hilfe von *Read-* und *Write-Sets* gespeichert, welche Lese- und Schreiboperationen die Transaktion ausgeführt hat. Die Ausführung der Operationen auf den Kopien erfolgt, wenn sie vom *Input-* in den *Output-Schedule* überführt werden, jedoch wird der *Output-Schedule* nicht mehr vom Speichermanager ausgelesen, damit die Änderungen noch nicht in den persistenten Speicher gelan-

gen können. Sobald eine *EOT*-Operation an der Reihe ist, wird die dazugehörige Transaktion in die Validierungsphase versetzt, sofern keine andere Transaktion schon in der Validierungsphase ist. In dem Fall muss die Transaktions-ID der fertigen Transaktion gesichert werden, damit sie zu einem späteren Zeitpunkt in die Validierungsphase übergehen kann.

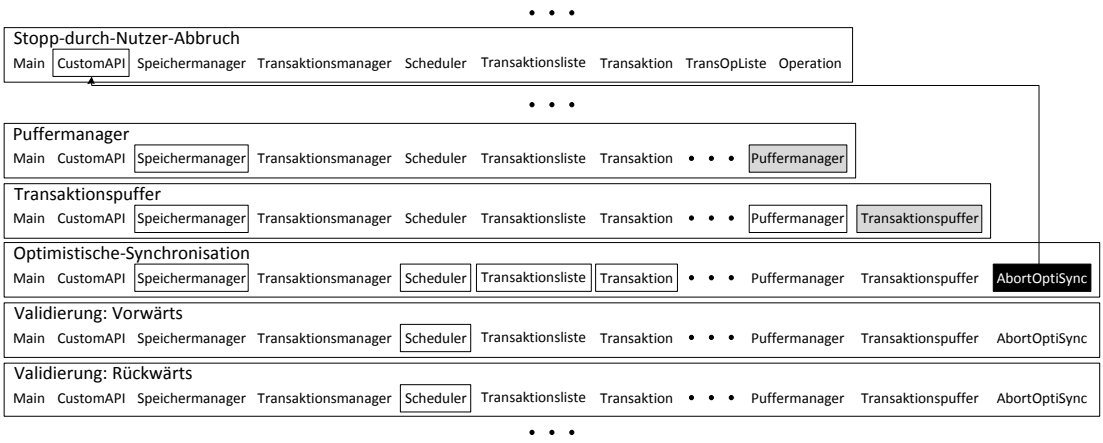

Abbildung 4.16: Kollaborationsdiagramm: Optimistische Synchronisation

Während der Validierungsphase werden die *Read-* und *Write-Sets* der Transaktionen auf Konflikte überprüft. Ist die *Rückwärtsvalidierung* aktiv, werden die *Write-Sets* von bereits abgeschlossenen Transaktionen, sofern sie vor Beginn der validierenden Transaktion beendet wurden, mit dem *Read-Set* der aktuellen Transaktion abgeglichen. Im Gegenzug dazu werden im Falle der *Vorwärtsvalidierung* die *Read-Sets* von noch laufenden Transaktionen gegen das *Write-Set* der validierenden Transaktion auf Konflikte getestet. Da jedoch auch beide Validierungsarten gleichzeitig aktiv sein können, müssen in diesem Fall beide Konfliktprüfungen stattfinden. Je nach Validierungsart, werden die noch laufenden beziehungsweise die validierende Transaktion abgebrochen. Wenn kein Konflikt auftritt, werden die Werte der Objektkopien durch den Speichermanager in den persistenten Speicher übernommen und anschließend die Validierungsphase für andere Transaktionen freigegeben. Auch für dieses Verfahren ist eine Anpassung des manuellen Abbruchs von Transaktionen notwendig, damit Transaktionspuffereinträge für die abgebrochenen Transaktionen entfernt werden können.

- **Sperrverfahren**

  Durch die Wahl der *Sperrverfahren* werden die Operationsarten erweitert, so dass auch das Erstellen von Sperroperationen möglich ist (siehe Abbildung 4.17). Außerdem wird der Test des *Schedulers* für das Überführen von Operationen aus dem *Input-* in den *Output-Schedule* angepasst, damit überprüft werden kann, ob Operationen ausgeführt werden können oder aufgrund von bestehenden Sperren wieder an das Ende des *Input-Schedules* gesetzt werden müssen. Das Verzögern von Operationen hat auch zur Folge, dass alle zur selben Transaktion gehörenden Operationen an das Ende des *Input-Schedules* versetzt werden müssen, um die korrekte Ausführungsreihenfolge beizubehalten. Wird durch den *Scheduler* eine Sperr- oder Entsperroperation aus dem *Input-Schedule* gelesen, so werden diese Sperr- beziehungsweise Entsperroperationen ausgeführt, sofern auf dem beteilig-

ten Objekt noch keine konkurrierende Sperre zu finden ist. Das Testen, Setzen und Entfernen von Sperren kann dabei auf zwei verschiedenen Wegen erfolgen. Ist als Sperrort *Amazon S3* (*Sperren-auf-S3-Objekte*) ausgewählt worden, so werden die Sperrinformationen als Metadaten an die Objekte übergeben und ausgelesen. In dem Fall, dass eine *lokale Sperrliste* geführt werden soll, wird diese als Hilfsstruktur angelegt und für jedes Objekt die aktuellen Sperren notiert.

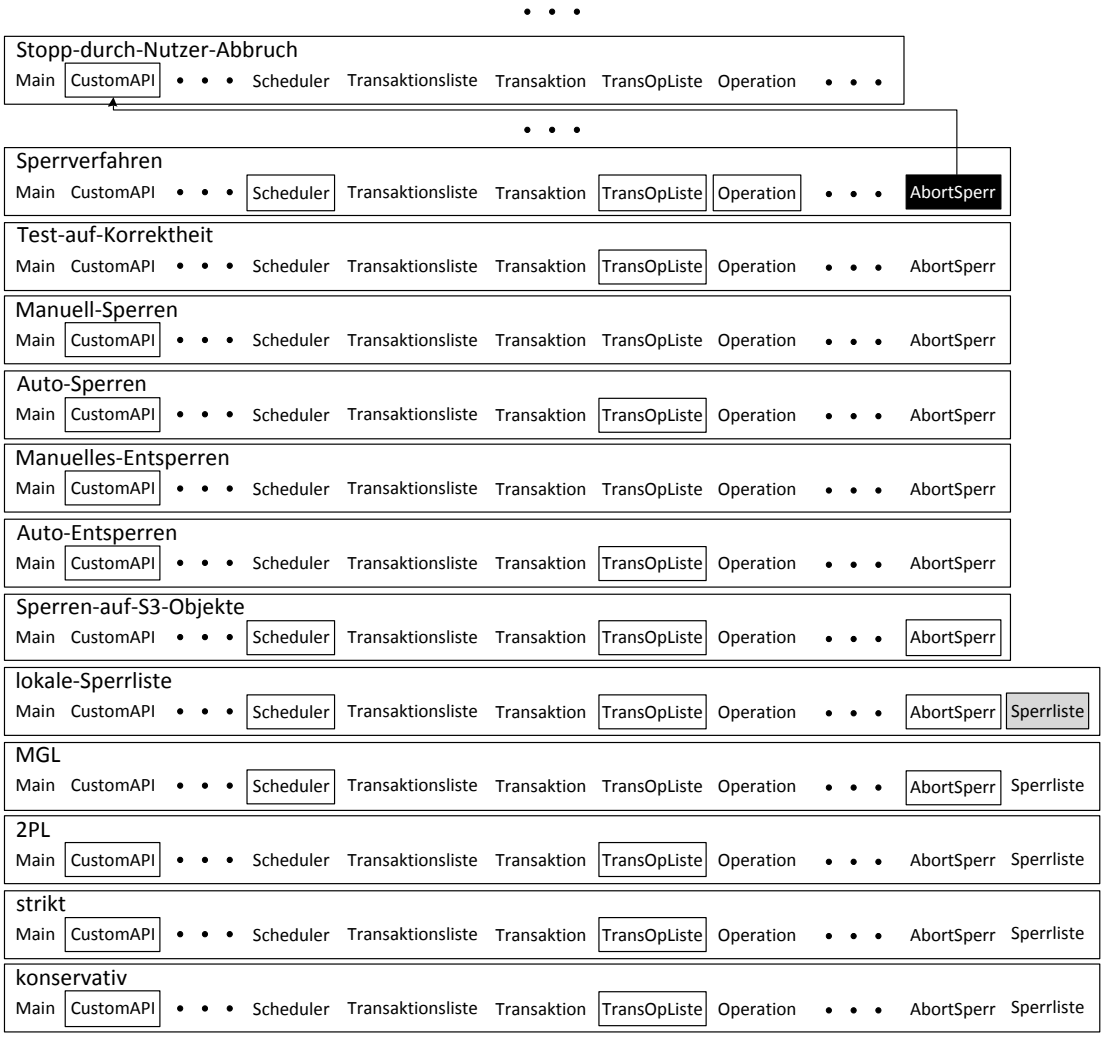

Abbildung 4.17: Kollaborationsdiagramm: Sperrverfahren

Das *Sperren* und *Entsperren* kann des Weiteren sowohl *manuell* durch den Entwickler veranlasst oder *automatisch* ausgeführt werden. Wenn der Entwickler die Sperren setzen möchte, werden ihm die zusätzlichen Funktionen *setLockToObject* und *deleteLockFromObject* durch die *CustomAPI* zur Verfügung gestellt. Damit das manuelle Sperren und Entsperren jedoch den Regeln des ausgewählten Sperrverfahrens genügt, wird für die Operationenliste ein *Test für die Korrektheit* benötigt, der beim Setzen und Entfernen von Sperren sowie beim Ausführen von normalen Operationen testet, ob die Sperrprotokolle eingehalten wurden. Beim automatischen Sperren hingegen werden Sperr- und Entsperroperationen erst hinzugefügt, wenn der Entwickler ein *commitTransaktion* aufruft. Auch die

Operationen *put*, *get* und *delete* werden beim Aufruf nicht in den *Input-Schedule* übernommen, sondern nur in die Operationenliste der jeweiligen Transaktion. Sobald ein *commitTransaktion* aufgerufen wird, wird die Operationenliste um die Sperr- und Entsperroperationen ergänzt.

Beim Zwei-Phasen-Sperr-Protokoll (*2PL*) werden Sperroperationen vor die Lese- und Bearbeitungsoperationen eines Objektes gesetzt, sofern nicht vorher schon die gleiche Sperroperation für dieses Objekt gesetzt wurde und Entsperroperationen hinter das letzte Auftreten von Operationen für ein Objekt. Bei den *strikten* und *konservativen* Verfahren wird das Einfügen von Sperr- und Entsperroperationen modifiziert, so dass zum Beispiel alle Entsperroperationen an das Ende der Operationenliste beziehungsweise alle Sperroperationen an den Anfang der Liste hinzugefügt werden. Gegebenenfalls muss dazu die Liste einmal durchgegangen werden, um herauszufinden, für welche Objekte Sperren gesetzt werden müssen. Sobald alle Sperr- und Entsperroperationen hinzugefügt wurden, werden die Operationen der Operationenliste in den *Input-Schedule* übernommen. Bei der Erweiterung durch das *Multi Granularity Locking* werden die Sperren zusätzlich auf die *Buckets* erweitert, so dass beim Ausführen der Sperroperationen auf die Objekte auch gleichzeitig das dazugehörige *Bucket* mit einer itentionalen Sperre versehen und beim Konflikttest abgefragt wird. Zum Entfernen aller Sperrinformationen beim Abbruch einer Transaktion werden sowohl die Funktion *abortTransaktion* als auch der *Scheduler* entsprechend angepasst.

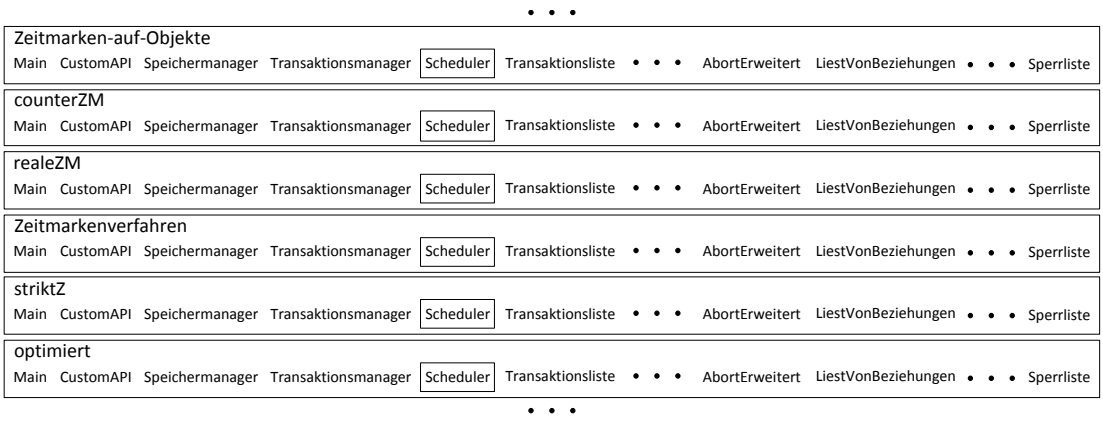

Abbildung 4.18: Kollaborationsdiagramm: Zeitmarkenverfahren

- **Zeitmarkenverfahren**

  Die *Zeitmarkenverfahren* bedingen, dass die Zeitmarkeninformationen der Transaktionen an die *S3*-Datenobjekte (*Zeitmarken-auf-Objekte*) weitergegeben werden können (siehe Abbildung 4.18). Dazu wird, sobald eine Operation vom *Input-* in den *Output-Schedule* überführt wird, die *BOT*-Zeitmarke der Transaktion als Metainformation an das Datenobjekt übergeben, welches durch die Operation angesprochen werden soll. Dabei wird je nach Art der Operation entweder die Lesezeitmarke oder die Schreibzeitmarke des Datenobjektes gesetzt. Da die *BOT*-Zeitmarke durch einen Zähler (*counterZM*) oder als reale Zeitmarke (*realeZM*) umgesetzt werden kann, müssen auch die Zeitmarkeninformationen für die Datenobjekte entsprechend realisiert werden. Mit Hilfe der Zeitmarken lässt sich durch das Zeitmarkenverfahren dann ermitteln, ob eine Operation ausgeführt werden

darf. Besteht kein Konflikt zwischen der *BOT*-Zeitmarke der Transaktion, die zu der Operation gehört, welche vom *Input-* in den *Output-Schedule* überführt werden soll, und der aktuellen Zeitmarke des zu bearbeitenden Datenobjektes, wird die Operation in den *Output-Schedule* überführt. Besteht jedoch ein Konflikt, wird die entsprechende Transaktion abgebrochen. Der Algorithmus zum Erkennen dieser Konflikte kann dabei durch das *strikte* und das *optimierte* Zeitmarkenverfahren modifiziert werden, um zum Beispiel kaskadierende Abbrüche zu vermeiden oder eine größere Menge an *Schedules* zu erlauben.

- **Multiversion Concurrency Control**

  Durch die *Multiversion Concurrency Control* werden alle Synchronisationsverfahren insofern erweitert, dass Leseoperationen immer erlaubt sind und nur Schreiboperationen mit einem der anderen Synchronisationsverfahren koordiniert werden müssen. Die Leseoperationen greifen dabei auf die jeweils aktuelle Kopie des Datenobjektes zu. Diese Kopien können entweder durch *Amazon S3* automatisch angelegt (*S3-Versionen*) oder in einer *lokalen Versionenliste* geführt werden (siehe Abbildung 4.19).

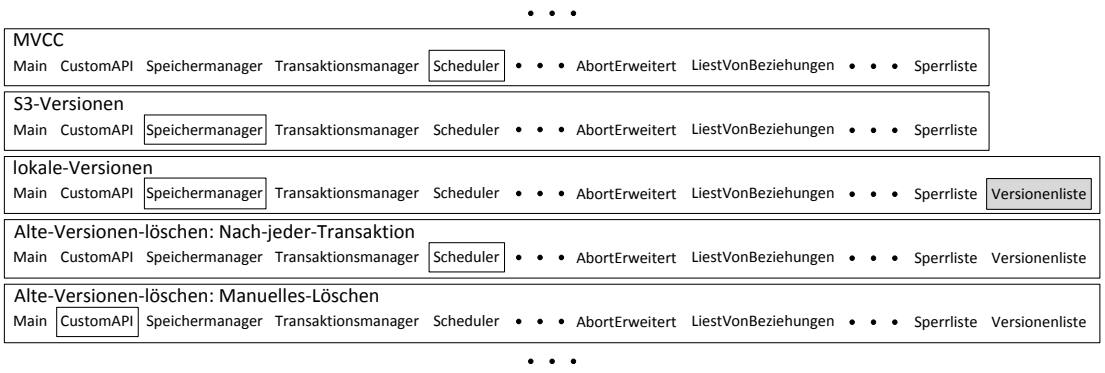

Abbildung 4.19: Kollaborationsdiagramm: *Multiversion Concurrency Control*

Um die Anzahl der Kopien nicht übermäßig stark ansteigen zu lassen, können alte Versionen, die nicht mehr benötigt werden entsprechend aus *Amazon S3* oder aus der lokalen Versionenliste entfernt werden. Das Löschen der Versionen ist zum einen durch den *Scheduler* möglich (*Alte-Versionen-löschen: Nach-jeder-Transaktion*) und zum anderen durch den Aufruf einer Funktion durch den Entwickler (*Alte-Versionen-löschen: Manuelles-Löschen*). Für den Entwickler wird zum Entfernen von Versionen die Funktion *alteVersionenLöschen* bereitgestellt. Wenn jedoch der *Scheduler* die Versionen automatisch nach jeder Transaktion löschen soll, geschieht das, sobald eine *EOT*-Operation durch den *Scheduler* ausgelesen wird.

4.4 Variable Merkmale des Speichermanagers

In den nachfolgenden Abschnitten werden die Klassen, Verfeinerungen und Aspekte näher erläutert, welche zu den variablen Merkmalen des Speichermanagers gehören. Dabei werden die Merkmale, welche in den vorhergehenden Abschnitten schon erläutert wurden, nicht noch einmal aufgegriffen. Ein Beispiel für ein solches Merkmal ist der

Transaktionspuffer, welcher zwar zum Speichermanager gehört, jedoch ausschließlich für die optimistische Synchronisation benötigt wird.

4.4.1 Log

Das *Log* ist Bestandteil des *Recoverymanagers*, der im Speichermanager initialisiert wird (siehe Abbildung 4.20). Wenn der Speichermanager eine Operation ausführt, die er vom *Output-Schedule* des *Schedulers* bekommt, wird die Operation zusätzlich ins Log geschrieben. Dabei wird zwischen *BOT*-, *EOT*- und normalen Einträgen unterschieden. Je nach Art des Eintrags müssen unterschiedliche Informationen zum Log hinzugefügt werden, wobei einige Informationen wie zum Beispiel die Transaktions-ID der Transaktion, durch die der Eintrag entstanden ist, immer benötigt werden. Je nachdem, ob es *undo*- und *redo*-Informationen (*Undo-Log*, *Redo-Log*) gibt und je nachdem, ob es physische (*Before-Image*, *After-Image*) oder logische (*Redo-Info*, *Undo-Info*) Wiederherstellungsinformationen sind, müssen diese ebenfalls entsprechend ins Log eingetragen werden.

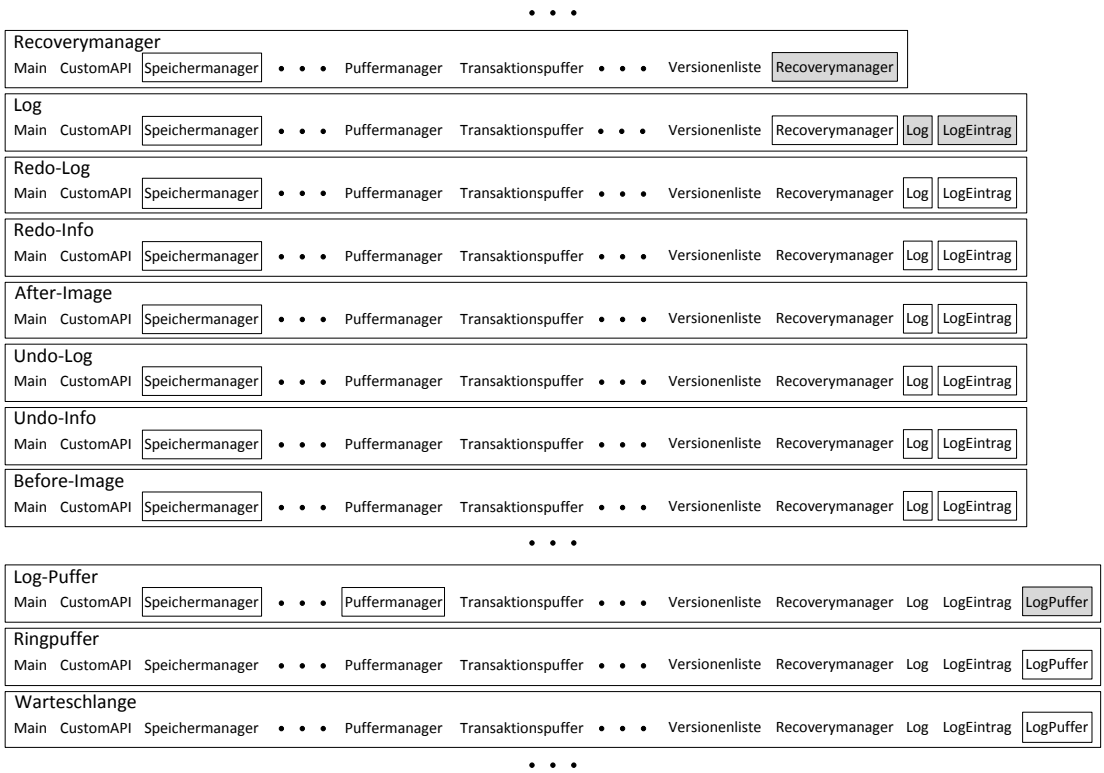

Abbildung 4.20: Kollaborationsdiagramm: Log und Log-Puffer

Da zum Zwecke des kontinuierlichen Schreibens von Logeinträgen auf den persistenten Speicher ein Log-Puffer verwendet werden muss, wird dieser im Puffermanager angelegt, so dass die Einträge zunächst in den Puffer und von dort aus in regelmäßigen Abständen beziehungsweise abhängig von zusätzlichen Bedingungen, beispielsweise dem WAL-Prinzip, in den persistenten Speicher geschrieben werden. Der *Log-Puffer* kann dabei entweder als *Ringpuffer* oder als *Warteschlange* umgesetzt werden, was dazu führt, dass das Hinzufügen, Halten und Entfernen von Einträgen in und aus dem Puffer entsprechend erweitert werden muss.

4.4.2 Datenpuffer

Sollen Daten nicht direkt in den persistenten Speicher geschrieben, sondern vorher in den Puffer gelesen und dort verändert werden, muss der Puffermanager um einen solchen Datenpuffer (*DB-Puffer*) erweitert werden (siehe Abbildung 4.21). Dabei wird statt der direkten Ausführung von Operationen durch den Speichermanager eine Objektkopie im Puffer angelegt und die Operation auf dieser Pufferkopie durchgeführt. Wenn das jeweilige Objekt in den persistenten Speicher zurückgeschrieben werden muss oder durch ein anderes Objekt ersetzt wird, weil der Puffer voll ist, wird das jeweilige Objekt aus dem Puffer entfernt.

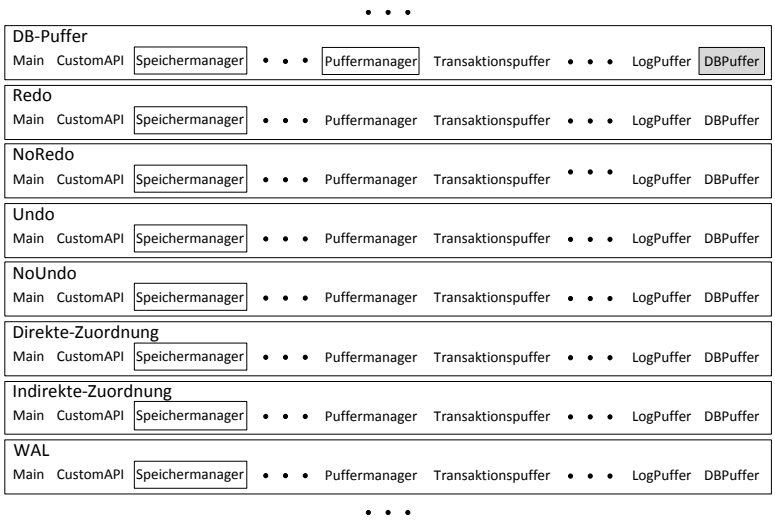

Abbildung 4.21: Kollaborationsdiagramm: Datenpuffer

Für alle Objektkopien im Puffer muss bekannt sein, durch welche Transaktionen sie verändert wurden, da zum Beispiel bei der *NO-UNDO*-Seitenersetzungsstrategie nur Objekte im Puffer ersetzt werden dürfen, deren Transaktionen ein *Commit* aufgeführt haben. Die Seitenersetzungs- und Propagierungsstrategie erweitern dabei die Überführung von Daten aus dem persistenten Speicher in den Puffer und umgekehrt, so werden zum Beispiel bei der *NO-REDO*-Propagierungstrategie beim *Commit* einer Transaktion alle Objekte, die durch diese Transaktion verändert wurden, aus dem Puffer in den persistenten Speicher geschrieben. Die Einbringstrategien hingegen verändern das Zurückschreiben, so dass durch die *direkte Zuordnung* die persistenten Daten ersetzt werden beziehungsweise durch die *indirekte Zuordnung* mehrere Kopien angelegt und die jeweils älteste Kopie ersetzt wird. Zusätzlich lässt sich das *WAL*-Prinzip aktivieren, wodurch weitere Änderungen der Ausführungsfunktion des Speichermanagers durchgeführt werden müssen. So erfolgt ein Ausschreiben der Logeinträge spätestens dann, wenn eine Transaktion ein *Commit* durchführen will oder ein Objekt vom Puffer in den persistenten Speicher überführt wird.

4.4.3 Sicherungspunkte

Wenn *Sicherungspunkte* angelegt werden sollen, müssen diese in definierten Abständen angemeldet werden, so dass der Speichermanager die Ausführung von Operationen nur noch solange ausführt, bis die Bedingungen für die jeweilige Art des Setzens von Sicherungspunkten erfüllt wurden (siehe Abbildung 4.22). Anschließend wird gegebenen-

falls die Ausführung von weiteren Operationen ausgesetzt, bis das jeweilige Sicherungspunktverfahren den Sicherungspunkt angelegt und alle Pufferdaten persistent gemacht hat, die das jeweilige Verfahren bedingt. Bei den *unscharfen* Sicherungspunkten werden zusätzlich die *Dirty-Page*-Listen angelegt und die Transaktions-ID gesichert, durch die diese unsauberen Daten erzeugt wurden. Sicherungspunkte benötigen des Weiteren einen eigenen Eintrag im Log, damit bei der Wiederherstellung nicht das gesamte Log eingelesen werden muss. Zusätzlich ermöglichen Sicherungspunkte zum Beispiel auch, dass Logeinträge ermittelt werden können, die nicht mehr benötigt und somit gelöscht werden können (*Alte-Log-Einträge-löschen*), falls das dazugehörige Merkmal aktiviert wurde.

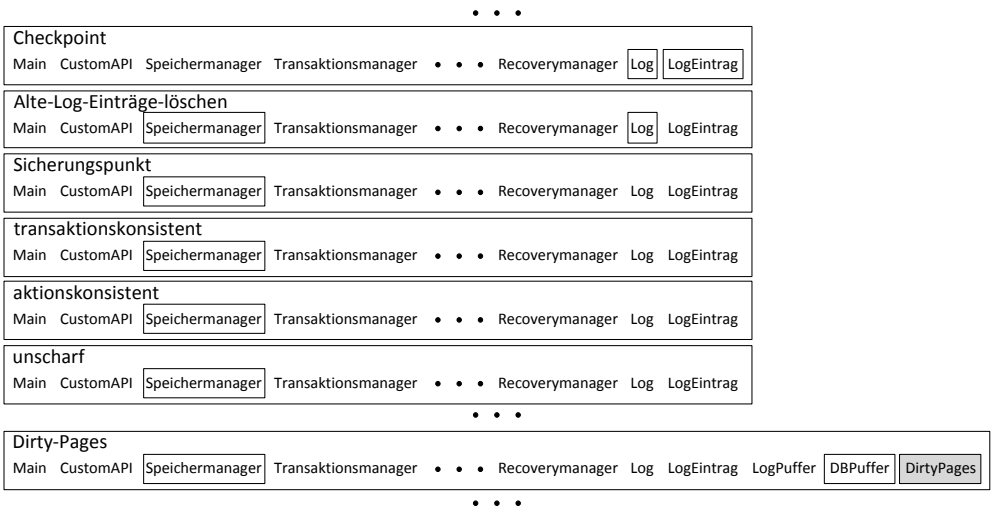

Abbildung 4.22: Kollaborationsdiagramm: Sicherungspunkte

4.4.4 Recovery-Verfahren

Durch die *Recovery*-Verfahren wird der Speichermanager um die Möglichkeit erweitert, eine Systemwiederherstellung durchzuführen, wenn ein Systemfehler aufgetreten ist (siehe Abbildung 4.23). Dazu wird die Wiederherstellung mit Hilfe des ausgewählten *Recovery*-Verfahrens durchgeführt und erst im Anschluss mit der normalen Ausführung von Operationen fortgefahren. Wurde zum Zwecke der Wiederherstellung ein Log angelegt, werden auch Sicherungspunkte in den Prozess der Wiederherstellung miteinbezogen. Existieren keine Sicherungspunkte, wird der Loganfang automatisch als erster Sicherungspunkt interpretiert. Je nachdem, ob ein *Undo* beziehungsweise *Redo* durchgeführt werden muss, wird zunächst alles Fehlerhafte und Unvollständige rückgängig gemacht und anschließend alles wiederholt, was notwendig ist, um den aktuellen Systemzustand wiederherzustellen. Da das *UNDO/NO-REDO*-Verfahren durch die *MVCC-Versionierung* erweitert werden kann, muss für dieses Verfahren angepasst werden, von wo die *Before-Images* ausgelesen werden.

Alternativ können auch *ARIES* oder die *Schattenspeicherverfahren* ausgeführt werden. Für das ARIES-Verfahren muss während der Ausführung von Operationen für jede Transaktion die ID des letzten durch diese Transaktion angelegten Logeintrags (*lastLSN*) gespeichert und jedem persistent gespeicherten Objekt ein Verweis auf den Logeintrag der letzten Änderung dieses Objektes mitgegeben werden. Beim Ausfüh-

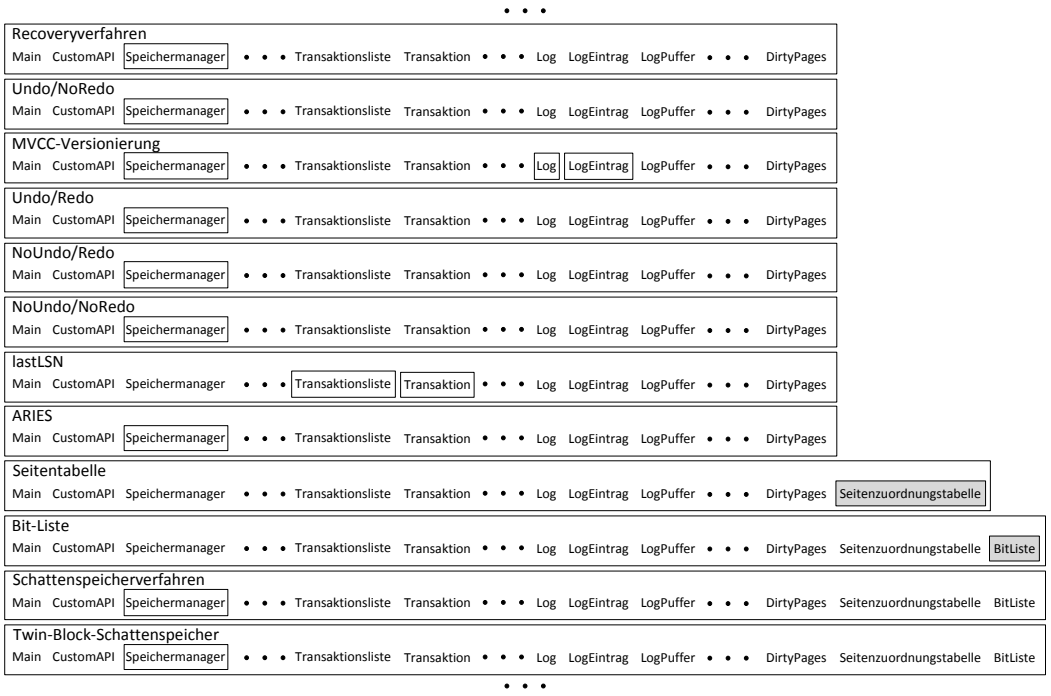

Abbildung 4.23: Kollaborationsdiagramm: *Recovery*-Verfahren

ren der Wiederherstellung mit der Hilfe von ARIES werden diese Informationen und zum Beispiel auch die *Dirty-Page*-Listen miteinbezogen. Im Falle der Verwendung des Schattenspeicherverfahrens hingegen wird, während der Ausführung von Operationen durch den Speichermanager, eine Schattenkopie der Objekte angelegt und zusätzlich Seitenzuordnungstabellen gehalten, um sowohl die Originale als auch die Kopien nutzen zu können. Führt der Speichermanager ein *Commit* einer Transaktion aus, dann werden die Originaltabelle und die Schattenspeichertabelle ausgetauscht, so dass bei einer Wiederherstellung die alten Daten ausgelesen werden können. Ist die Erweiterung des Schattenspeicherverfahrens aktiv, werden statt Seitenzuordnungstabellen Bitlisten angelegt, die markieren, welches *Bucket* die Originaldaten und welches die Schattendaten enthält.

4.5 Zusammenfassung

Auf der Grundlage des Merkmalsdiagramms aus Kapitel 3 wurde in diesem Kapitel ein Kollaborationsdiagramm zur Visualisierung der benötigten Aspekte und Verfeinerungen entworfen. In Tabelle 4.2 und Tabelle 4.3 werden die Verfeinerungen und in Tabelle 4.1 die Aspekte noch einmal als Übersicht zusammengefasst, wobei die Zeilen mit den Merkmalen gleichzusetzen sind und die farblichen Markierungen der jeweils zweiten Spalte den Vergleich mit den korrespondierenden Merkmalen aus dem Merkmalsdiagramm vereinfachen sollen.

Hinsichtlich der Erläuterungen zum Kollaborationsdiagramm wurde zunächst auf die Basisklassen eingegangen, die zum einen als Einstiegspunkt für die zu entwickelnde Anwendung dienen und zum anderen vordefinierte Funktionen zur Kommunikation mit *Amazon S3* liefern. Des Weiteren wurden die Kernmerkmale vorgestellt. Diese Kernmerkmale stellen die grundlegenden Klassen und Funktionen bereit, um eine Trans-

aktionsunterstützung in eine Anwendung integrieren zu können. Dazu werden durch die Aktivierung der Transaktionsunterstützung beispielsweise Klassen für Operationen, Transaktionen sowie den Transaktionsmanager, den Scheduler und den Speichermanager erstellt und die vordefinierten Funktionen *put*, *get* und *delete* der *CustomAPI* modifiziert, so dass die Ausführung von Operationen auf *Amazon S3* unter der Verwendung der neuen Klassen und der neuen Funktionalität erfolgt. Dazu gehört zum Beispiel, dass Operationen erst ausgeführt werden dürfen, wenn der *Scheduler* festgestellt hat, dass die Operationen den fehlerfreien Ablauf des Systems nicht stören.

Zusätzlich werden weitere Klassen und Funktionalitäten durch die variablen Merkmale bereitgestellt. Da diese jedoch nur unter bestimmten Bedingungen auswählbar sind, erweitern sie die Anwendung dementsprechend nur dann, wenn sie aktiviert wurden. Wird beispielsweise das Zeitmarkenverfahren ausgewählt, werden dessen Verfeinerungen und Aspekte in die Anwendung integriert, aber die Schichten der alternativen Synchronisationsverfahren deaktiviert. Das Kollaborationsdiagramm unterstützt somit eine mögliche Umsetzung der Merkmale, indem kenntlich gemacht wird, welche Aspekte und Verfeinerungen implementiert werden müssen und welche Klassen sie beeinflussen.

Zusammen mit dem Merkmalsdiagramm wird verdeutlicht, welche Merkmale gleichzeitig aktiv sein können und inwiefern sie dieselben Klassen erweitern. Dazu werden im folgenden Kapitel die Ergebnisse und Erkenntnisse, die sich aus dem Merkmalsdiagramm und dem Kollaborationsdiagramm ergeben, zusammengefasst und aufgetretene Probleme und mögliche Lösungsmöglichkeiten vorgestellt.

		AbortKomp	AbortKompLog	AbortErweitert	AbortKask	AbortWart	AbortSGT	AbortOptiSync	AbortSperr
Durch-Nutzer		X							
Durch-Log			X						
Erweitert				X					
Entkoppelte-Subtransaktionen				X					
Sagas				X					
Flexible				X					
Geschlossene				X					
Offene				X					
Ignoriere-nicht-vitale				X					
Abbruch-Vatertransaktion				X					
Neustart-ST				X					
Ersatztransaktionen				X					
Kaskadierende-Abbrüche					X				
Wartegraph						X			
SGT							X		
Optimistische-Synchronisation								X	
Sperrverfahren									X
Sperren-auf-S3-Objekte									X
Lokale-Sperrliste									X
MGL									X

Tabelle 4.1: Kollaborationsdiagramm: Aspekte

	Main	CustomAPI	Speichermanager	Transaktionsmanager	Scheduler	Transaktionsliste	Transaktion	TransOpListe	Operation	ListeVonBeziehungen	Wartegraph	Konfliktgraph	Puffermanager	Transaktionspuffer	Sperrliste	Versionenliste	Recoverymanager	Log	LogEintrag	LogPuffer	DBPuffer	DirtyPages	Seitenzuordnungstabelle	BitListe
Basis	X	X																						
Transaktionsunterstützung		X																						
Speichermanager		X	X																					
Transaktionsmanager		X		X																				
Einfach		X																						
Scheduler				X	X																			
Transaktionsliste				X		X																		
Transaktion						X	X																	
Trans-Zustand							X																	
Laufend							X																	
Committed							X																	
Abgebrochen							X																	
Trans-Op-Liste							X	X																
Operation								X	X															
Gestoppt							X																	
Verzögert							X																	
Stopp-durch-Nutzer-Abbruch		X																						
Stopp-nach-Fehler					X																			
Neustart-nach-Fehler					X																			
Stopp-nach-zu-vielen-Versuchen					X		X																	
Durch-Nutzer		X			X	X	X																	
Durch-Log					X																			
Erweitert		X			X	X	X																	
Entkoppelte-Subtransaktionen		X			X	X	X																	
Sagas		X			X																			
Flexible		X			X	X	X																	
Geschlossene		X			X																			
Offene		X			X																			
Nicht-vitale-Subtransaktionen		X				X	X																	
Ignoriere-nicht-vitale		X			X																			
Abbruch-Vatertransaktion					X																			
Neustart-ST					X																			
Ersatztransaktionen					X	X	X																	
TransZeitmarke		X				X	X																	
Counter		X				X	X																	
Real		X				X	X																	
Kaskadierende-Abbrüche				X	X					X														
Wartegraph				X	X						X													
Abbruch-nach-Deadlock					X																			
Neustart-nach-Deadlock					X																			
Alle-Transaktionen					X																			
Zyklus-auslösende-Transaktion					X																			
Transaktionslänge					X																			
Anzahl-aufgebrochener-Zyklen					X																			
Rücksetzaufwand					X																			
Seriell					X																			
SGT					X							X												
Topologisches-Sortieren												X												
Tiefensuche												X												
Puffermanager			X										X											
Transaktionspuffer			X										X	X										
Optimistische-Synchronisation			X		X	X	X																	
Vorwärts					X																			
Rückwärts					X																			
Sperrverfahren					X			X	X															
Test-auf-Korrektheit								X																
Manuell-Sperren		X																						
Auto-Sperren		X						X																
Manuelles-Entsperren		X																						
Auto-Entsperren		X						X																
Sperren-auf-S3-Objekte					X			X																
Lokale-Sperrliste					X			X							X									
MGL					X																			
2PL		X						X																
Strikt		X						X																
Konservativ		X						X																

Tabelle 4.2: Kollaborationsdiagramm: Verfeinerungen Teil 1

		Main	CustomAPI	Speichermanager	Transaktionsmanager	Scheduler	Transaktionsliste	Transaktion	TransOpListe	Operation	ListeVonBeziehungen	Wartegraph	Konfliktgraph	Puffermanager	Transaktionspuffer	Sperrliste	Versionenliste	Recoverymanager	Log	LogEintrag	LogPuffer	DBPuffer	DirtyPages	Seitenzuordnungstabelle	BitListe
Zeitmarken-auf-Objekte						X																			
CounterZM						X																			
RealeZM						X																			
Zeitmarkenverfahren						X																			
StriktZ						X																			
Optimiert						X																			
MVCC						X																			
S3-Versionen				X																					
Lokale-Versionen				X													X								
Nach-jeder-Transaktion						X																			
Manuelles-Löschen			X																						
Recoverymanager				X														X							
Log				X														X	X	X					
Redo-Log				X															X	X					
Redo-Info				X															X	X					
After-Image				X															X	X					
Undo-Log				X															X	X					
Undo-Info				X															X	X					
Before-Image				X															X	X					
Checkpoint																			X	X					
Alte-Log-Einträge-löschen				X															X						
Sicherungspunkt				X																					
transaktionskonsistent				X																					
aktionskonsistent				X																					
unscharf				X																					
Log-Puffer				X										X							X				
Ringpuffer																					X				
Warteschlange																					X				
DB-Puffer				X										X								X			
Redo				X																					
NoRedo				X																					
Undo				X																					
NoUndo				X																					
Direkte-Zuordnung				X																					
Indirekte-Zuordnung				X																					
WAL				X																					
Dirty-Pages				X																		X	X		
Recoveryverfahren				X																					
Undo/NoRedo				X																					
MVCC-Versionierung				X															X	X					
Undo/Redo				X																					
NoUndo/Redo				X																					
NoUndo/NoRedo				X																					
LastLSN							X	X																	
ARIES				X																					
Seitentabelle																								X	
Bit-Liste																									X
Schattenspeicherverfahren				X																					
Twin-Block-Schattenspeicher				X																					

Tabelle 4.3: Kollaborationsdiagramm: Verfeinerungen Teil 2

Kapitel 5

5 Ergebnisdiskussion

Im folgenden Kapitel wird erläutert, inwiefern die Entwicklung des Merkmalsdiagramms und des Kollaborationsdiagramms die modulare Integration von Transaktionsmechanismen in *Amazon S3* unterstützt und welche Probleme und Erkenntnisse während des Entwurfs der Modelle aufgetreten sind. Des Weiteren werden Lösungsansätze für aufgetretene Probleme vorgestellt. Dazu gehören beispielsweise das Erweitern von Parameterlisten und das optionale Weben sowie die Verwendung von *Hook*-Methoden und des *Aspect Refinements*.

5.1 Merkmalsdiagramm einer Transaktionsunterstützung

Das Merkmalsdiagramm ist das Ergebnis einer umfangreichen Merkmalsanalyse der Kernmerkmale und variablen Eigenschaften einer Transaktionsunterstützung. Das Ziel ist, eine große Auswahl an Synchronisationsverfahren und *Recovery*-Maßnahmen anbieten zu können und gleichzeitig eine sinnvolle Auswahl zu garantieren. So besteht das Merkmalsdiagramm aus 169 Merkmalen, wovon 54 abstrakt sind und nicht implementiert werden müssen (siehe Tabelle 5.1).

Merkmale		169 (100 %)
	Primitiv	Zusammengesetzt
Abstrakt	19 (11 %)	35 (21 %)
Konkret	80 (47 %)	35 (21 %)

cross-tree-Beschränkungen	56

Tabelle 5.1: Unterteilung: Merkmalsdiagramm

Des Weiteren lassen sich die Merkmale in 99 primitive und 70 zusammengesetzte Merkmale unterteilen. Zu den zusammengesetzten Merkmalen gehört beispielsweise der *Scheduler*. Dieser bietet eine Grundfunktionalität an, wird aber zusätzlich durch die Synchronisationsverfahren erweitert. Zur Unterstützung von korrekten Zusammenstellungen bei der Konfiguration der Transaktionsunterstützung wurde das Modell zusätzlich um 56 *cross-tree*-Beschränkungen ergänzt. Diese stellen sicher, dass beispielsweise durch die Aktivierung des Zeitmarkenverfahrens auch alle benötigten Hilfsmerkmale wie Transaktionszeitmarken und Zeitmarken auf den Datenobjekten aktiviert werden. Zusätzlich werden Merkmalskombinationen ausgeschlossen, die keine Vorteile bringen,

aber die Effektivität des Systems verringern würden. So wird beispielsweise die Aktivierung des Wartegraphen nur dann erlaubt, wenn ein Synchronisationsverfahren *Deadlocks* auslösen kann und der Wartegraph somit auch wirklich benötigt wird. Die *cross-tree*-Beschränkungen erstrecken sich dabei über das gesamte Merkmalsdiagramm.

Die 115 konkreten Merkmale teilen sich des Weiteren in ein obligatorisches Basismerkmal, welches gleichzeitig die Wurzel des Merkmalsdiagramms ist und die Basisklassen bereitstellt, 13 Kernmerkmale und 101 variable Merkmale auf (siehe Tabelle 5.2), wobei zwischen 25 optionalen variablen Merkmalen und 76 alternativen variablen Merkmalen unterschieden werden kann. Optionale Merkmale können deaktiviert werden, wohingegen von alternativen Merkmalen mindestens ein Merkmal aktiviert werden muss, was zum Beispiel die Synchronisationsverfahren zu obligatorischen Merkmalen macht, sofern die Transaktionsunterstützung aktiviert wurde, obwohl jedes für sich optional ist. Die Kernmerkmale werden dahingegen immer aktiviert, sobald die Transaktionsunterstützung durch den Nutzer ausgewählt wird. Sie bilden die Grundlage für eine funktionstüchtige Transaktionsunterstützung und für mögliche Erweiterungen durch die variablen Merkmale. Das Basismerkmal stellt ein spezielles Kernmerkmal dar, da es immer aktiv ist, selbst wenn die Transaktionsunterstützung nicht ausgewählt wurde, da es Funktionalität bereitstellt, die ohne die Transaktionsunterstützung verwendet werden kann. Zu dieser Funktionalität gehören beispielsweise die vordefinierten Funktionen zur Kommunikation mit *Amazon S3*.

Konkrete Merkmale	115	100 %
Basismerkmale	1	1 %
Kernmerkmale	13	11 %
Variable Merkmale	101	88 %

Tabelle 5.2: Unterteilung: Konkrete Merkmale

5.2 Kollaborationsdiagramm

Aufbauend auf dem Merkmalsdiagramm wurde das Kollaborationsdiagramm entwickelt, welches spezifiziert, inwiefern Klassen erstellt, verfeinert und um Aspekte ergänzt werden könnten. Dabei steht das Kollaborationsdiagramm in direkter Verbindung zum Merkmalsdiagramm, aufgrund der Tatsache, dass die konkreten Merkmale des Merkmalsdiagramms die einzelnen Schichten des Kollaborationsdiagramms darstellen. Werden Merkmale aus dem Merkmalsdiagramm ausgewählt beziehungsweise abgewählt, werden die korrespondierenden Schichten ein- beziehungsweise ausgeblendet, was dafür sorgt, dass nur die Verfeinerungen und Aspekte in Kraft treten, die für die aktuelle Konfiguration benötigt werden. In Verbindung mit den *cross-tree*-Beschränkung wird des Weiteren auch sichergestellt, dass alle Merkmale aktiv werden, die Klassen und Eigenschaften einführen, auf die durch andere Klassen anderer Merkmale zugegriffen wird. Ein Beispiel für einen solchen Fall bietet das *ARIES*-Verfahren, welches ein um die *lastLSN* erweitertes Transaktionsobjekt benötigt. Sobald der Ablauf des *ARIES*-Verfahrens durch das *ARIES*-Merkmal integriert wird, setzt die Abarbeitung des Verfahrens voraus, dass es auf die *lastLSN*-Eigenschaft der Transaktion zugreifen kann. Dies wird nur gewährleistet, wenn durch die Aktivierung des *ARIES*-Verfahrens auch gleichzeitig das *lastLSN*-Merkmal aktiviert und somit die korrespondierende Schicht

im Kollaborationsdiagramm eingebunden wird, wodurch im generierten Quellcode beide Merkmale miteinander interagieren können.

Insgesamt besteht das Kollaborationsdiagramm aus 24 verschiedenen Klassen, wobei 2 durch die Basisschicht, 7 durch die Kernmerkmale und 15 durch die variablen Merkmale eingeführt werden (siehe Tabelle 5.3). Außerdem erfolgen zusätzliche Erweiterungen durch 8 Aspekte.

Klassen	24	100,00 %
Basisklassen	2	8 %
Zusätzliche Klassen der Kernmerkmale	7	29 %
Zusätzliche Klassen der variablen Merkmale	15	63 %
Klassen mit Verfeinerungen	16	67 %
Klassen ohne Verfeinerungen	8	33 %

Aspekte	8	100 %
Aspekte ohne Verfeinerungen	6	75 %
Aspekte mit Verfeinerungen	2	25 %

Tabelle 5.3: Unterteilung: Klassen und Aspekte

Die Klassen und Aspekte werden 187 mal verfeinert, wobei 41 Verfeinerungen die *Scheduler*-Klasse, 37 den Speichermanager, 25 die *CustomAPI* und 19 Verfeinerungen die Transaktionsklasse betreffen (siehe Tabelle 5.4). Das bedeutet, dass die *Scheduler*-Klasse von 36 %, der Speichermanager von 32 %, die *CustomAPI* von 22 % und die Transaktionsklasse von 17 % der konkreten Merkmale verfeinert werden. 8 Klassen und 6 Aspekte werden gar nicht verfeinert, so dass sich die übrigen 65 Verfeinerungen in 12 Verfeinerungen von 2 Aspekten und 53, die restlichen 12 Klassen betreffende Verfeinerungen aufteilen.

	Aufteilung der Verfeinerungen		An den Verfeinerungen beteiligte konkrete Merkmale
	187	100 %	
Aspektverfeinerungen	12	7 %	10 %
Verfeinerungen der *Scheduler*-Klasse	41	22 %	36 %
Verfeinerungen des Speichermanagers	37	20 %	32 %
Verfeinerungen der *CustomAPI*	25	13 %	22 %
Verfeinerungen der Transaktionsklasse	19	10 %	17 %
Sonstige Verfeinerungen	53	28 %	

Tabelle 5.4: Aufteilung der Verfeinerungen auf Aspekte und Klassen

Durch diese Verteilung der Verfeinerungen auf die einzelnen Klassen wird ersichtlich, dass wenige Klassen sehr häufig durch unterschiedliche Merkmale erweitert werden, wohingegen vielen Klassen selten beziehungsweise überhaupt nicht modifiziert werden. Aus dieser Häufung von Verfeinerungen ein und derselben Klasse ergibt sich ein komplexer Vorgang zum Integrieren mehrerer Merkmale in dieselbe Klasse. Unter Umständen kann diese Komplexität dazu führen, dass sich bei der Umsetzung der konfigurierbaren

Transaktionsunterstützung auf der Grundlage des Kollaborationsdiagramms zusätzliche Klassen, Aspekte oder Verfeinerungen ergeben, die von Vorteil sein könnten, aber nicht durch das Kollaborationsdiagramm abgedeckt werden.

5.3 Probleme und Erkenntnisse

Während der Modellierung des Merkmalsdiagramms und des Kollaborationsdiagramms wurde deutlich, dass eine darauf aufbauende Umsetzung der Transaktionsunterstützung an einigen Punkten auf Komplikationen treffen wird. Zu diesen Komplikationen gehören beispielsweise das Erweitern von Übergabewerten verwendeter Funktionen, das Erweitern von optionalen Elementen und das zeilenweise Verändern von Algorithmen. Für jedes dieser Probleme gibt es bereits Lösungsansätze, die im Folgenden kurz erläutert werden.

5.3.1 Erweitern von Parameterlisten

Beim Erstellen von Transaktionen durch die Funktion *beginTransaktion* werden keine zusätzlichen Informationen benötigt. Erst durch Erweiterungen wie das Anlegen von kompensierenden Transaktionen durch den Nutzer oder von Subtransaktionen, welche beispielsweise Eigenschaften wie vital und nicht-vital erhalten können, werden weitere Übergabewerte für die Funktion *beginTransaktion* notwendig. Da sowohl Verfeinerungen als auch Aspekte nur zusätzlichen Code vor, um oder nach bestehenden Quellcode integrieren können und der Originalcode somit erhalten bleibt, können Methodensignaturen nicht einfach erweitert werden. Eine Lösungsmöglichkeit besteht darin, ein Übergabeobjekt zu kreieren, dessen benötigte Eigenschaften vor dem Funktionsaufruf gesetzt werden und nach Bedarf innerhalb der Funktion ausgelesen werden können. So muss bei Erweiterungen nur das Setzen und Auslesen von Eigenschaften dieses Objektes modifiziert werden, um zusätzliche Informationen übergeben zu können. Nachteilig an diesem Verfahren ist jedoch, dass der Entwickler, welcher die Transaktionsunterstützung nutzen möchte, zusätzlichen Aufwand im Erstellen und Übergeben dieser Objekte hat.

5.3.2 Optionales Weben

Ein weiteres Problem zeigt sich beim Verfeinern von Funktionen und Eigenschaften, die erst durch optionale Merkmale integriert werden. Im bestehenden Modell zur Transaktionsunterstützung betrifft dies vor allem die Einführung der Funktion *abortTransaktion*. Diese existiert genau dann, wenn das Merkmal zum Ermöglichen des manuellen Funktionsabbruchs durch den Nutzer aktiv ist. Daraus folgt, dass andere Merkmale, welche die *CustomAPI* verfeinern wollen, alle immer in der *CustomAPI* vorhandenen Funktionen ohne Probleme erweitern können, die Funktion *abortTransaktion* jedoch nur dann erweitert werden darf, wenn sie auch wirklich vorhanden ist. Unter Verwendung der merkmalsorientierten Programmierung ist das Erweitern von optionalen Merkmalen problematisch, wenn Merkmale versuchen mit nicht aktivierten optionalen Merkmalen zu interagieren beziehungsweise sie zu verfeinern [Liu et al., 2005]. Durch die Verwendung von *Aspectual Feature Modules* können jedoch zusätzlich Aspekte verwendet werden, so dass das optionale Weben als Lösungsmöglichkeit zur Verfügung steht [Leich et al., 2005a; Kästner, 2007; Adler, 2010]. Zum Erweitern der Funktion *abortTransaktion* wird somit ein Aspekt für das jeweilige Merkmal angelegt, welcher die Modifikation

der Funktion enthält und nur dann angewendet wird, wenn die Funktion und somit der *Joinpoint* existiert. Das bedeutet jedoch auch, dass für jedes weitere Merkmal, welches die Funktion *abortTransaktion* erweitern möchte, ein separater Aspekt angelegt werden muss, um die Verfeinerungen zu unterstützen.

5.3.3 Hook-Methoden

Damit Algorithmen nicht vollständig überschrieben werden müssen um beispielsweise das Zeitmarkenverfahren zu einem optimierten Zeitmarkenverfahren zu machen, obwohl nur wenige Zeilen abzuändern wären, ist die Einführung von *Hook*-Methoden notwendig [Alwis et al., 2000; Eaddy und Aho, 2006]. Die Modifikationen können so auch innerhalb von Algorithmen angewendet werden, was mit dem Einbinden von Änderungen vor, nach oder um einen bestehenden Quellcode nicht möglich ist. Auch für die Unterstützung der Kooperation zwischen verschiedenen zusammenarbeitenden Merkmalen sind *Hook*-Methoden hilfreich. Während der Umsetzung muss jedoch bekannt sein, an welchen Stellen solche *Hook*-Methoden verwendet werden müssen, damit diese frühestmöglich angelegt werden und Erweiterungen diese anschließend modifizieren können (*Preplanning Problem [Czarnecki, 1998]*). Gegebenenfalls sind zunächst sogar leere Methoden notwendig, welche erst durch das Einbinden von weiteren Merkmalen Funktionalität erhalten. Die Erweiterung von *Hook*-Methoden ist dabei sowohl durch Verfeinerungen als auch durch Aspekte möglich.

5.3.4 Aspect Refinement

Die für das optionale Weben eingeführten Aspekte benötigen eine Möglichkeit, wie normale Klassen erweitert zu werden. So wird zum Beispiel der Aspekt des Merkmals für die erweiterten Transaktionsmodelle zur Erweiterung der Funktion *abortTransaktion* durch jede Form der erweiterten Transaktionsmodelle modifiziert, um die Folgeaktionen beim Abbruch von Transaktionen anzupassen. Da vor allem der *Advicecode* verfeinert werden muss, ist die Verwendung von benamten Versionen vom *Advice* von Vorteil [Apel et al., 2007b]. Ein benamter *Advice* kann dabei wie eine Methode behandelt werden, da auch ein benamter *Advice* einen Namen, einen Ergebnistypen und eine Argumentenliste besitzt. Um einen solchen *Advice* verfeinern zu können, wird die *Advice*-Methode überschrieben, indem eine Methode mit demselben Namen und derselben Signatur wie der benamte *Advice* innerhalb der Verfeinerung des Aspektes erstellt wird. Auch der Aufruf des urspünglichen *Advicecodes* ist mit der Hilfe von Befehlen wie zum Beispiel *super* oder *original* möglich.

5.4 Zusammenfassung

In der Ergebnisdiskussion wurde festgestellt, dass das Merkmalsdiagramm für eine Konfigurationsunterstützung hilfreich ist und eine umfangreiche Anzahl an Merkmalen liefert. So bietet sich einem Nutzer, welcher eine modulare Transaktionsunterstützung nutzen möchte, eine umfassende Auswahl an Alternativen zur individuellen Anpassung der Merkmale an seine Bedürfnisse. Insgesamt wurde unter Beachtung von korrekten Merkmalskombinationen eine modulare Transaktionsunterstützung entworfen, die einmal umgesetzt, einer breiten Masse an Nutzern zur Verfügung gestellt werden kann und trotzdem die Möglichkeit bietet, auf die Nutzeranforderungen individuell zugeschnitten zu werden. Beim Entwurf des korrespondierenden Kollaborationsdiagramms

traten Überlegungen zu möglichen Umsetzungsproblemen auf, die durch bestehende Hilfskonstruktionen und -mechanismen jedoch bereits gelöst werden können. Zu diesen Problemen gehörten zum Beispiel das Erweitern von optionalen Konstrukten und das Verfeinern von Aspekten.

Des Weiteren lässt sich feststellen, dass im Falle der Nutzung der Transaktionsunterstützung immer noch die Möglichkeit für den Entwickler gegeben ist, individuell zu entscheiden, ob er einen Bereich im Quellcode in die Transaktionsunterstützung miteinbeziehen möchte. Durch die Einführung der Funktionen *beginTransaktion* und *commitTransaktion* beziehungsweise *abortTransaktion*, zum Starten und Beenden von Transaktionen können Anweisungsblöcke, die unterstützt werden sollen, eingekapselt werden, so dass beim Aufruf der vordefinierten Funktionen *put*, *get* und *delete* zum Zugriff auf *Amazon S3* die Transaktionsunterstützung wirksam wird. Gleichzeitig kann jedoch weiterhin auf *Amazon S3* zugegriffen werden, indem an den entsprechenden Stellen auf die Nutzung der vordefinierten Funktionen aus der *CustomAPI* verzichtet wird und die Zugriffe somit nicht in die Synchronisation mit eingebunden werden.

Kapitel 6

6 Ausblick und Zusammenfassung

Im Folgenden wird ein Ausblick auf mögliche Erweiterungen und eine mögliche Verwendung für andere Systeme als *Amazon S3* gegeben und das Ergebnis der Arbeit kurz zusammengefasst.

6.1 Ausblick

Aufbauend auf dem Kollaborationsdiagramm besteht nun die Möglichkeit, die Transaktionsunterstützung umzusetzen, sofern eine Programmiersprache verwendet wird, die *Aspectual Feature Modules* unterstützt und die gleichzeitig den Zugriff auf *Amazon S3* ermöglicht. Angefangen bei den Funktionen zum Zugriff auf die *Amazon S3*-Daten würden zunächst die Kernmerkmale erstellt und implementiert und nach und nach um die variablen Merkmale erweitert werden. Dabei ist von Anfang an zu bedenken, welche Merkmale miteinander interagieren müssen, so dass gegebenenfalls *Hook*-Methoden eingesetzt werden können, um auch innerhalb eines Algorithmus beziehungsweise einer Anweisungsfolge Funktionalität integrieren zu können. Zusätzlich wäre eine Erweiterung der Funktionen der *CustomAPI* denkbar, so dass zum Beispiel auch Funktionen mit Zugriffsbeschränkungen und Sicherheitsmechanismen angeboten werden könnten. Sobald alle Merkmale umgesetzt sind, müsste ein Testsystem erstellt und evaluiert werden. Dazu wäre es notwendig möglichst viele Konfigurationskombinationen auf ihre korrekte Umsetzung zu überprüfen und Fehler im Quellcode gegebenenfalls zu korrigieren. Im Grunde müssten alle möglichen Kombinationen getestet werden, was jedoch zu einem erheblichen Zeitaufwand führt und die Entwicklung von alternativen Analysetechniken notwendig macht [Thüm et al., 2012]. Umgesetzt wäre die modulare Transaktionsunterstützung, sofern sie effizient arbeitet, für den Einsatz in realen Entwicklungsumgebungen verwendbar. Alle Entwickler, die ihre Anwendungen aufbauend auf *Amazon S3* als Datenspeicher entwickeln und Mehrbenutzerzugriffe koordinieren müssen, könnten die Transaktionsunterstützung aktivieren, an ihre Bedürfnisse anpassen und an gewünschten Stellen im Quellcode integrieren.

Mögliche Erweiterungen

Das Merkmalsdiagramm zur Konfigurationsunterstützung und das Kollaborationdiagramm als Orientierungshilfe für die eigentliche Implementierung können nicht nur für den Zugriff auf *Amazon S3*-Objekte verwendet werden. Denkbar ist eine Anpassung an weitere *Cloud*-Dienste, mit deren Hilfe unterschiedliche Nutzer auf einen geteilten

Speicher zugreifen. Die Merkmale bleiben dabei identisch. Nur die Zugriffsmethoden müssten an das jeweilige System angepasst werden. Dabei könnte die Adaption auf andere *NoSQL*-Systeme leichter umgesetzt werden, sofern sie zu dem System von *Amazon S3* ähnlich sind. Aber auch eine Erweiterung auf Systeme mit höheren Zugriffssprachen beziehungsweise -befehlen wäre möglich. Dazu benötigt die jeweilige Zugriffsmethode eventuell ein Modul, welches den komplexeren Befehl in die Grundbefehle aufschlüsselt, so dass diese wieder mit Hilfe der Transaktionsunterstützung verwaltet werden können.

Eine weitere Ergänzung könnte durch die Anwendung der Transaktionsunterstützung auf Dateiinhalte erfolgen. Statt nur den Zugriff auf ganze *Amazon S3*-Objekte zu koordinieren, wäre es möglich, Funktionen zur Verfügung zu stellen, die für bestimmte Dateitypen Zugriffe auf deren Inhalt synchronisieren. Dazu müssten jedoch für jeden unterstützten Dateityp separate Zugriffsmethoden erstellt werden. Als Beispiel könnten dafür XML-Dateien (engl. *Extensible Markup Language* (XML); dt. erweiterbare Auszeichnungssprache) angeführt werden, bei denen die Synchronisation die jeweiligen XML-Elemente betrifft, so dass mehrere Nutzer gleichzeitig an einem XML-Dokument arbeiten könnten.

6.2 Zusammenfassung

Im Rahmen dieser Arbeit wurden zum Zwecke der Entwicklung einer modularen und konfigurierbaren Transaktionsunterstützung sowohl ein Merkmalsdiagramm zur Unterstützung der Konfigurierbarkeit als auch ein darauf aufbauendes Kollaborationsdiagramm zur Visualisierung der benötigten Klassen und Erweiterungen erstellt. Zunächst wurden dazu alle benötigten Grundlagen bezüglich der *Cloud* als gesetztes Zielsystem, der Transaktionsverwaltung und der Wiederherstellungskomponenten als zu integrierende Merkmale sowie der *Aspectual Feature Modules* als verwendete Umsetzungstechnik aufgezeigt. Anschließend wurde zunächst die Zieldomäne analysiert und darauf aufbauend das Merkmalsdiagramm entworfen. Dabei stand im Vordergrund, eine umfangreiche Funktionsvielfalt zu ermitteln und entsprechend zu modellieren. Prinzipiell ergaben sich dadurch zwei, die Hauptpfeiler einer Transaktionsunterstützung betreffenden Teilbäume - zum einen den Transaktionsmanager für die Synchronisation und Steuerung der Zugriffsbefehle und zum anderen den Speichermanager zur Verwaltung des Puffers und der Wiederherstellungskomponenten. Durch das Merkmalsdiagramm ergab sich eine definierte Menge an Kern- und variablen Merkmalen, so dass durch die Kernmerkmale eine grundlegende Transaktionsunterstützung gewährleistet und mit Hilfe der variablen Merkmale für verschiedene Endanwender eine individuelle Transaktionsunterstützung zur Verfügung gestellt werden kann. Das Merkmalsdiagramm besteht sowohl aus abstrakten als auch konkreten Merkmalen, wobei die konkreten Merkmale für eine mögliche praktische Umsetzung relevant sind. Ein erster Schritt in Richtung Implementierung bildet dabei das anschließend entworfene Kollaborationsdiagramm, welches spezifiziert, inwiefern Klassen eingeführt, verfeinert und durch Aspekte ergänzt werden könnten und wie der Ablauf einer Transaktionsunterstützung aussehen würde. Ausgegangen wurde dabei davon, dass die Transaktionsunterstützung nur dann angewendet wird, wenn dies vom Nutzer ausdrücklich gewünscht wird. Dazu wurde eine Möglichkeit entwickelt, durch die der Entwickler angeben kann, ab wann er Zugriffe synchronisieren möchte. Zugriffe auf *Amazon S3* werden nur dann in die Transaktionsunterstützung mit aufgenommen, wenn die vordefinierte *API* verwendet wird, so dass dem Nutzer frei steht, Zugriffe auch ohne eine Unterstützung auszuführen.

Im Rahmen der Entwicklung der Modelle ergaben sich letztendlich mehrere Erkenntnisse. So besteht das Ergebnis aus einer Vielzahl an Merkmalen, die umgesetzt werden müssten. Dabei muss beachtet werden, dass mehrere Merkmale ein und dieselbe Klasse und teilweise auch Funktionen beziehungsweise Eigenschaften modifizieren, was gegebenenfalls zu dem Vermischen von Quellcodebestandteilen verschiedener Merkmale führt. Daraus folgt, dass während der Umsetzung jede mögliche Kombination von Merkmalen bedacht werden muss, um den Aufwand zu minimieren, im Nachhinein wieder etwas ändern zu müssen. Das bedeutet auch, dass *Hook*-Methoden rechtzeitig integriert werden, so dass die Integration von weiteren Merkmalen möglich wird, wenn sie Änderungen in Anweisungsblöcken vornehmen müssen. Weitere aufgetretene Erkenntnisse bestehen darin, dass durch das Hinzufügen des manuellen Abbruchs von Transaktionen durch den Nutzer die optionale Funktion *abortTransaktion* entstanden ist, welche wiederum nicht durch Verfeinerungen erweiterbar ist. Dies führte dazu, dass das optionale Weben eingesetzt wird und Aspekte verwendet werden mussten. Zusätzlich dazu ergab sich dann, dass auch *Aspect Refinement* notwendig wird, da ein Aspekt zum Beispiel durch die erweiterten Transaktionsmodelle weiter verfeinert wird.

Im Endeffekt wurde durch die Entwicklung der beiden Modelle eine mögliche Grundlage geschaffen, um eine modulare Integration von konfigurierbaren Transaktionsmechanismen in *Cloud*-basierte Datenspeichersysteme möglich zu machen, wobei in dieser Arbeit *Amazon S3* als Beispielsystem verwendet wurde. Eine Adaption für andere Systeme schließt dies jedoch nicht aus, da nur die Zugriffsfunktionen und wenige Merkmale an das jeweilige Zielsystem angepasst werden müssen.

Literaturverzeichnis

[Abadi 2009] Abadi, Daniel J.: Data Management in the Cloud: Limitations and Opportunities. In: *IEEE Data Engineering Bulletin* 32 (2009), Nr. 1, S. 3–12

[Abbott und Garcia-Molina 1992] Abbott, Robert K. ; Garcia-Molina, Hector: Scheduling real-time Transactions: A Performance Evaluation. In: *ACM Transactions on Database Systems* 17 (1992), Nr. 3, S. 513–560

[Abts 2010] Abts, Dietmar: *Grundkurs Java: Von den Grundlagen bis zu Datenbank- und Netzanwendungen.* Vieweg+Teubner Verlag, 2010

[Adelsberger und Drechsler 2010] Adelsberger, Heimo ; Drechsler, Andreas: Ausgewählte Aspekte des Cloud-Computing aus einer IT-Management-Perspektive: Cloud Governance, Cloud Security und Einsatz von Cloud Computing in jungen Unternehmen / Institut für Informatik und Wirtschaftsinformatik (ICB), Universität Duisburg-Essen. 2010 (41). – Technischer Bericht

[Adler 2010] Adler, Constanze: *Optional Composition - A Solution to the Optional Feature Problem?*, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Diplomarbeit, 2010

[Afek et al. 2010] Afek, Yehuda ; Korland, Guy ; Yanovsky, Eitan: Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. In: *Proceedings of the 14th International Conference on Principles of Distributed systems*, Springer-Verlag, 2010 (OPODIS'10), S. 395–410

[Agrawal et al. 2006] Agrawal, Kunal ; Leiserson, Charles E. ; Sukha, Jim: Memory Models for Open-Nested Transactions. In: *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, 2006 (MSPC '06), S. 70–81

[Alwis et al. 2000] Alwis, Brian D. ; Gudmundson, Stephan ; Smolyn, Greg ; Kiczales, Gregor: Coding Issues in AspectJ. In: *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000

[Amazon.com 2011a] Amazon.com, Inc. oder T.: *Amazon Simple Storage Service (Amazon S3).* `http://aws.amazon.com/de/s3/`. Version: 2011. – Stand: 12. September 2011

[Amazon.com 2011b] Amazon.com, Inc. oder T.: *Produkte & Dienstleistungen.* `http://aws.amazon.com/de/products/`. Version: 2011. – Stand: 12. September 2011

[Annapureddy 2010] Annapureddy, Koushik: Security Challenges in Hybrid Cloud Infrastructures / Aalto University - School of Science and Technology. 2010. – Technischer Bericht

[Apel et al. 2006a] Apel, Sven ; Batory, Don ; Rosenmüller, Marko: On the Structure of Crosscutting Concerns: Using Aspects or Collaborations. In: *Proceedings of GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2006

[Apel et al. 2007a] Apel, Sven ; Kästner, Christian ; Kuhlemann, Martin ; Leich, Thomas: Pointcuts, Advice, Refinements, and Collaborations: Similarities, Differences, and Synergies. In: *Innovations in Systems and Software Engineering (ISSE)* 3 (2007), Nr. 4, S. 281–289

[Apel et al. 2007b] Apel, Sven ; Kästner, Christian ; Leich, Thomas ; Saake, Gunter: Aspect Refinement - Unifying AOP and Stepwise Refinement. In: *Journal of Object Technology* 6 (2007), Nr. 9, S. 13–33

[Apel et al. 2005a] Apel, Sven ; Leich, Thomas ; Rosenmüller, Marko ; Saake, Gunter: Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution. In: *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2005 (RAM-SE'05), S. 3–16

[Apel et al. 2005b] Apel, Sven ; Leich, Thomas ; Saake, Gunter: Aspectual Mixin Layers / Department of Computer Science, Otto-von-Guericke University. 2005. – Technischer Bericht

[Apel et al. 2006b] Apel, Sven ; Leich, Thomas ; Saake, Gunter: Aspectual Mixin Layers: Aspects and Features in Concert. In: *Proceedings of the 28th International Conference on Software Engineering*, 2006 (ICSE '06), S. 122–131

[Apel et al. 2008] Apel, Sven ; Leich, Thomas ; Saake, Gunter: Aspectual Feature Modules. In: *IEEE Trans. Softw. Eng.* 34 (2008), Nr. 2, S. 162–180

[Awessou 2009] Awessou, H. Romuald P.: *Eine Event-Description- und Subscription-Sprache für Webservices*, Universität Stuttgart - Fakultät für Informatik, Elektrotechnik und Informationstechnik, Diplomarbeit, 2009

[Aymerich et al. 2008] Aymerich, Francesco M. ; Fenu, Gianni ; Surcis, Simone: An approach to a Cloud Computing network. In: *First International Conference on the Applications of Digital Information and Web Technologies (ICADIWT)*, 2008, S. 113–118

[Back 2001] Back, Andrea: *Lexikon der Wirtschaftsinformatik*. Springer-Verlag, 2001

[Balling 2004] Balling, Derek J.: *High Performance MySQL*. O'Reilly Germany, 2004

[Bass et al. 2000] Bass, Len ; Clements, Paul ; Donohoe, Patrick ; McGregor, John ; Northrop, Linda: Fourth Product Line Practice Workshop Report / Software Engineering Institute, Carnegie Mellon University. 2000 (CMU/SEI-2000-TR-002). – Technischer Bericht

[Batory et al. 2003] Batory, Don ; Sarvela, Jacob N. ; Rauschmayer, Axel: Scaling Step-wise Refinement. In: *Proceedings of the 25th International Conference on Software Engineering*, 2003 (ICSE '03), S. 187–197

[Batory 2005] Batory, Don S.: Feature Models, Grammars, and Propositional Formulas. In: *Proceedings of the 9th International Software Product Lines Conference*, 2005 (SPLC '05), S. 7–20

[Baun et al. 2009] Baun, Christian ; Kunze, Marcel ; Nimis, Jens ; Tai, Stefan: *Cloud Computing: Web-basierte dynamische IT-Services.* Springer-Verlag, 2009 (Informatik im Fokus)

[Bernstein und Goodman 1981] Bernstein, Philip A. ; Goodman, Nathan: Concurrency Control in Distributed Database Systems. In: *ACM Computing Surveys* 13 (1981), Nr. 2, S. 185–221

[Bernstein und Goodman 1983] Bernstein, Philip A. ; Goodman, Nathan: Multiversion Concurrency Control - Theory and Algorithms. In: *ACM Transactions on Database Systems* 8 (1983), Nr. 4, S. 465–483

[Bernstein et al. 1987] Bernstein, Philip A. ; Hadzilacos, Vassos ; Goodman, Nathan: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987

[Beyerle et al. 2005] Beyerle, Marc ; Franz, Joachim ; Spruth, Wilhelm G.: Persistent Reusable Java Virtual Machine unter z/OS und Linux. In: *Informatik - Forschung und Entwicklung* 20 (2005), Nr. 1, S. 102–111

[Bhardwaj et al. 2010] Bhardwaj, Sushil ; Jain, Leena ; Jain, Sandeep: Cloud Computing: A Study of Infrastructure as a Service (IaaS). In: *International Journal of Engineering and Information Technology* 2 (2010), Nr. 1, S. 60–63

[Binder et al. 2007] Binder, Walter ; Spycher, Samuel ; Constantinescu, Ion ; Faltings, Boi: Multiversion Concurrency Control for Multidimensional Index Structures. In: *Database and Expert Systems Applications* Bd. 4653. Springer-Verlag, 2007, S. 172–181

[Birk und Wegener 2010] Birk, Dominik ; Wegener, Christoph: Über den Wolken: Cloud Computing im Überblick. In: *Datenschutz und Datensicherheit - DuD* 34 (2010), Nr. 9, S. 641–645

[Born 2010] Born, Günter: *Office 2010: Leicht, klar, sofort.* Pearson Deutschland GmbH, 2010

[Boswell 2003] Boswell, William: *Windows Server 2003 für Insider.: Umfassende Referenz für Netzwerkspezialisten.* Pearson Deutschland GmbH, 2003

[Brewer 2000] Brewer, Eric A.: Towards Robust Distributed Systems. In: *Symposium on Principles of Distributed Computing (PODC)*, 2000, S. 7

[Bruntink et al. 2004] Bruntink, Magiel ; Deursen, Arie van ; Engelen, Remco van ; Tourwe, Tom: An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In: *In Proc. of the International Conference on Software Maintenance (ICSM)*, 2004, S. 200–209

[Bun 2005] Bundesamt für Sicherheit in der Informationstechnik (Hrsg.): *Konzeption von Sicherheitsgateways. Der richtige Aufbau und die passenden Module für ein sicheres Netz.* Bundesanzeiger. Bundesamt für Sicherheit in der Informationstechnik, 2005

[Buxmann et al. 2011] Buxmann, Peter ; Lehmann, Sonja ; Draisbach, Tobias ; Koll, Corina ; Diefenbach, Heiner ; Ackermann, Tobias: Cloud Computing und Software as a Service: Konzeption und Preisgestaltung. In: *Online-Recht 2.0: Alte Fragen - neue Antworten?* Boorberg Verlag, 2011, S. 21–34

[Buyya et al. 2009] Buyya, Rajkumar ; Pandey, Suraj ; Vecchiola, Christian: Cloudbus Toolkit for Market-Oriented Cloud Computing. In: *Proceedings of the 1st International Conference on Cloud Computing*, 2009 (CloudCom '09), S. 24–44

[Camp et al. 1995] Camp, L. J. ; Sirbu, Marvin ; Tygar, J. D.: Token and Notational Money in Electronic Commerce. In: *Proceedings of the 1st Conference on USENIX Workshop on Electronic Commerce*, 1995 (WOEC '95), S. 1

[Cavilla et al. 2009] Cavilla, Horacio Andrés L. ; Whitney, Joseph A. ; Scannell, Adin M. ; Patchin, Philip ; Rumble, Stephen M. ; Lara, Eyal de ; Brudno, Michael ; Satyanarayanan, Mahadev: SnowFlock: Rapid virtual Machine Cloning for Cloud Computing. In: *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009 (EuroSys '09), S. 1–12

[Chen et al. 2010] Chen, Yanpei ; Paxson, Vern ; Katz, Randy H.: What's new about Cloud Computing Security? / EECS Department, University of California. 2010 (UCB/EECS-2010-5). – Technischer Bericht

[Clements und Northrop 2001] Clements, Paul ; Northrop, Linda: *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2001

[Colyer et al. 2004] Colyer, Adrian ; Rashid, Awais ; Blair, Gordon: On the Separation of Concerns in Program Families / Computing Department, Lancaster University. 2004. – Technischer Bericht

[Cormen et al. 2001] Cormen, Thomas H. ; Stein, Clifford ; Rivest, Ronald L. ; Leiserson, Charles E.: *Introduction to Algorithms.* McGraw-Hill Higher Education, 2001

[Cosoi und Vossen 2010] Cosoi, Catalin ; Vossen, Gottfried: On Scanning the Internet or the Curse of In-The-Cloud URL Scanning. In: *20th Virus Bulletin International Conference* (2010)

[Coulouris et al. 2005] Coulouris, George F. ; Dollimore, Jean ; Kindberg, Tim: *Distributed Systems: Concepts and Design.* Pearson Education, 2005

[Czarnecki 1998] Czarnecki, Krzysztof: *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, Technical University of Ilmenau, Diss., 1998

[Czarnecki und Eisenecker 1999] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: Components and Generative Programming. In: *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 1999 (ESEC/FSE-7), S. 2–19

[Czarnecki und Eisenecker 2000] Czarnecki, Krzysztof ; Eisenecker, Ulrich W.: *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley Publishing Co., 2000

[Dadam 1996] Dadam, Peter: *Verteilte Datenbanken und Client/Server-Systeme.* Springer-Verlag, 1996

[Dayal et al. 1991] Dayal, Umeshwar ; Hsu, Meichun ; Ladin, Rivka: A Transactional Model for Long-Running Activities. In: *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91)*, 1991, S. 113–122

[Deussen et al. 2010] Deussen, Peter H. ; Strick, Linda ; Peters, Johannes: *Cloud-Computing für die öffentliche Verwaltung: ISPRAT-Studie.* Fraunhofer-Insitut für Offene Kommunikationssysteme FOKUS, 2010

[Eaddy und Aho 2006] Eaddy, Marc ; Aho, Alfred: Statement Annotations for Fine-Grained Advising. In: *In ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, 2006, S. 89–99

[Edlich et al. 2011] Edlich, Stefan ; Friedland, Achim ; Hampe, Jens ; Brauer, Benjamin: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken.* Carl Hanser Fachbuchverlag, 2011

[Effelsberg und Härder 1984] Effelsberg, Wolfgang ; Härder, Theo: Principles of Database Buffer Management. In: *ACM Transactions on Database Systems* 9 (1984), Nr. 4, S. 560–595

[Eichberg et al. 2004] Eichberg, Michael ; Mezini, Mira ; Ostermann, Klaus: Pointcuts as Functional Queries. In: *Asian Symposium on Programming Languages and Systems*, 2004 (APLAS '04), S. 366–381

[Eisentraut und Helmle 2010] Eisentraut, Peter ; Helmle, Bernd: *PostgreSQL-Administration.* O'Reilly Germany, 2010

[Elmagarmid 1992] Elmagarmid, Ahmed K. (Hrsg.): *Database Transaction Models for Advanced Applications.* Morgan Kaufmann Publishers Inc., 1992

[Elmagarmid et al. 1990] Elmagarmid, Ahmed K. ; Leu, Yungho ; Litwin, Witold ; Rusinkiewicz, Marek: A Multidatabase Transaction Model for InterBase. In: *Proceedings of the 16th International Conference on Very Large Data Bases*, 1990 (VLDB '90), S. 507–518

[Elrad et al. 2001] Elrad, Tzilla ; Filman, Robert E. ; Bader, Atef: Aspect-oriented programming: Introduction. In: *Communincations of the ACM* 44 (2001), Nr. 10, S. 29–32

[Foster et al. 2008] Foster, Ian T. ; Zhao, Yong ; Raicu, Ioan ; Lu, Shiyong: Cloud Computing and Grid Computing 360-Degree compared. In: *Grid Computing Environments Workshop*, 2008 (GCE '08), S. 1–10

[Gal et al. 2002] Gal, Andreas ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: On Aspect-Orientation in Distributed Real-time Dependable Systems. In: *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002 (WORDS '02), S. 261–270

[Garcia-Molina und Salem 1987] Garcia-Molina, Hector ; Salem, Kenneth: Sagas. In: *SIGMOD Rec.* 16 (1987), Nr. 3, S. 249–259

[Gonzalez et al. 2009] Gonzalez, Luis Miguel V. ; Rodero-Merino, Luis ; Caceres, Juan ; Lindner, Maik A.: A Break in the Clouds: Towards a Cloud Definition. In: *Computer Communication Review* 39 (2009), Nr. 1, S. 50–55

[Gray 1978] Gray, Jim: Notes on data base operating systems. In: *Operating Systems* Bd. 60. Springer-Verlag, 1978, S. 393–481

[Gray et al. 1981a] Gray, Jim ; Homan, Pete ; Korth, Henry F. ; Obermarck, Ron: A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System. In: *Berkeley Workshop on Distributed Data Management and Computer Networks*, 1981, S. 125

[Gray et al. 1981b] Gray, Jim ; McJones, Paul ; Blasgen, Mike ; Lindsay, Bruce ; Lorie, Raymond ; Price, Tom ; Putzolu, Franco ; Traiger, Irving: The Recovery Manager of the System R Database Manager. In: *ACM Computing Surveys* 13 (1981), Nr. 2, S. 223–242

[Gray und Reuter 1992] Gray, Jim ; Reuter, Andreas: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992

[Großmann und Koschek 2005] Grossmann, Martina ; Koschek, Holger: Referenzarchitektur. In: *Unternehmensportale*. Springer-Verlag, 2005, S. 155–181

[Gropengießer und Sattler 2011] Gropengiesser, Francis ; Sattler, Kai-Uwe: Transactions a la Carte - Implementation and Performance Evaluation of Transactional Support on Top of Amazon S3. In: *IPDPS Workshops*, 2011, S. 1082–1091

[Group 2007] Group, ISRD: *Relational Database Management Systems*. Tata McGraw-Hill Education, 2007 (MSBTE)

[Guruzu und Mak 2010] Guruzu, Srinivas ; Mak, Gary: Transactions and Concurrency. In: *Hibernate Recipes: A Problem-Solution Approach*. Apress, 2010, S. 219–236

[Haselmann und Vossen 2010a] Haselmann, Till ; Vossen, Gottfried: Abgrenzung des Database-as-a-Service-Begriffs aus Benutzerperspektive / European Research Center for Information Systems (ERCIS). 2010. – Technischer Bericht

[Haselmann und Vossen 2010b] Haselmann, Till ; Vossen, Gottfried: Database-as-a-Service für kleine und mittlere Unternehmen / Westfälische Wilhelms-Universität. 2010. – Technischer Bericht

[Hering 2005] Hering, Ekbert: *Taschenbuch der Mechatronik*. Hanser Verlag, 2005

[Herlihy 1990] Herlihy, Maurice: Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. In: *ACM Transactions on Database Systems* 15 (1990), Nr. 1, S. 96–124

[Heuer et al. 2008] Heuer, Andreas ; Saake, Gunter ; Sattler, Kai-Uwe: *Datenbanken: Konzepte und Sprachen*. Hüthig Jehle Rehm, 2008

[Hewitt 2011] Hewitt, Eben: *Cassandra - The Definitive Guide: Distributed Data at Web Scale*. Springer-Verlag, 2011

[Härder und Rahm 2001] Härder, Theo ; Rahm, Erhard: *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer-Verlag, 2001

[Härder und Reuter 1983] Härder, Theo ; Reuter, Andreas: Principles of Transaction-Oriented Database Recovery. In: *ACM Computing Surveys* 15 (1983), Nr. 4, S. 287–317

[Hürsch und Lopes 1995] Hürsch, Walter ; Lopes, Cristina V.: Separation of concerns / College of Computer Science. 1995 (NU-CCS-95-03). – Technischer Bericht

[Jensen et al. 2009] Jensen, Meiko ; Schwenk, Jörg ; Gruschka, Nils ; Iacono, Luigi L.: On Technical Security Issues in Cloud Computing. In: *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, 2009 (CLOUD '09), S. 109–116

[Jimenez 2010] Jimenez, Jaime: Security Concerns in commercial Cloud Computing / Aalto University - School of Science and Technology. 2010. – Technischer Bericht

[Kamara und Lauter 2010] Kamara, Seny ; Lauter, Kristin: Cryptographic Cloud Storage. In: *Proceedings of the 14th international Conference on Financial Cryptograpy and Data Security*, 2010 (FC'10), S. 136–149

[Kang et al. 1990] Kang, Kyo C. ; Cohen, Sholom G. ; Hess, James A. ; Nowak, William E. ; Peterson, Spencer: Feature-Oriented Domain Analysis (FODA) Feasibility Study / Carnegie-Mellon University Software Engineering Institute. 1990. – Technischer Bericht

[Kedar 2009a] Kedar, Seema: *Database Management Systems.* Technical Publications, 2009

[Kedar 2009b] Kedar, Seema: *GUI and Database Management.* Technical Publications, 2009

[Kemper und Eickler 2006] Kemper, Alfons ; Eickler, André: *Datenbanksysteme.* Oldenbourg Wissenschaftsverlag, 2006

[Khoshafian und Baker 1996] Khoshafian, Setrag ; Baker, A. B.: *Multimedia and imaging databases.* Morgan Kaufmann Publishers Inc., 1996

[Kiczales et al. 1997] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina V. ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997 (Lecture Notes in Computer Science), S. 220–242

[Kienzle 2003] Kienzle, Jörg: *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming.* Springer-Verlag, 2003

[Kim 2009] Kim, Won: Cloud Computing: Today and Tomorrow. In: *Journal of Object Technology* 8 (2009), Nr. 1, S. 65–72

[König 2003] König, Wolfgang: *Taschenbuch der Wirtschaftsinformatik und Wirtschaftsmathematik.* Harri Deutsch Verlag, 2003

[Kossmann et al. 2010] Kossmann, Donald ; Kraska, Tim ; Loesing, Simon: An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In: *Proceedings of the 2010 International Conference on Management of Data*, 2010 (SIGMOD '10), S. 579–590

[Kraska et al. 2009] Kraska, Tim ; Hentschel, Martin ; Alonso, Gustavo ; Kossmann, Donald: Consistency Rationing in the Cloud: Pay Only When It Matters. In: *Proc. VLDB Endow.* 2 (2009), Nr. 1, S. 253–264

[Kästner 2007] Kästner, Christian: *Aspect-Oriented Refactoring of Berkeley DB*, University of Magdeburg, Diplomarbeit, 2007

[Kästner et al. 2008] Kästner, Christian ; Apel, Sven ; Kuhlemann, Martin: Granularity in software product lines. In: *Proceedings of the 30th international conference on Software engineering*, 2008 (ICSE '08), S. 311–320

[Kudraß 2007] Kudrass, Thomas: *Taschenbuch Datenbanken.* Hanser Verlag, 2007

[Kundra 2010] Kundra, Vivek: State of Public Sector Cloud Computing / Federal Government Chief Information Office. 2010. – Technischer Bericht

[Lahres und Rayman 2009] Lahres, Bernhard ; Rayman, Gregor: *Objektorientierte Programmierung - Das umfassende Handbuch.* Galileo Computing, 2009

[Leavitt 2009] Leavitt, Neal: Is Cloud Computing really ready for Prime Time? In: *Computer* 42 (2009), Nr. 1, S. 15–20

[Lee 2010] Lee, Craig A.: A Perspective on scientific Cloud Computing. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010 (HPDC '10), S. 451–459

[Lee und Hughes 2010] Lee, Kevin ; Hughes, Danny: System Architecture Directions for Tangible Cloud Computing. In: *First ACIS International Symposium on Cryptography, and Network Security, Data Mining and Knowledge Discovery, E-Commerce and Its Applications, and Embedded Systems (CDEE)* (2010), S. 258–262

[Lee und Liou 1996] Lee, Suh-Yin ; Liou, Ruey-Long: A Multi-Granularity Locking Model for Concurrency Control in Object-Oriented Database Systems. In: *IEEE Transaction on Knowledge and Data Engineering* 8 (1996), Nr. 1, S. 144–156

[Lee und Lam 1999] Lee, Victor C. S. ; Lam, Kwok-Wa: Optimistic Concurrency Control in Broadcast Environments: Looking Forward at the Server and Backward at the Clients. In: *Proceedings of the First International Conference on Mobile Data Access*, Springer-Verlag, 1999 (MDA '99), S. 97–106

[Leich et al. 2005a] Leich, Thomas ; Apel, Sven ; Rosenmüller, Marko ; Saake, Gunter: Handling Optional Features in Software Product Lines. In: *Proceedings of OOPSLA Workshop on Managing Variabilities consistently in Design and Code*, 2005

[Leich et al. 2005b] Leich, Thomas ; Apel, Sven ; Saake, Gunter: Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In: *Proceedings of the 9th East European conference on Advances in Databases and Information Systems*, 2005 (ADBIS'05), S. 324–337

[Levene und Loizou 1999] Levene, Mark ; Loizou, George: *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999

[Lieberherr et al. 2003] Lieberherr, Karl ; Lorenz, David H. ; Ovlinger, Johan: Aspectual Collaborations: Combining Modules and Aspects. In: *Computer Journal* 46 (2003), Nr. 5, S. 542–565

[Liu et al. 2005] Liu, Jia ; Batory, Don S. ; Nedunuri, Srinivas: Modeling Interactions in Feature Oriented Software Designs. In: *Feature Interactions in Telecommunications and Software Systems VIII*, 2005, S. 178–197

[Lorie 1977] Lorie, Raymond A.: Physical Integrity in a Large Segmented Database. In: *ACM Transactions on Database Systems* 2 (1977), S. 91–104

[Maggiani 2009] Maggiani, Rich: Cloud computing is changing how we communicate. In: *IEEE International Professional Communication Conference (IPCC)* (2009), S. 1–4

[Mahrenholz et al. 2002a] Mahrenholz, Daniel ; Spinczyk, Olaf ; Gal, Andreas ; Schröder-Preikschat, Wolfgang: An Aspect-Oriented Implementation of Interrupt Synchronization in the PURE Operating System Family. In: *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems*, 2002 (ECOOP-OOOSWS '02), S. 49–54

[Mahrenholz et al. 2002b] Mahrenholz, Daniel ; Spinczyk, Olaf ; Schröder-Preikschat, Wolfgang: Program Instrumentation for Debugging and Monitoring with AspectC++. In: *Symposium on Object-Oriented Real-Time Distributed Computing'02*, 2002, S. 249–256

[Malita und Boffo 2010] Malita, Laura ; Boffo, Vanna: *Digital Storytelling for Employability*. Firenze University Press, 2010

[Mandl 2009] Mandl, Peter: Konzepte, Modelle und Standards verteilter Transaktionsverarbeitung. In: *Masterkurs Verteilte betriebliche Informationssysteme*. Vieweg+Teubner, 2009, S. 225–314

[Masak 2006] Masak, Dieter: Architekturen und Sprachen. In: *Legacysoftware*. Springer-Verlag, 2006, S. 299–372

[Masuhara und Kiczales 2003] Masuhara, Hidehiko ; Kiczales, Gregor: Modeling Crosscutting in Aspect-Oriented Mechanisms. In: Cardelli, Luca (Hrsg.): *ECOOP 2003 - Object-Oriented Programming* Bd. 2743. Springer-Verlag, 2003, S. 219–233

[Masuhara et al. 2003] Masuhara, Hidehiko ; Kiczales, Gregor ; Dutchyn, Chris: A Compilation and Optimization Model for Aspect-Oriented Programs. In: *Proceedings of the 12th international conference on Compiler construction*, 2003 (CC '03), S. 46–60

[Meier 2004] Meier, Andreas: *Relationale Datenbanken: Leitfaden für die Praxis*. Springer-Verlag, 2004

[Meier 2010] Meier, Andreas: Elemente der Systemarchitektur. In: *Relationale und postrelationale Datenbanken*. Springer-Verlag, 2010, S. 93–129

[Mell und Grance 2009] Mell, Peter ; Grance, Tim: The NIST Definition of Cloud Computing / National Institute of Standards and Technology. 2009 (15). – Technischer Bericht

[Melzer 2010] Melzer, Ingo: Web Services und Transaktionen. In: *Service-orientierte Architekturen mit Web Services.* Spektrum Akademischer Verlag, 2010, S. 275–296

[Merkt et al. 2007] Merkt, Marianne ; Mayrberger, Kerstin ; Schulmeister, Rolf ; Berk, Ivo van d.: *Studieren neu erfinden - Hochschule neu denken.* Waxmann Verlag, 2007

[Mezini und Ostermann 2004] Mezini, Mira ; Ostermann, Klaus: Variability Management With Feature-Oriented Programming and Aspects. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004, S. 127–136

[Michelmann 2010] Michelmann, Marcel: *Anforderungsanalyse und Bewertung von Cloud Computing-Lösungen*, Ludwig-Maximilians-Universität München, Diplomarbeit, 2010

[Microsoft 2011] Microsoft: *So installieren Sie das Microsoft Office Compatibility Pack, mit dessen Hilfe Sie Dateien aus Office 2007-Programmen mit früheren Versionen von Excel, PowerPoint und Word öffnen und speichern können.* `http://support.microsoft.com/kb/923505/de`. Version: 16.Mai 2011. – Stand: 13. September 2011

[Miller 2008] Miller, Michael: *Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online.* Que Publishing, 2008

[Mirzaei 2009] Mirzaei, Nariman: Cloud Computing / Indiana University. 2009. – Technischer Bericht

[Mohan et al. 1992] Mohan, C. ; Haderle, Don ; Lindsay, Bruce ; Pirahesh, Hamid ; Schwarz, Peter: Aries: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In: *ACM Transactions on Database Systems* 17 (1992), S. 94–162

[Moss 1981] Moss, J. Eliot B.: *Nested transactions: An Approach to Reliable Distributed Computing.* 1981

[Mowbray 2009] Mowbray, Miranda: The Fog over the Grimpen Mire: Cloud Computing and the Law. In: *Scripted Journal of Law, Technology and Society* 6 (2009), Nr. 1

[Napper und Bientinesi 2009] Napper, Jeffrey ; Bientinesi, Paolo: Can Cloud Computing Reach the Top 500? In: *Proceedings of the combined Workshops on Unconventional high Performance Computing Workshop plus Memory Access Workshop*, 2009 (UCHPC-MAW '09), S. 17–20

[Neto 2011] Neto, Paulo: Demystifying Cloud Computing / Faculdade de Engenharia da Universidade do Porto. 2011. – Technischer Bericht

[Newton 2007] Newton, Keenan: The Policy Injection Application Block. In: *The Definitive Guide to the Microsoft Enterprise Library.* Apress, 2007, S. 391–415

[Niemann 2001] Niemann, Heiko: EKKIS-04: Transaktionskonzepte / Carl von Ossietzky Universität Oldenburg. 2001. – Technischer Bericht

[Ossher 2009] Ossher, Harold: *Transactions on Aspect-Oriented Software Development V*. Springer-Verlag, 2009

[Ossher und Tarr 2000] Ossher, Harold ; Tarr, Peri: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In: *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000

[Ozsu und Valduriez 1999] Ozsu, M. T. ; Valduriez, Patrick: *Principles of Distributed Database Systems (2nd Edition)*. Prentice Hall, 1999

[Pepper 2007] Pepper, Peter: Vererbung. In: *Programmieren lernen*. Springer-Verlag, 2007 (eXamen.press), S. 223–236

[Pernul und Unland 2003] Pernul, Günther ; Unland, Rainer: *Datenbanken im Unternehmen: Analyse, Modellbildung und Einsatz*. Oldenbourg Wissenschaftsverlag, 2003

[Pessemier et al. 2006] Pessemier, Nicolas ; Seinturier, Lionel ; Coupaye, Thierry ; Duchien, Laurence: A Model for Developing Component-Based and Aspect-Oriented Systems. In: *Software Composition* Bd. 4089. Springer-Verlag, 2006, S. 259–274

[Peterson 2011] Peterson, Nils: *Vergleich und Evaluation zwischen modernen und traditionellen Datenbankkonzepten unter den Gesichtspunkten Skalierung, Abfragemöglichkeit und Konsistenz*. Diplomica Verlag GmbH, 2011

[Pocatilu et al. 2010] Pocatilu, Paul ; Alecu, Felician ; Vetrici, Marius: Measuring the Efficiency of Cloud Computing for e-Learning Systems. In: *WSEAS Transactions on Computers* 9 (2010), Nr. 1, S. 42–51

[Pohl et al. 2005] Pohl, Klaus ; Böckle, Günter ; Linden, Frank J. d.: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005

[Prehofer 1997] Prehofer, Christian: Feature-oriented programming: A Fresh Look at Objects. In: *ECOOP '97 - Object-Oriented Programming*. Springer-Verlag, 1997 (Lecture Notes in Computer Science), S. 419–443

[Rahimi und Haug 2010] Rahimi, Saeed K. ; Haug, Frank S.: *Distributed Database Management Systems: A Practical Approach*. John Wiley and Sons, 2010

[Redkar 2009] Redkar, Tejaswi: Introducing Cloud Services. In: *Windows Azure Platform*. Apress, 2009, S. 1–51

[Rodewig und Negm-Awad 2006] Rodewig, Klaus M. ; Negm-Awad, Amin: *Objective-C und Cocoa - Programmieren unter Apple Mac OSX*. SmartBooks, 2006

[Rodriguez-Martinez et al. 2010] Rodriguez-Martinez, Manuel ; Seguel, Jaime ; Greer, Melvin: Open Source Cloud Computing Tools: A Case Study with a Weather Application. In: *Proceedings of the 3rd International Conference on Cloud Computing*, 2010 (CLOUD '10), S. 443–449

[Saake et al. 2005] Saake, Gunter ; Heuer, Andreas ; Sattler, Kai-Uwe: *Datenbanken: Implementierungstechniken.* mitp, 2005

[Saltzer und Kaashoek 2009] Saltzer, Jerome H. ; Kaashoek, Frans: *Principles of Computer System Design: An Introduction.* Morgan Kaufman Publ Inc, 2009

[Santos et al. 2009] Santos, Nuno ; Gummadi, Krishna P. ; Rodrigues, Rodrigo: Towards trusted Cloud Computing. In: *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009 (HotCloud'09)

[Schadt et al. 2010] Schadt, Eric E. ; Linderman, Michael D. ; Sorenson, Jon ; Lee, Lawrence ; Nolan, Garry P.: Computational Solutions to large-scale Data Management and Analysis. In: *Nature Reviews Genetics* 11 (2010), Nr. 9, S. 647–657

[Schatten et al. 2010] Schatten, Alexander ; Demolsky, Markus ; Winkler, Dietmar ; Biffl, Stefan ; Gostischa-Franta, Erik ; Östreicher, Thomas: Komponentenorientierte Software-Entwicklung. In: *Best Practice Software-Engineering.* Spektrum Akademischer Verlag, 2010, S. 301–375

[Schill und Springer 2007] Schill, Alexander ; Springer, Thomas: Transaktionen. In: *Verteilte Systeme.* Springer-Verlag, 2007, S. 95–119

[Schubert 2007] Schubert, Matthias: Konkurrierende Zugriffe. In: *Datenbanken.* Teubner, 2007, S. 332–351

[Seimert 2010] Seimert, Winfried: *Windows live.* Hüthig Jehle Rehm, 2010

[Silberschatz et al. 2001] Silberschatz, Abraham ; Korth, Henry F. ; Sudarshan, S.: *Database System Concepts.* McGraw-Hill Book Company, 2001

[Singh 2009] Singh, Shio K.: *Database Systems: Concepts, Design and Applications.* Pearson Education India, 2009

[Smaragdakis und Batory 2002] Smaragdakis, Yannis ; Batory, Don: Mixin Layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. In: *ACM Trans. Softw. Eng. Methodol.* 11 (2002), Nr. 2, S. 215–255

[Sosinsky 2011] *Kapitel* 1. In: Sosinsky, Barrie: *Cloud Computing Bible.* Wiley Publishing, Inc., 2011

[Spinczyk und Lohmann 2005] Spinczyk, Olaf ; Lohmann, Daniel: Aspect-Oriented Programming with C++ and AspectC++. In: *4th International Conference on Aspect-Oriented Software Development*, 2005 (AOSD '05)

[Spinczyk et al. 2005] Spinczyk, Olaf ; Lohmann, Daniel ; Urban, Matthias: AspectC++: an AOP Extension for C++. In: *Software Developer's Journal* (2005), Nr. 5, S. 68–76

[Sriram und Khajeh-Hosseini 2010] Sriram, Ilango ; Khajeh-Hosseini, Ali: Research Agenda in Cloud Technologies. In: *1st ACM Symposium on Cloud Computing*, 2010 (SOCC '10)

[Steyer 2010] Steyer, Ralph: *Das JavaScript-Handbuch: Einführung, Praxis und Referenz.* Pearson Deutschland GmbH, 2010

[Tarr et al. 1999] Tarr, Peri ; Ossher, Harold ; Harrison, William ; Jr., Stanley M. S.: N degrees of separation: multi-dimensional separation of concerns. In: *Proceedings of the 21st international conference on Software engineering*, 1999 (ICSE '99), S. 107–119

[Taubenfeld und Francez 1984] Taubenfeld, Gadi ; Francez, Nissim: Proof Rules for Communication Abstractions. In: *Foundations of Software Technology and Theoretical Computer Science* Bd. 181. Springer-Verlag, 1984, S. 443–465

[Thüm et al. 2012] Thüm, Thomas ; Apel, Sven ; Kästner, Christian ; Kuhlemann, Martin ; Schaefer, Ina ; Saake, Gunter: Analysis Strategies for Software Product Lines / School of Computer Science, University of Magdeburg. 2012. – Technischer Bericht

[Vossen 2000] Vossen, Gottfried: *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. Oldenbourg Wissenschaftsverlag, 2000

[Wald 2010] Wald, Heather: Cloud Computing for the Federal Community. In: *Information Assurance Newsletter* 13 (2010), Nr. 2, S. 10

[Wand et al. 2004] Wand, Mitchell ; Kiczales, Gregor ; Dutchyn, Christopher: A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. In: *ACM Trans. Program. Lang. Syst.* 26 (2004), Nr. 5, S. 890–910

[Ward und Dafoulas 2006] Ward, Patricia ; Dafoulas, George: *Database Management Systems*. Thomson Learning, 2006

[Wegner 1990] Wegner, Peter: Concepts and Paradigms of Object-Oriented Programming. In: *SIGPLAN OOPS Mess.* 1 (1990), Nr. 1, S. 7–87

[Weikum und Vossen 2002] Weikum, Gerhard ; Vossen, Gottfried: *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufman Publ Inc, 2002

[Wilshusen 2010] Wilshusen, Gregory: *Information Security: Federal Guidance Needed to Address Control Issues with Implementing Cloud Computing*. DIANE Publishing, 2010

[Yan et al. 2009] Yan, Liang ; Rong, Chunming ; Zhao, Gansen: Strengthen Cloud Computing Security with Federal Identity Management Using Hierarchical Identity-Based Cryptography. In: *Proceedings of the 1st International Conference on Cloud Computing*, 2009 (CloudCom '09), S. 167–177

[Youseff et al. 2008] Youseff, Lamia ; Butrico, Maria ; Silva, Dilma D.: Toward a Unified Ontology of Cloud Computing. In: *Grid Computing Environments Workshop*, 2008 (GCE '08), S. 1–10

[Zhang et al. 2010] Zhang, Qi ; Cheng, Lu ; Boutaba, Raouf: Cloud Computing: State-of-the-Art and Research Challenges. In: *Journal of Internet Services and Applications* 1 (2010), Nr. 1, S. 7–18

[Zhong und Kambayashi 1989a] Zhong, Xingguo ; Kambayashi, Yahiko: Timestamp Ordering Concurrency Control Mechanisms for Transactions of Various Length. In: *Foundations of Data Organization and Algorithms* Bd. 367. Springer-Verlag, 1989, S. 505–516

[Zhong und Kambayashi 1989b] Zhong, Xingguo ; Kambayashi, Yahiko: Timestamp Ordering Concurrency Control Mechanisms for Transactions of Various Length. In: *3rd International Conference, FODO 1989 on Foundations of Data Organization and Algorithms*, Springer-Verlag, 1989, S. 505–516

[Zwienenberg und Willems 2010] Zwienenberg, Righard J. ; Willems, Eddy: Attacks From the Inside ... In: *20th Virus Bulletin International Conference* (2010)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 14. September 2012

Ivonne Schröter