Bakkalaureusarbeit

Eine verteilte Architektur zur Simulation von UML Aktivitätsdiagrammen

Student:	Steven Reinisch
Universität:	Bauhaus Universität Weimar
Fakultät:	Medien
Fachbereich:	Web Technology and Information Systems
Betreuer:	Prof. Dr. Benno Stein, Dipl.-Informatiker Martin Potthast
Abgabedatum:	11.September 2007

Für
Margarete & Paul
und
Michaela & Marcel

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bakkalaureusarbeit selbständig angefertigt, anderweitig nicht für Prüfungszwecke vorgelegt, alle zitierten Quellen angegeben sowie wörtliche und sinngemäße Zitate gekennzeichnet habe.

Steven Reinisch

Weimar, den 10. September 2007

Zusammenfassung

In dieser Arbeit geht es um die Erstellung einer nachrichtenorientierten Middleware (MOM) auf Basis eines relationalen Datenbankmanangementsystems (RDBMS). Eine Middleware bietet eine Abstraktionsschicht über ein verteiltes System, die die Komplexität der Benutzung eines solchen Systems verbirgt. Ein verteiltes System ist eine Gruppe von Maschinen, die in Zusammenarbeit ein Problem lösen und dem Nutzer wie eine einzige Maschine erscheinen. Eine nachrichtenorientierte Middleware erhöht die Verlässlichkeit der Benutzung eines verteilten Systems durch die persistente Zwischenspeicherung von Nachrichten, die die verschiedenen Maschinen zur gegenseitigen Kommunikation versenden. Die MOM dient der transparenten, verteilten Ausführung von Information Retrieval (IR) Prozessen auf mehreren Maschinen. Die auszuführenden IR-Prozesse werden mit Hilfe von UML 2 Aktivitätsdiagrammen modelliert, die von der MOM ausgewertet und simuliert werden.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Umgang mit der alltäglicher Datenflut wird es immer schwieriger Informationsbedürfnisse adäquat zu befriedigen. Wichtige Informationen gehen in großen Datenmengen verloren und der Mensch ist selbst nicht mehr in der Lage diese zu verarbeiten. Zwar können zum Auffinden von Informationen generische Suchmaschinen wie Google [11] benutzt werden, aber die von ihnen zur Verfügung gestellten Resultate sind oft so zahlreich und unstrukturiert, dass viele Nutzer dennoch überfordert sind. Mit der rechnergestützten Bewältigung dieser Datenflut beschäftigt sich die Technologie des Information-Retrieval (IR). Die IR-Technologie ist jedoch nicht als ein Universalverfahren zur Bewältigung der Datenflut zu verstehen sondern als eine Zusammenfassung verschiedener Verfahren, die sich mit der Wiederauffindung von Informationen in großen Datenmengen beschäftigen.
Bei der Anwendung von IR-Verfahren gibt es zwei Herausforderungen: ihre Vielfalt und Komplexität einerseits und ihre rechenintensive Ausführung andererseits. Die Bewältigung der ersten Herausforderung setzt ein hohes Maß an Erfahrung voraus. Die einzelnen Verfahren müssen je nach zu befriedigendem Informationsbedürfnis ausgewählt und in einen Gesamtprozess integriert werden. Zur Überwindung der zweiten Herausforderung bedarf es einer Laufzeitumgebung, die die parallele Ausführung der einzelnen Verfahren eines solchen IR-Prozesses ermöglicht, um eine für den Nutzer akzeptable Ausführungsgeschwindigkeit zu erzielen.
Einen Lösungsansatz bietet *TIRA (Text Based Information Retrieval Architecture)* [25] [18].
TIRA ist eine Software-Architektur, die auf dem Java Development Kit [31] und einer Menge von IR-Verfahren in Form von *IR-Modulen* aufbaut. Wie in Abb. 1.1 zu sehen, ist TIRA in Schichten aufgeteilt. In der ersten Schicht (*PIM specification*) ist eine grafische Oberfläche (GUI) implementiert, die die Integration der IR-Module zu einem IR-Prozess ermöglicht. Das mit Hilfe der GUI erstellte Modell eines IR-Prozesses basiert auf der Syntax von Aktivitätsdiagrammen der *Unified Modeling Language (UML) 2* [20]. Die zur Modellierung eines IR-Prozesses notwendige Erfahrung stellt TIRA in Form von *Constraints* zur Verfügung. Diese überprüfen während der Modellierung, ob die IR-Module richtig miteinander verknüpft wurden. Gemäß dem Paradigma der *Model Driven Architecture (MDA)* [19] sind zur Erstellung des Modells keinerlei Kenntnisse über die Laufzeitumgebung (Plattform), in der der modellierte IR-Prozess ausgeführt wird, erforderlich. Es ist ein plattformunabhängiges Modell (*Platform Independent Model - PIM*). Damit der modellierte IR-

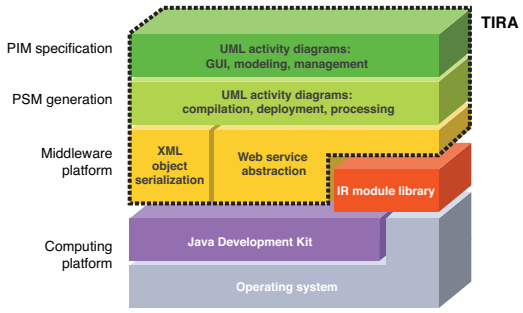

Abbildung 1.1: Die Schichtenarchitektur von TIRA [18]

Prozess in der von TIRA zur Verfügung gestellten Laufzeitumgebung ausgeführt werden kann, muss das PIM in ein Modell transformiert werden, dass die technischen Details der Laufzeitumgebung berücksichtigt. Solch ein Modell ist plattformspezifisch. Es wird als *Plattform Specific Model (PSM)* bezeichnet. Die Generierung des PSM aus einem PIM ist in der zweiten Schicht (*PSM generation*) implementiert. In der dritten Schicht (*Middleware platform*) ist eine Middleware implementiert. Sie ermöglicht eine transparente verteilte Ausführung der verschiedenen IR-Module, indem sie die verteilte Natur der Laufzeitumgebung verbirgt. Die verteilte Laufzeitumgebung ist in Abb. 1.2 zu sehen. Nutzer stellen mit Hilfe eines Browsers Anfragen an das System. Die Anfragen werden vom *Gateway* entgegengenommen. Bei Eintreffen einer Anfrage initialisiert das Gateway einen IR-Prozess und übernimmt die Koordination seiner Ausführung. Die Ausführung wird an die *JobServer*[1] delegiert. Ein JobServer bietet ein oder mehrere IR-Module als einen Dienst (*WebService*) im Internet an, der vom Gateway genutzt wird. Die von den IR-Modulen benötigten Eingabedaten werden ihnen in Form von XML-Dokumenten zur Verfügung gestellt.

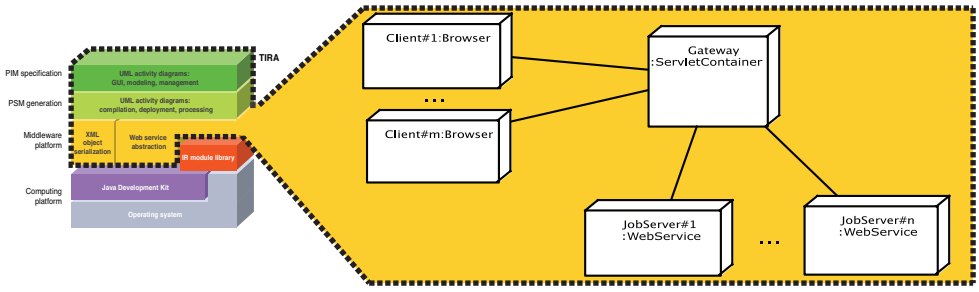

Abbildung 1.2: Darstellung der verteilten Laufzeitumgebung von TIRA: Im rechten Teil des Bildes sind Details der Schicht *Middleware platform* anhand eines vereinfachten UML Deploymentdiagramms dargestellt: Das Gateway nimmt die Nutzeranfragen von mehreren Browsern entgegen. Die Ausführung eines IR-Prozesses wird an mehrere JobServer delegiert, die die Funktionalitäten der verschiedenen IR-Module in Form von WebServices anbieten.

1.1 Ziel der Arbeit

Wie in Abb. 1.2 zu sehen ist, kommuniziert das Gateway *direkt* mit den JobServern. Die direkte Kommunikation ist jedoch nicht verlässlich. Ist ein JobServer während der Ausführung eines

[1] Ein JobServer dient der Bearbeitung von *Jobs*. Ein Job ist die Ausführung eines IR-Moduls.

IR-Prozesses kurzzeitig nicht erreichbar, kommt es zum Verlust des Aufrufes und der gesamte IR-Prozess kann nicht ausgeführt werden. Um eine verlässliche Kommunikation zu ermöglichen, muss eine *Nachrichtenwarteschlange (Messagequeue)* zwischen Gateway und JobServer geschaltet werden. Die resultierende Laufzeitumgebung ist in Abb. 1.3 zu sehen. Das Gateway kommuniziert *indirekt* mit den JobServern: die Nachrichten über die Ausführung eines IR-Moduls übergibt das Gateway der Messagequeue. In der Messagequeue werden die Nachrichten solange persistent gespeichert, bis die JobServer in der Lage sind, diese zu verarbeiten. Eine Middleware, die Nachrichten persistent speichert, heisst *nachrichtenorientierte Middleware (Message Oriented Middleware - MOM)*.
Ziel dieser Arbeit ist die Umgestaltung der bestehenden Middleware, so dass die Kommunikation zwischen Gateway und IRModulen bzw. JobServern auf Basis einer Messagequeue erfolgt. Im Gegensatz zur alten Architektur (siehe Abb. 1.2) wird die Koordination der Ausführung der verschiedenen IR-Module nicht vom Gateway sondern von der Messageque übernommen. Um die Konsistenz der zur Koordination notwendigen Daten zu gewährleisten, muss der parallele Zugriff auf die Messagequeue durch die verschiedenen JobServer und das Gateway kontrolliert werden. Die Kontrolle paralleler Zugriffe auf eine Ressource wird als *Mehrbenutzersynchronisation* bezeichnet. Aufgrund der Komplexität der Implementierung einer Mehrbenutzersynchronisation ist es sinnvoll, auf ein diese Kontrollmechanismen umsetzendes System zurückzugreifen. Aus diesem Grund wird als Basis der Implementierung der Messagequeue ein *Datenbankmanagementsystem (DBMS)* verwendet. Da die in der ersten Schicht (*PIM specification*) implementierte grafische Oberfläche

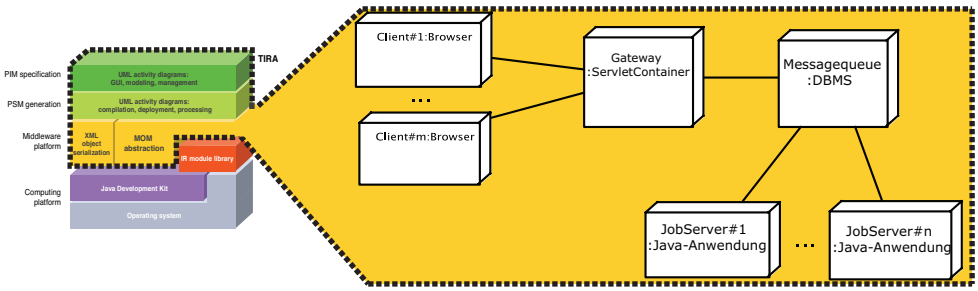

Abbildung 1.3: Darstellung der *zu erstellenden* Laufzeitumgebung: Die Kommunikation zwischen Gateway und JobServer erfolgt indirekt über eine Nachrichtenwarteschlange (Messagequeue). Die Messagequeue wird auf Basis eines Datenbankmanagementsystems (DBMS) implementiert. Die JobServer sind einfache Java-Anwendungen, die mit der Messagequeue interagieren.

(GUI) weiterhin zur Modellierung von IR-Prozessen verwendet werden soll, muss die zweite Schicht (*PSM generation*) reimplementiert werden, so dass die neu erstellte Laufzeitumgebung in der Lage ist, das mit Hilfe der GUI erstellte Modell eines IR-Prozesses auszuwerten und zu simulieren.

1.2 Struktur der Arbeit

In Kapitel 2 werden zunächst die zum Verständnis der Arbeit grundlegenden Begriffe eingeführt. In Kapitel 3 wird einleitend auf die Unterschiede zwischen objektorientierter und komponentenorientierter Softwareentwicklung eingegangen. Anschließend wird gezeigt, inwiefern die Abbildung

der Konzepte der Komponentenorientierung auf die Java- bzw. MySQL-Plattform möglich ist. An die Erläuterung der Architektur einer MOM schliesst sich die Analyse der Anforderungen an ein verteiltes System an. Ein weiterer Abschnitt beschäftigt sich mit Konzepten zur Simulation von UML Aktivitätsdiagrammen. Abgeschlossen wird Kapitel 3 mit der Darstellung des Ablaufs der Transformation eines Platform Independent Model (PIM) in ein Platform Specific Model (PSM). Kapitel 4 zeigt wie die MOM und die Transformation des PIMs in das PSM implementiert wurden. Eine abschließende Diskussion der Ergebnisse wird in Kapitel 5 durchgeführt.

Kapitel 2

Begriffsbildung

In den folgenden Abschnitten sollen die theoretischen Grundlagen bzgl. der in Abb. 1.1 dargestellten Schichtenarchitektur gelegt werden. Bezogen auf die Schichtenarchitektur wird dabei *Buttom-Up* vorgegangen: zunächst wird auf die unterste Schicht (*Middlewareware platform*) von TIRA Bezug genommen. Ziel ist das Verständnis der Begriffe *verteiltes System* und *(nachrichtenorientierte) Middleware*. Des weiteren soll geklärt werden, wie sich Informationssyteme softwaretechnisch strukturieren lassen.
In Abschnitt 2.3 wird die Schicht *PSM Generation* näher erläutert. Nach diesem Abschnitt ist der Leser in der Lage, die Begriffe *Platform Independent Model (PIM)* und *Platform Specific Model (PSM)* einzuordnen.
Den Abschluss bildet Abschnitt 2.4. Er erläutert *UML Aktivitätsdiagramme*, auf deren Basis die Modellierung von IR-Prozessen mit der in der obersten Schicht (*PIM specification*) implementierten grafischen Oberfläche geschieht.

2.1 Verteiltes System

Ein verteiltes System ist eine Menge von Maschinen, die unabhängig voneinander arbeiten und dem Nutzer das Gefühl vermitteln, er interagiere mit einer einzigen Maschine [2].
Die einzelnen Computer sind über ein Datenaustauschsystem miteinander verbunden. Das Datenaustauschsystem stellt die Infrastruktur für die Kommunikation der Computer untereinander zur Verfügung. Ein Datenaustauschsystem ist z.B. ein Local Area Network (LAN) oder das Internet. Verteilte Systeme sind durch zwei Merkmale gekennzeichnet:

1. Aufteilung
2. Transparenz

Aufteilung heisst, dass nicht jede Maschine die gesamte, durch das verteilte System zur Verfügung gestellte, Funktionalität erbringt. Jede Maschine übernimmt eine Teilaufgabe zur Erzielung des Gesamtergebnisses. So wird z.B. in einem verteilten System, das einem Onlineshop zu Grunde

liegt, die Überprüfung der Gültigkeit einer Kreditkarte von einer anderer Maschine durchgeführt als der, die eine Auftragsbestätigung per Email verschickt. Durch Aufteilung ist eine echte parallele Verarbeitung möglich. Des Weiteren ist die Möglichkeit gegeben, das System durch hinzufügen zusätzlicher Maschinen erweitern zu können, ohne das die Funktionalität dadurch beeinträchtigt wird.
Durch die *Transparenz* bleibt dem Nutzer bzw. einer Applikation die Aufteilung des Systems verborgen. Der Zugriff auf Ressourcen wie z.B. Dateien erfolgt ohne Kenntnisse des Ortes, an dem sie gespeichert sind (*Ortstransparenz*). Des Weiteren wird noch zwischen folgenden Arten von Transparenz unterschieden, die kennzeichnend für ein verteiltes System sind [2]:

- *Zugriffstransparenz* ermöglicht einen einheitlichen Zugriff auf lokale und entfernte Ressourcen.
- *Parallelitätstransparenz* erlaubt den gleichzeitigen Zugriff auf Ressourcen, ohne dass sich diese Zugriffe gegenseitig beeinflussen.
- *Replikationstransparenz* ebnet den Weg für das Hinzufügen zusätzlicher Maschinen zur Vergrößerung der Ausfallsicherheit, ohne dass dies der Nutzer oder eine Applikation mitbekommt.
- *Verlagerungstransparenz (Migration)* befähigt ein verteiltes System, Ressourcen ohne Beeinträchtigung eines Nutzers/einer Applikation zu verschieben.
- *Fehlertransparenz* verbirgt Ausfälle einzelner Teile des Systems, so dass die Arbeit eines Nutzers/einer Applikation nicht beeinträchtigt wird.
- *Skalierungstransparenz* erlaubt die Vergrößerung des Systems, wobei die Konfiguration des Systems aus Sicht eines Nutzers/einer Applikation nicht verändert werden muss.

2.1.1 Middleware

Um Applikationen auf Basis eines verteilten Systems implementieren zu können, ist eine Softwareschicht sinnvoll, die von der verteilten Natur des dieser Applikationen zu Grunde liegende Systems abstrahiert. Solch eine Software-Schicht heisst *Middleware*. Wie in Abb. 2.1 zu sehen, ist sie zwischen der höheren Ebene aus Benutzern und Applikationen und der darunter liegenden Ebene aus Betriebsystemen platziert. Die Middleware unterstützt den Entwickler verteilter Applikationen, indem sie ihm eine einheitliche Schnittstelle über die verschiedenen (heterogenen), in einem Netzwerk verteilten Maschinen und ihren Prozessen anbietet. Sie setzt die oben genannten Arten von Transparenz um. Eine verteilte Applikation ist eine Software, die zur Erfüllung ihrer Aufgabe von einem verteilten System Gebrauch macht.

2.1.2 Kommunikationsparadigmen

Die Kommunikation in einem verteilten System kann auf zwei Arten erfolgen: *synchron* und *asynchron*. Bei der synchronen Kommunikation wartet der Sender solange, bis die Kommunikation mit dem Empfänger abgeschlossen ist. Der Sender kann in seinem Programmfluss nicht fortfahren - er blockiert. Im Falle asynchroner Kommunikation fährt der Sender unmittelbar nach dem Erhalt

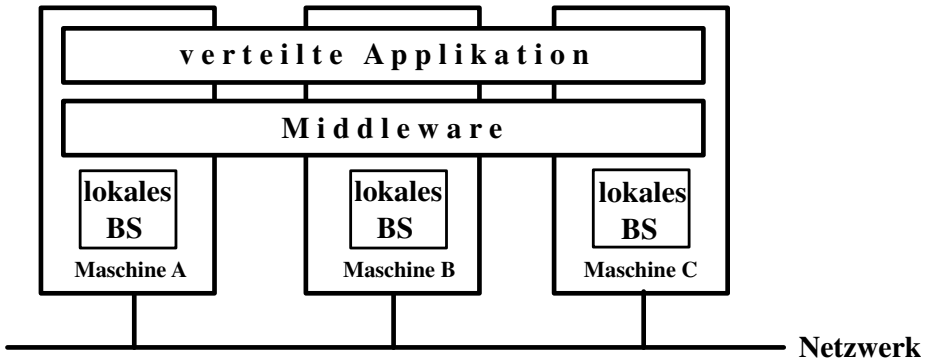

Abbildung 2.1: Eine Middleware ist eine Softwareschicht die von der heterogenen, über ein Netzwerk verteilten Hardware abstrahiert. Jede im Netzwerk verteilte Maschine verfügt über ein eigenes, lokales Betriebssystem (BS) und ist somit autonom. [2]

einer Bestätigungsnachricht über die Akzeptanz des Aufrufes in seinem Programmfluss fort.
Die Klassifikation der Kommunikation in verteilten Systemen erfolgt nicht nur hinsichtlich des Synchronisationsverhaltens der Teilnehmer. Es wird des Weiteren bzgl. der Speicherung nicht übermittelter Nachrichten unterschieden. Die Einteilung erfolgt in *transienter* und *persistenter* Kommunikation. Bei persitenter Kommunikation wird die Nachricht dauerhaft zwischengespeichert, bis der Empfänger in der Lage ist, sie zu empfangen. Bei transienter Kommunikation geschieht dies nicht - ist der Empfänger nicht erreichbar, geht die Nachricht verloren.

2.1.3 Message Oriented Middleware (MOM)

Eine Middleware, die eine persistente Kommunikation ermöglicht, heisst nachrichtenorientierte Middleware (*Message Oriented Middleware* - MOM) [2].
In einer MOM kommunizieren Applikationen miteinander, indem sie ihre Nachrichten in Warteschlangen einstellen. Diese Nachrichten müssen korrekt adressiert sein, um von der Warteschlange an das entsprechende Ziel ausgeliefert werden zu können. Der Sender erhält lediglich die Garantie, dass die Nachricht in die Warteschlange eingestellt wurde nicht jedoch, dass sie vom Empfänger gelesen wird. Eine Konsequenz ist, dass Sender und Empfänger völlig unabhängig voneinander ausgeführt werden können.

2.2 Softwaretechnische Strukturierung von Informationssystemen

In diesem Abschnitt sollen die grundlegenden softwaretechnischen Strukturierungsmerkmale der zu erstellenden Middleware-Plattform eingeführt und erläutert werden.

Zunächst werden die Begriffe *Informationssystem*, *Komponente*, *Schnittstelle* und *Kompositionsmanager* bestimmt, da Abschnitt 2.2.5 auf ihnen fußt.

2.2.1 Informationssystem

TIRA unterstützt die technische Realisierung von Informationssystemen, indem sie eine Kommunikationsinfrastruktur zur Verfügung stellt. Ein Informationssystem dient der Erfassung, Speicherung, Verarbeitung, Analyse und Distribution von Daten auf Basis elektronischer Datenverarbeitung [38].

2.2.2 Komponente

Eine (Software-)Komponente wird in [32] als eine Softwareeinheit mit vertraglich festgelegten Schnittstellen und expliziten Kontextabhängigkeiten definiert. Sie kann unabhängig angewandt und von Dritten zur Komposition weiterer Komponenten verwendet werden.

Um die Charakteristik einer Komponente vollständig zu beschreiben, werden in [29] folgende sechs Merkmale definiert:

1. Sie exportiert eine oder mehrere Schnittstellen, die im Sinne eines Vertrags garantiert sind (Design By Contract [37]).
2. Import anderer Schnittstellen - Die Komponente ist erst lauffähig, wenn alle importierten Schnittstellen zur Verfügung stehen, was die Aufgabe des *Konfigurationsmanagers* ist. Die importierten Schnittstellen sind die einzige (minimale) Annahme der Komponente über ihre Umgebung.
3. Sie versteckt ihre Implementierung und ermöglicht somit ihre Ersetzbarkeit durch Komponenten, die dieselbe Schnittstelle exportieren.
4. Sie eignet sich als Einheit der Wiederverwendung, da sie nur minimale Annahmen über die Umgebung, in der sie läuft, macht.
5. Sie lässt sich aus vorhandenen Komponenten zusammensetzen (*Komposition*).
6. Sie ist neben der Schnittstelle zentrales Element des Entwurfs und der Implementierung eines Softwaresystems.

2.2.3 Schnittstelle

In [29] wird zwischen Programm- und Benutzerschnittstelle unterschieden. Letztere dient der Interaktion eines Benutzers mit einer Komponente, erstere der Interaktion zwischen Komponenten untereinander. Jede Programmschnitstelle definiert eine Menge von Methoden und stellt eine Beschreibung eines *Dienstes* dar, der von einer anderen Komponente benutzt werden kann. Eine Komponente, die eine Programmschnittstelle exportiert, stellt den durch die Programmschnittstelle beschriebenen Dienst zur Verfügung. Wird im nachfolgenden Text von Schnittstelle gesprochen, ist eine Programmschnittstelle gemeint.

Den Zusammenhang zwischen Komponenten und Schnittstellen veranschaulicht Abb. 2.2.

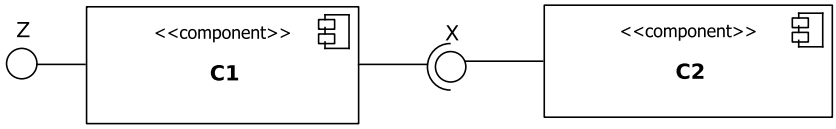

Abbildung 2.2: Zusammenhang zwischen Komponenten und Schnittstellen: Komponente C2 exportiert die Schnittstelle X. Komponente C1 importiert die Schnittstelle X und exportiert die Schnittstelle Z.

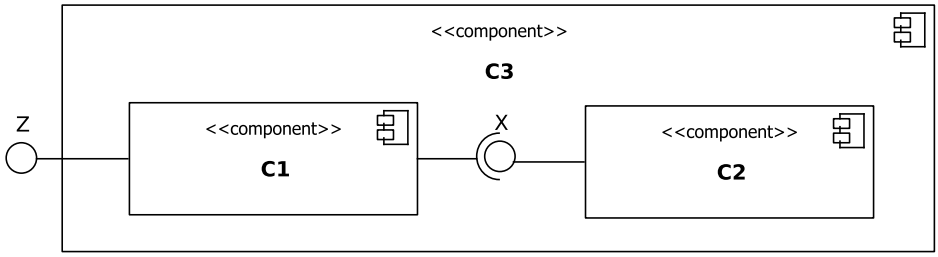

Abbildung 2.3: Die Komponente C3 ist eine *Komposition* aus den Komponenten C1 und C2. C1 und C2 sind *Subkomponenten* von C3. C3 *delegiert* Aufrufe auf Z an C1.

2.2.4 Kompositionsmanager

Der Kompositionsmanager übernimmt die Konfiguration bzw. Komposition der Komponenten. Die Konfiguration ist der Vorgang des Verknüpfens der verschiedenen Komponenten: Komponenten, die eine bestimmte Schnittstelle importieren, müssen mit der diese Schnittstelle exportierenden Komponente verknüpft werden, um den von dieser Schnittstelle beschriebenen Dienst nutzen zu können (Abb. 2.4). Die Komposition ist die Zusammensetzung mehrerer Komponenten zu einer neuen Komponente (siehe Abb. 2.3). Die zur Komposition verwendeten Komponenten heißen *Subkomponenten*. Von Subkomponenten exportierte Schnittstellen können von der komponierten Komponente exportiert werden. Ist dies der Fall, werden Aufrufe an die Subkomponente *delegiert*.

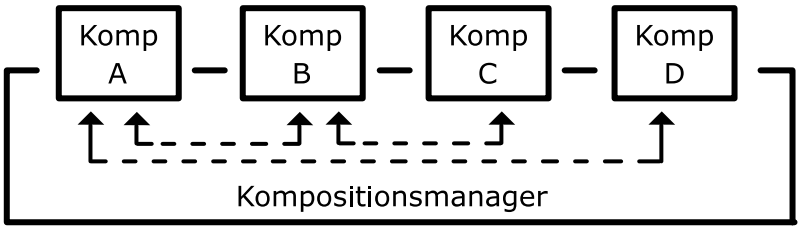

Abbildung 2.4: Der Kompositionsmanager übernimmit die Verknüpfung der Komponenten. Ein gestrichelter Pfeil zwischen zwei Komponenten steht für deren Verknüpfung.

2.2.5 Softwarearchitektur für Informationsysteme

Die Struktur eines Softwaresystems wird auch als Softwarearchitektur bezeichnet. Das Zusammenspiel aus Softwarearchitektur und der Hardware, auf der sie eingesetzt wird, wird in [29] als

Systemarchitektur bezeichnet. Sie besteht aus zwei Einheiten:

1. Die Architektur der technischen Infrastruktur (*TI-Architektur*) beschreibt die Struktur und das Zusammenspiel physischer Geräte (Rechner, Netzleitungen etc.) und installierter Systemsoftware (Betriebssystem, Anwendungsserver etc.).
2. Die Softwarearchitektur beschreibt das System bestehend aus Komponenten, deren Schnittstellen und dem Zusammenspiel zwischen beiden.

Innerhalb der Softwarearchitektur wird nochmals unterschieden zwischen der *Anwendungs-* und der *Technikarchitektur*:

1. Die Anwendungsarchitektur (*A-Architektur*) beschreibt die Struktur der Software aus Sicht der Anwendung. Technische Sachverhalte wie der verwendete Persistenz- oder Transaktionsmechanismus fließen nicht in die A-Architektur ein. Komponenten der A-Architektur (*A-Komponenten*) sind technikfrei, sie konzentrieren sich lediglich auf die Anwendungslogik.
2. Die Technikarchitektur (*T-Architektur*) bestimmt all die Komponenten des Systems, die von der Anwendung unabhängig sind, z.B. Persistenzschicht oder Fehlerbehandlung. Sie verbindet die A-Architektur mit der TI-Architektur und definiert den Umgang mit Betriebssystemresourcen und Nachbarsysteme wie z.B. Datenbanken. Sie beschreibt also eine Laufzeitumgebung, in der die A-Komponenten laufen.

Abb. 2.5 veranschaulicht den Zusammenhang zwischen A-, T- und TI-Architektur.

Die strikte Trennung von Anwendung und Technik hilft, komplexe Anwendungen beherrschbar zu machen und führt in Anbetracht der oftmals schnelleren Entwicklung der Technik gegenüber der Anwendung zu einer besseren Wart- und Erweiterbarkeit. Die Anwendungslogik kann unabhängig von der verwendeten Technik entwickelt werden.

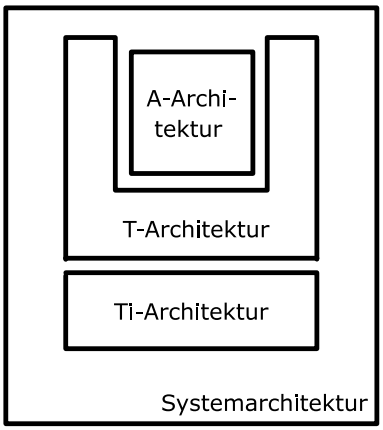

Abbildung 2.5: Zusammenhang zwischen A-, T- und TI-Architektur

2.3 Model Driven Software Development & Model Driven Architecture

Wie einleitend erwähnt, erfolgt die Spezifikation des IR-Prozesses unabhängig von der Laufzeitumgebung in der er letztendlich ausgeführt wird. Es tauchten die Begriffe *Plattform*, *PIM* (Platform Independent Model), *PSM* (Platform Specific Model) sowie der der *Transformation* zwischen diesen auf. In diesem Abschnitt werden die erwähnten Begriffe näher erläutern und in den Kontext der *Modellgetriebenen Softwareentwicklung (MDSD - Model Driven Software Development)* bzw. *MDA (Model Driven Architecture)* eingeordnet. Bezogen auf die Schichtenarchitektur (Abb. 2.6) befinden wir uns in der Schicht *PSM generation*.

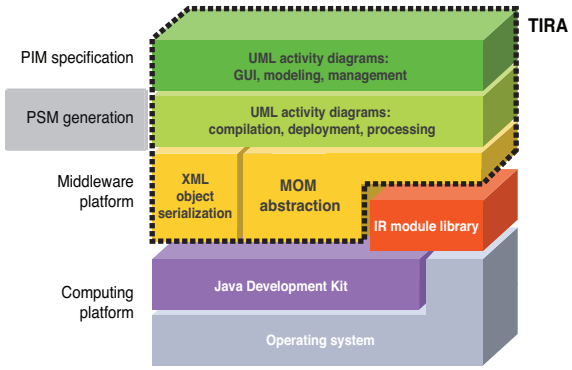

Abbildung 2.6: In diesem Abschnitt wird die Schicht *PSM Generation* näher erläutert.

MDSD beschreibt eine Vorgehensweise der Softwareentwicklung bei der die fachliche Spezifikation der Systemfunktionalität klar von der Spezifikation der Implementierung dieser Funktionalität getrennt wird. Die Spezifikation der Systemfunktionalität erfolgt mit Hilfe von formalen Modellen (z.B. UML Modelle), die mit Hilfe von (mehrstufigen) Transformationen automatisiert in lauffähigen Code übersetzt werden. MDA [19] ist eine Standardisierungsinitiative der Object Management Group (OMG) zum Thema MDSD [17].

Es sollen lediglich die grundlegenden MDSD-Konzepte eingeführt und der Zusammenhang zwischen diesen und den MDA-Konzepten PIM und PSM aufgezeigt werden. Für eine ausführliche Darstellung der MDA sei auf [9] verwiesen. Nachfolgende Erläuterungen stützen sich auf [17].

2.3.1 Domäne, Domänenspezifische Sprache, Modell und Metamodell

Die Spezifikation eines IR-Prozesses mit Hilfe eines UML Aktivitätsdiagrammes wird allgemein als Modellierung bezeichnet. Der Ausgangspunkt bei der Modellierung im Sinne modellgetriebener Entwicklung ist immer eine *Domäne*. Eine Domäne ist ein abgegrenztes Wissensgebiet. Die Konzepte einer Domäne lassen sich mit Hilfe einer *domänenspezifischen Sprache (Domain Specific Language - DSL)* beschreiben. Es ist unbedingt notwendig, sich über die Struktur der Domäne soweit im Klaren zu sein, dass diese formalisiert werden kann. Lässt sich die Domäne nicht systematisch (formal) beschreiben, ist die Erstellung von Generierungsvorschriften (*Transformationen*)

nicht möglich. Die Formalisierung findet in Form der Definition eines *Metamodells* statt. Ein Metamodell kann auch als Grammatik einer DSL verstanden werden.

Abb. 2.7 verdeutlicht den Zusammenhang zwischen Domäne, DSL, Modell und Metamodell, den Begriffen des Konzeptraumes von MDSD.

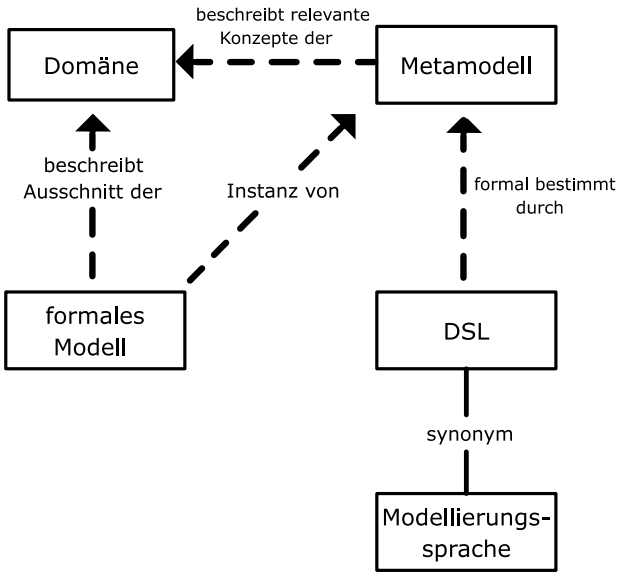

Abbildung 2.7: Zusammenhang zwischen den Konzepten Domäne, Domänenspezifische Sprache, Modell und Metamodell [17]

Die von der TIRA GUI zur Verfügung gestellte DSL bedient sich einer Auswahl von Sprachkonzepten des UML Aktivitätsmodells. Mit Hilfe dieser DSL kann die Domäne *Spezifikation und Ausführung von IR-Prozessen* beschrieben werden. Die formale Definition, das Metamodell, orientiert sich am UML Metamodell der OMG [18] [21], im folgenden als *TIRA-MM* bezeichnet.

2.3.2 Plattform

Die Plattform ist die technologische Basis einer Domäne. Die Konzepte der Domäne werden auf eine Plattform abgebildet. Die Plattform *unterstützt* die Domäne. Je besser die Plattform die Domäne unterstützt, desto geringer ist der Aufwand, die Domäne auf die Plattform abzubilden. Diese Abbildung wird von *Transformationen* durchgeführt. Sind die Konzepte einer Domäne ohne großen Aufwand abzubilden, spricht man von einer *reichhaltigen, domänenspezifischen* Plattform. Eine Plattform kann z.B. das Java Development Kit [31] sein. Eine Plattform setzt sich aus verschiedenen Bausteinen zusammen. Beispiele für solche Bausteine sind z.B. Bibliotheken, Frameworks, Komponenten oder eine Middleware. Abb. 2.8 verdeutlicht diesen Zusammenhang. Eine Plattform im Sinne von MDSD wird von nun an als *MDSD-Plattform* bezeichnet.

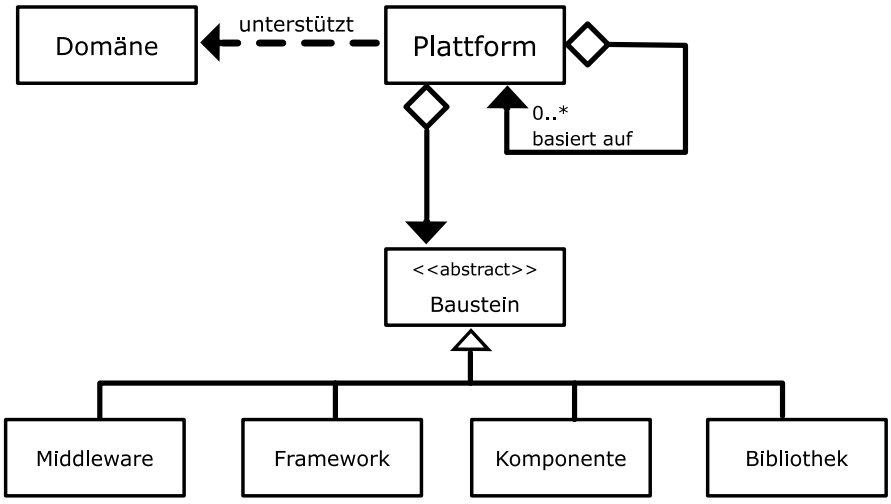

Abbildung 2.8: Zusammensetzung einer MDSD-Plattform aus Bausteinen [17]

2.3.3 Modelltransformation

Modelltransformation und Transformation werden im folgenden synonym verwendet.
Eine Modelltransformation überführt ein Quellmodell in ein Zielmodell und basiert auf einem Quellmetamodell. Es gibt zwei verschiedene Arten von Modelltransformationen:
Modell-zu-Modell-Transformationen und *Modell-zu-Plattform-Transformationen*, auch als Modell-zu-Code-Transformationen bezeichnet. Das von einer Modell-zu-Modell-Transformation erzeugte Modell kann auf einem anderen Metamodell basieren als das Quellmodell. Diese Art von Transformationen beschreiben also die Abbildung der Konstrukte des Quellmetamodells auf die Konstrukte des Zielmetamodells. Eine Modell-zu-Plattform-Transformation *kennt* die Plattform und generiert *Artefakte* (z.B. Code, der sich in ein Framework einfügt oder Konfigurationsdateien), die auf dieser Plattform basieren. Bei dieser Art von Transformation wird kein Zielmetamodell benötigt, da es sich hierbei um Makroexpansionen handelt. Der MDA-Standard sieht hierfür ein *Platform Description Model (PDM)* vor.

2.3.4 Platform Independent Model (PIM), Platform Specific Model (PSM)

Ein mit der TIRA GUI erstelltes Modell eines IR-Prozesses ist eine Instanz des TIRA-MM und macht keinerlei Aussagen über die Laufzeitumgebung in welcher der modellierte IR-Prozess ausgeführt werden soll. Es ist ein plattformunabhängiges Modell (PIM).

Damit nun dieses Modell in der von TIRA zur Verfügung gestellten Laufzeitumgebung ausgewertet und simuliert werden kann, muss es in ein PSM transformiert werden, das die spezifischen Aspekte dieser Laufzeitumgebung berücksichtigt.

2.3.5 Zusammenhang zwischen MDA und MDSD

Die Einordnung der MDA-Konzepte in den MDSD-Konzeptraum ist in Abb. 2.9 zu sehen. Die Konzepte PIM und PSM sind Spezialisierungen des Konzeptes eines formalen Modells. Im Sinne

der MDA wird die MDSD-Plattform auch durch ein formales Modell, das PDM, beschrieben. PSM und PDM beziehen sich jeweils auf eine technologische Basis.

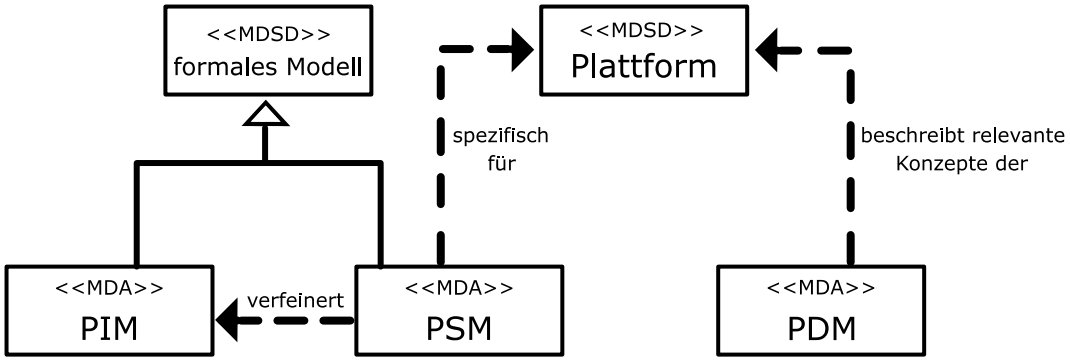

Abbildung 2.9: Abbildung der MDA-Konzepte *PIM, PSM, PDM* auf die MDSD-Konzepte *formales Modell* und *Plattform* [17]

2.4 UML Aktivitätsdiagramme

Dieser Abschnitt befasst sich mit der Grundlage der grafischen Modellierungsoberfläche, die in der obersten Schicht (*PIM specification*) (siehe Abb 2.10) implementiert ist.

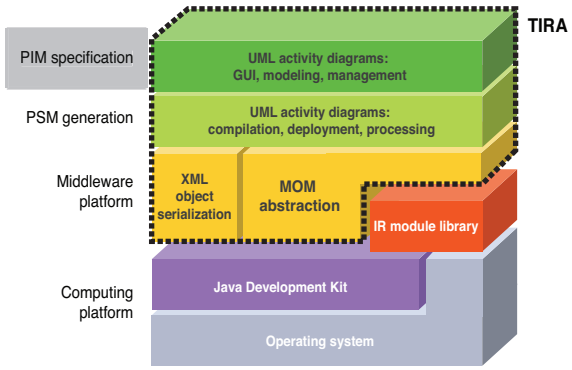

Abbildung 2.10: In diesem Abschnitt wird die Schicht *PIM specification* näher erläutert.

2.4.1 Unified Modeling Language (UML)

Mit der UML [20] ist es möglich, Softwaresysteme grafisch zu modellieren. Grundlegender Gedanke bei der Modellierung mit der UML ist die Aufteilung des zu modellierenden Systems in einzelne Modelle. Jedes dieser Modelle bildet einen Realitätsausschnitt des zu modellierenden Systems ab. Ein Modell beschreibt z.B. die statische Struktur des Systems in Form von Klassendiagrammen, ein anderes das Verhalten in Form von Aktivitätsdiagrammen.

Die aktuellle Version der UML ist 2.1.1 [21]. Die in dieser Arbeit verwendete Version ist 2.0 [20], da die von TIRA zur Verfügung gestellte grafische Modellierungsoberfläche auf der Version 2.0 basiert.

2.4.2 Aktivitätsdiagramme

Aktivitätsdiagramme fokussieren sich auf die Modellierung von prozeduralen Verarbeitungsaspekten und verwenden Konzepte basierend auf ablauforientierten Sprachen zur Definition von Geschäftsprozessen, wie z.B. die *Business Process Execution Language (BPEL)* [8], als auch auf Konzepten zur Beschreibung nebenläufiger Prozesse, wie dem Tokenkonzept in Petrinetzen [16].

Die durch Aktivitätsdiagramme modellierten Aktivitäten formen einen Flussgraph (Ablaufgraph) bestehend aus Knoten verbunden durch gerichtete Kanten. Kontroll- und Datenwerte (allgemein *Token*) fließen entlang der Kanten und werden von Knoten verarbeitet, zu anderen Knoten weitergeleitet oder temporär gespeichert [6]. Es gibt drei verschiedene Arten von Knoten im UML Aktivitätsmodell:

Aktionsknoten verarbeiten Token, die sie erhalten, und stellen Token anderen Aktionen zur Verfügung.

Kontrollknoten leiten Token durch den Ablaufgraph. Sie umfassen Knoten zur Steuerung alternativer Abläufe (Entscheidungsknoten), zur Zusammenführung alternativer Abläufe (Vereinigungsknoten), zur Modellierung der Aufspaltung von Abläufen (Parallelisierungsknoten) und zur Zusammenführung nebenläufiger Abläufe (Synchronisierungsknoten). Initialknoten markieren den Anfang, Aktivitätsendknoten das Ende einer Aktivität. Abb. 2.11 zeigt die erwähnten Kontrollknoten.

Objektknoten speichern Datenwerte temporär.

Wie in Abb. 2.12 zu sehen, besteht eine Aktivität aus mehreren *Aktionen*. Ein- bzw. Ausgabeparameter der Aktionen können als Objektknoten modelliert werden. So ist der Knoten *data* Ausgabeparameter der Aktion *action1* und Eingabeparameter der Aktion *action2*.

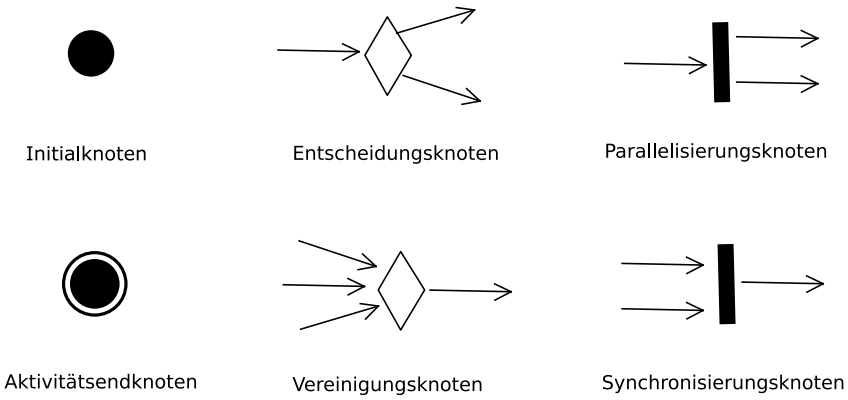

Abbildung 2.11: Arten von Kontrollknoten

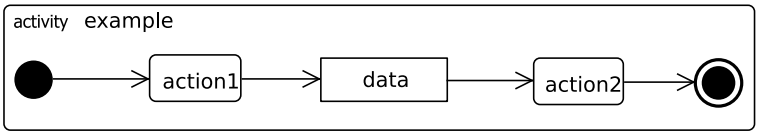

Abbildung 2.12: Eine Aktivität bestehend aus zwei Aktionen.

2.4.3 Simulation von Aktivitätsdiagrammen

„ Alle Arten von Knoten einer Aktivität sind durch Kanten miteinander verknüpft, die zusammen mit den Kontrollknoten die zeitlich logische Reihenfolge der Aktionen, und damit die möglichen Abläufe der gesamten Aktivität, festlegen. “ [16]

Der UML-Standard sieht keine konkrete Implementierung der Aktivitätsabläufe vor. Er beschreibt lediglich eine virtuelle Maschine basierend auf der Weiterleitung von Token durch den Ablaufgraph inspiriert vom Tokenkonzept in Petri-Netzen [6] [24]. Die *Laufzeitregeln* dieser virtuellen Maschine sollen das Verhalten des Ablaufgraphs lediglich vorhersagbar machen. Sie beschränken die Implementierung nicht auf die Umsetzung des Tokenkonzepts.

Kapitel 3

Paradigmen und Verfahren zur Implementierung der MOM und der Modelltransformation

In diesem Kapitel soll gezeigt werden, nach welchen Paradigmen und mit welchen Verfahren die nachrichtenorientierte Middleware (MOM) und die Modelltransformation des Platform Independent Model (PIM) in das Platform Specific Model (PSM) umgesetzt wurden.
Nach einem einführenden Vergleich der Programmierparadigmen der objektorientierten und komponentenorientierten Softwareentwicklung wird in diesem Kapitel untersucht, inwiefern die Abbildung der Konzepte der Komponentenorientierung auf die Java- bzw. MySQL-Plattform möglich ist. Bevor in Abschnitt 3.4 ein Verfahren zur Simulation von UML Aktivitätsdiagrammen gezeigt wird, soll die grundlegende Architektur der MOM erläutert werden. In Abschnitt 3.3 werden die Verfahren zur Umsetzung der Anforderungen an ein verteiltes Systems untersucht, die in Bezug auf die Umsetzung der MOM erfüllt sein sollten. Die zur Transformationen eines PIMs in ein PSM notwendigen Schritte werden abschließend eingeführt.

3.1 Komponentenorientierte Softwareentwicklung

Die folgenden Ausführungen über die Schwächen der Objektorientierung (OO) als Grundlage für die Definition der Software- und Systemarchitektur von großen, verteilten Softwaresystemen und die daraus resultierende Bevorzugung der Komponentenorientierung orientieren sich an [15].
Die Klasse als ein wesentliches Element der OO ist als grundlegende Strukturierungseinheit des Systementwurfs zu klein, zu granular. Konzepte wie Attribute und Vererbung tragen nichts zur klaren Strukturierung eines Softwaresystems bei, da sie sich auf die Implementierung und nicht auf die Konstruktion beziehen. Die Komposition (vgl. Abschnitt 2.2.2) von Klassen ist nicht möglich, da die OO kein Konzept einer zusammengesetzten Klasse bietet. Beim Entwurf und der Entwicklung von verteilten Softwaresystemen ergibt sich zudem das Problem der Objektidentität. OO-Sprachen

definieren die Identität eines Objektes über seine Speicheradresse, die in nur einem Adressraum gültig ist. In verteilten Systemen haben wir es aber mit mehreren Maschinen und somit mehreren Adressräumen zu tun. Die Frage nach der Identifizierung eines Objektes in verschiedenen Adressräumen beantwortet die OO nicht. Die Identität eines Objektes ist in verteilten Anwendungen über einen fachlichen Schlüssel, vergleichbar mit der Identität eines Tupels in einer Relation eines relationalen Datenbankmanagementsystems, festgelegt. Dieses Konzept ist in der OO jedoch nicht vorgesehen.
Zentrale Elemente der komponentenorientierten Softwareentwicklung sind, wie in Abschnitt 2.2.2 erklärt, die Komponente und die Schnittstelle. Das Merkmal der Komposition einer Komponente und ihre minimalen Annahmen über ihre Umwelt auf Basis der Schnittstelle ermöglichen eine strukturierte und skalierbare Konstruktion der Software- und Systemarchitektur. Eine detaillierte Darstellung der Konstruktion von Softwaresystemen auf der Basis von Komponenten und Schnittstellen ist in [29] und [13] nachlesbar.

Die dieser Arbeit zu Grunde liegende Vorstellung eines Softwaresystems zeigt Abb. 3.1. Das Gesamtsystem setzt sich aus Subsystemen zusammen. Jedes (Sub)System besitzt eine eigene A-Architektur. Die A-Architektur setzt sich aus A-Komponenten zusammen. Die Gesamtheit der A-Komponenten ist der Anwendungskern eines (Sub)Systems.

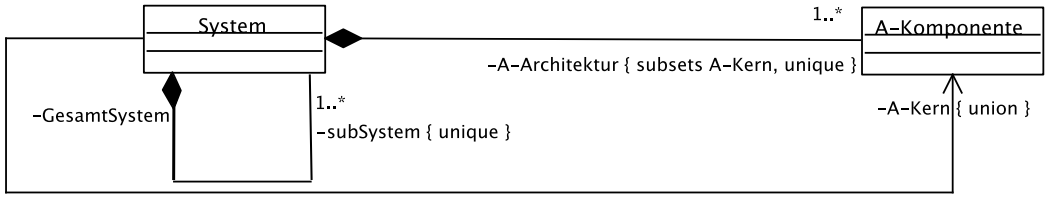

Abbildung 3.1: Modell eines Softwaresystems in Form eines UML Klassendiagramms.

A-Komponenten implementieren die Anwendungslogik, die auf einem *Datenmodell* operiert. Ein Datenmodell beschreibt Konzepte, die *Entitäten*, der realen Welt. Im Datenmodell eines Softwaresystems, das z.B. eine Autovermietung abbildet, kommen die Entitäten *Auto* und *Ausleiher* vor. Das Datenmodell des Gesamtsystems wird in disjunkte Teilbereiche aufgeteilt. Jeder Anwendungskern verwaltet einen dieser Teilbereiche [29]. Ist diese Aufteilung des Datenmodells zu grobgranular, können die Teilbereiche auch auf einzelne A-Komponenten aufgeteilt werden. Stellt sie sich als zu feingranular heraus, können die Teilbereiche zusammengefasst und auf mehrere Anwendungskerne verteilt werden.

Bei einer komponentenorientierten Entwicklung ist zu beachten, dass die Konzepte der OO nicht strikt befolgt werden. Die Entitäten, sind bloße Datencontainer - sie haben keinerlei Verhalten. Als illustrierendes Beispiel soll hier das Data-Access-Object (DAO) Pattern [30] dienen. Entitäten sollen persistent in einer Datenbank gespeichert werden. Im Fall einer strikten OO-Umsetzung würde man dies wie in im *Package oo* in Abb. 3.2 implementieren. Die Klasse *MyEntity* implementiert die Methode *save()*. Die Klasse ist selbst für die Speicherung verantwortlich. Bei einem komponentenorientierten Vorgehen wird der *Dienst* der Speicherung in eine DAO-Komponente ausgelagert (*Package ko* in Abb. 3.2). Die Klasse *MyEntity* verliert die Fähigkeit, sich zu speichern.
Dieses Vorgehen hat den Vorteil, dass das *Datenmodell* und die (Persistenz-)Technologie voneinan-

der getrennt sind. Die Architektur wird flexibler, da Anwendungslogik, die auf dem Datenmodell aufbaut, und die technologieabhängige Speicherlogik unabhängig voneinander entwickelt werden können. Das heisst nicht, dass die OO-Konzepte nicht mehr angewandt werden - sie werden nur nicht zur Definition der Struktur (Architektur) herangezogen. Die Funktionalität der Komponenten, also die Innensicht, wird mit den Konzepten der OO umgesetzt.

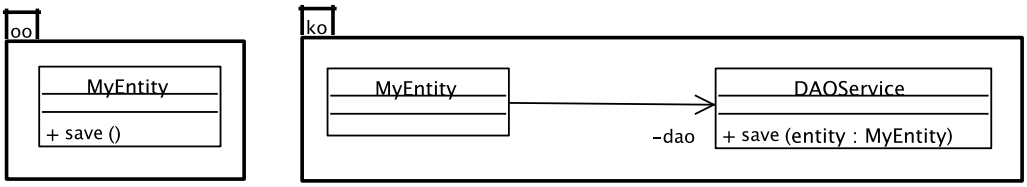

Abbildung 3.2: Gegenüberstellung eines OO- (*Packge oo*) und eines KO- Ansatzes (*Packge ko*) zur Modellierung der Speicherung eines Objektes.

3.1.1 Komponenten auf der Java-Plattform

Da Java eine OO-Sprache ist, bietet sie von sich aus keine direkte Unterstützung für die Implementierung von Komponenten. Die Semantik, die sich hinter dem Komponentenbegriff verbirgt (siehe Abschnit 2.2.2) muss mit den von Java zur Verfügung gestellten Mitteln simuliert werden. Schnittstellen haben in Java eine direkte Entsprechung durch das Sprachelement *Interface* und müssen somit nicht simuliert werden. Alle in einem *Interface* definierten Methoden beschreiben einen Dienst. Die Umsetzung der Komponentensemantik soll anhand der Abbildungen 2.2 und 2.3 veranschaulicht werden.
Die Komponenten *C1*, *C2* und die Schnittstellen Z und X aus Abb. 2.2 können in Java folgendermaßen implementiert werden:

```
1 public  Interface Z{
2
3         public void aMethodDefinedInZ();
4
5         public void setX(X aX);
6 }
```

```
1 public  class C1 implements Z{
2         private X myX;
3
4         public void aMethodDefinedInZ(){
5                 //implementation
6         }
7
8         public void setX(X aX){
9                 myX = aX;
10        }
11 }
```

```
1 public  Interface X{
2
3         public void aMethodDefinedInX();
4 }
```

```
1 public  class C2 implements X{
2
3         public void aMethodDefinedInX(){
4                 //implementation
5         }
6 }
```

Für jede exportierte Schnittstelle erfolgt ein Eintrag im *implements*-Abschnitt der Klassendefinition. Importierte Schnittstellen werden als Instanzvariablen vom Typ der jeweiligen importierten Schnittstelle modelliert. Um eine Komponente mit einer Implementierung einer importierten Schnittstelle zu versorgen, versieht man die Klasse mit einer entsprechenden *set*-Methode.

Die Komponente *C3* aus Abb. 2.3 kann in Java folgendermaßen implementiert werden:

```
1 public  class C3 implements Z{
2         private Z myZ;
3
4         public void setZ(Z aZ){
5                 myZ = aZ;
6         }
7
8         public void aMethodDefinedInZ(){
9                 //delegate call
10                myZ.aMethodDefinedInZ();
11        }
12
13        public void setX(X aX){
14                //delegate
15                myZ.setX(aX);
16        }
17 }
```

Sollen die von Subkomponenten exportierten Schnittstellen auch durch eine Komposition, wie die Komponente C3, exportiert werden, so wird dies im *implements*-Abschnitt der Klassendefinition deklariert. Aufrufe an Kompositionen werden von dieser an die entsprechende Subkomponente delegiert. Die Subkomponenten *C1* und *C2* werden, wie in den vorherigen Listings beschrieben, implementiert.
Die Konfiguration bzw. Komposition der Komponenten kann auf zwei Wegen erfolgen. Einerseits kann der entsprechende Programmcode selbst geschrieben werden. Hierzu müssen die Klassen, die die Komponenten repräsentieren, zuerst erzeugt werden. Die Verknüpfung erfolgt durch Aufruf der

entsprechenden *set*-Methoden, die durch setzen der entsprechenden Instanzvariable eine Komponente der anderen übergeben. Für die Verknüpfung der Komponente *C1* mit der Komponente *C2* sieht der entsprechende Java-Code so aus:

```
1 public void bindC2ToC1 (){
2
3         X myC2 = new C2();
4         Z myC1 = new C1();
5
6         //bind C2 to C1
7         myC1.setX(myC2);
8 }
```

Ein zweiter Weg führt zur Benutzung eines Laufzeit-Containers. Einem Laufzeit-Container werden die Klassendefinitionen, anhand derer er die zu erledigenden Verknüpfungen eigenständig bestimmt und durchführt, übergeben.

3.1.2 Komponenten auf der MySQL-Plattform

Die MySQL-Plattform ermöglicht die Formulierung von Programmen bzw. Programmroutinen in der Programmiersprache *Stored Program Language (SPL)*. SPL ist eine blockorientierte Sprache und enthält typische Befehle zur Implementierung konditionaler und iterativer Ausführung, Variablenmanipulation und Fehlerbehandlung [10]. Die in SPL definierten Programmroutinen, *Stored Procedures (SP)*, werden innerhalb des MySQL-Datenbank-Servers (MDS) ausgeführt. Da ein wesentlicher Teil der A-Architektur der MOM auf dieser Plattform entworfen und implementiert wird, soll geklärt werden, inwiefern es möglich ist, eine Komponentensemantik auf dieser Plattform umzusetzen.
Die MySQL-Plattform bietet keine Einschränkung der Sichtbarkeit von Programmroutinen bzw. Daten wie dies die Sichtbarkeitsmodifizierer *private*, *protected* und *public* auf der Java-Plattform ermöglichen. Auf einem MDS erstellte SP sind somit global auf dem ganzen Server verfügbar. Sie können an allen Stellen des SP-Codes aufgerufen werden. Des Weiteren sind alle auf einem MDS gespeicherten Daten prinzipiell an jedem Punkt der Ausführung einer SP verfügbar. Dies widerspricht der Anforderung an eine Komponente, ihre Implementierungsdetails nach außen hin zu verbergen. Theoretisch wäre denkbar, das Konzept eines Nutzers des MDS zur Einschränkung der Datenzugriffe bzw. SP-Aufrufe zu benutzen. Bei der Definition einer SP kann ein Nutzer angegeben werden, der am MDS angemeldet sein muss, um diese aufzurufen. Pro Schnittstelle würde also ein Nutzer erstellt, der als einziger in der Lage ist, die durch die Schnittstelle gruppierten SP aufzurufen. Eine Komponente die nun einen durch eine Schnittstelle beschriebenen Dienst nutzen möchte, muss unter dem der Schnittstelle entsprechenden Nutzernamen angemeldet sein. Dies hieße aber, dass eine Komponente bei der Nutzung mehrerer Dienste sich während der Laufzeit unter verschiedenen Nutzernamen anmelden müsste. Diese Art von Nutzerwechsel ist auf dem MDS nicht möglich. Eine Komponentensemantik kann folglich nur auf konzeptueller Ebene angewandt werden. D.h. dass sie lediglich in den Diagrammen repräsentiert ist, aber keine Entsprechung in der Implementierung findet.

3.2 Architektur einer MOM

Die folgenden Ausführungen basieren auf den in [2] dargestellten Konzepten einer allgemeinen MOM-Architektur.

Eine MOM stellt eine Kommunikationsinfrastruktur für verteilte Applikationen bereit. Die Kommunikation innerhalb einer MOM basiert auf *Nachrichtenwarteschlangen (Messagequeues)*, die von ihr zur Verfügung gestellt und verwaltet werden. Die miteinander kommunizierenden Komponenten einer verteilten Applikation stellen Nachrichten nur in Nachrichtenwarteschlangen, die lokal für sie sind, d.h. in Warteschlangen, die sich auf der selben Maschine nicht jedoch weiter entfernt als im selben lokalen Netzwerk (LAN) befinden. Diese werden als Quellwarteschlangen bezeichnet. Analog dazu werden Nachrichten nur aus lokalen Zielwarteschlangen gelesen. Aufgabe der MOM ist es, die Nachrichten von einer Quell- in eine Zielwarteschlange zu kopieren. Die Nachrichtenwarteschlangen sind über mehrere Maschinen verteilt. Die Verwaltung der jeweiligen Warteschlange übernimmt dabei ein *Warteschlangenmanager*. Dieser kommuniziert direkt mit der Komponente, die Nachrichten empfängt bzw. sendet.

Abb. 3.3 zeigt eine schematische Darstellung der in dieser Arbeit umgesetzten MOM-Architektur. Nutzeranfragen nimmt das *Gateway* entgegen und speist sie unter Benutzung eines Warteschlangenmanagers *MQManager* in die für sich lokale Warteschlange *GatewayQueue* ein. Die eingestellten Nachrichten werden von der MOM in die Zielwarteschlange *JobServerQueue* übertragen. Jeder *JobServer* verfügt ebenfalls über einen Warteschlangenmanager, der die Nachrichten aus der JobServerQueue entnimmt und sie dem entsprechenden JobServer aushändigt. Ein JobServer verarbeitet diese Nachricht und fordert nach Beendigung der Verarbeitung seinen Warteschlangenmanager dazu auf, die Antwortnachricht in die JobServerQueue einzustellen. Von dort aus wird sie zur GatewayQueue weitergeleitet.

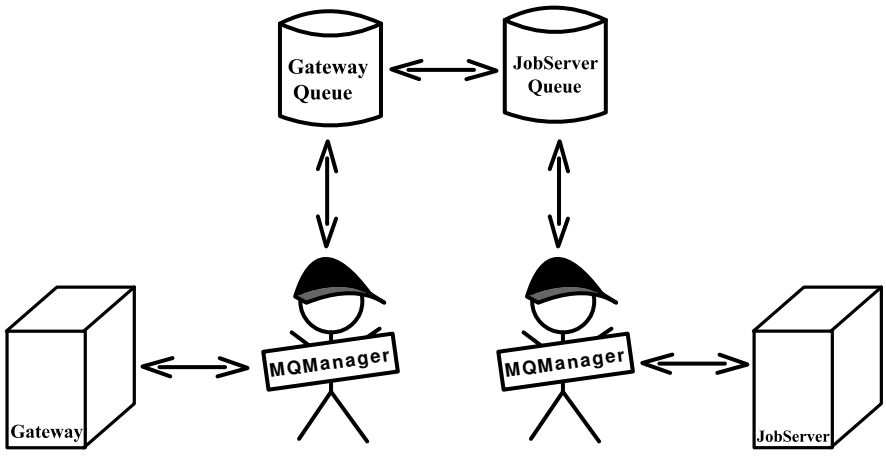

Abbildung 3.3: Schematische Darstellung der MOM-Architektur

3.3 Anforderungen an ein Verteiltes System

Ein die in Abschnitt 2.1 aufgeführten Arten von Transparenz erfüllendes verteiltes System muss bestimmte Anforderungen bzgl. der nebenläufigen Ausführung von Prozessen, Skalierbarkeit und Mehrbenutzersynchronisation umsetzen. Die der Umsetzung zu Grunde liegenden Verfahren werden im folgenden beschrieben.

3.3.1 Nebenläufigkeit

Ziel der Nebenläufigkeit ist eine höhere Ausführungsgeschwindigkeit durch die Aufteilung eines Problems in Teilprobleme, die von mehreren Maschinen gleichzeitig (parallel) bearbeitet werden. Die für die Erstellung der Laufzeitumgebung relevanten Typen von Parallelität beschreibt [28] folgendermaßen:

Netzwerkprozessoren und verteiltes Rechnen Hierbei sind mehrere Maschinen lose über ein Datenaustauschsystem wie z.B. ein Local Area Network (LAN) gekoppelt. Im Gegensatz zur direkten Kopplung besteht bei einer losen Kopplung zwischen den Maschinen keine feste physikalische Verbindung. Des weiteren verfügt jede Maschine über ihren eigenen lokalen Speicher und zwischen den verschiedenen Maschinen gibt es keine gemeinsam genutzte Ressource. Die Maschinen entziehen sich einer zentralen Kontrolle und sind somit autonom.
Die Benutzung solch autonomer Maschinen ermöglicht die Ausprägung des Merkmals der *Aufteilung* eines verteilten Systems (siehe Abschnitt 2.1). Auf TIRA bezogen, kann die Ausführung eines IR-Prozesses auf mehrere autonome Maschine verteilt werden, um die Ausführungsgeschwindigkeit zu steigern.

Multiprogramming (Mehrbenutzerbetrieb) Hierunter verteht man die gleichzeitige (nebenläufige, parallele) Ausführung mehrerer Programme auf einer Maschine mit einer einzigen CPU (Pseudo-Parallelität). Ziel ist die höhere Auslastung der Maschine, indem die Wartezeit, also die Zeit, in der die CPU untätig ist, minimiert wird. Diese Wartezeiten entstehen, wenn Programme auf langsamere Betriebsmittel wie z.B. eine Festplatte oder ein Netzwerk zugreifen. Beim Multiprogramming werden mehrere Programme so miteinander verzahnt, dass bei einer Untätigkeit der CPU bei der Ausführung eines Programms ein anderes Programm ausgeführt wird. Im Gegensatz zum Multiprocessing, bei dem zwei oder mehrere CPUs an der Ausführung beteiligt sind, handelt es sich um ein Softwarekonzept.
Ein JobServer wird unter Verwendung des Multiprogramming in die Lage versetzt, mehrere IR-Module gleichzeitig auszuführen. Die zu minimierenden Wartezeiten bei der Ausführung eines IR-Moduls entstehen beim Laden der Module und beim Laden bzw. Schreiben der Ein- und Ausgabedaten. Während die Ausführung eines IR-Moduls aufgrund von Lade- bzw. Schreibaktivitäten blockiert, kann mit der Ausführung eines weiteren IR-Moduls fortgefahren werden.

3.3.2 Skalierbarkeit und Erweiterbarkeit

Bei Lastzuwächsen soll die Leistungsfähigkeit des Systems ohne hohen Aufwand gesteigert werden können. Die Steigerung der Leistungsfähigkeit kann durch Hinzufügen zusätzlicher Maschinen, die die Ausführung der IR-Module übernehmen, erfolgen. Ziel ist solch eine Erweiterung des Systems ohne jedoch andere Teile des Systems anpassen bzw. anhalten zu müssen. Das System soll zur Laufzeit, also *dynamisch*, erweitert werden können. Außerdem müssen die Maschinen in der Lage sein, jedes der an der Ausführung eines IR-Prozesses beteiligten IR-Module auszuführen, ohne vorab über die von ihnen auszuführenden IR-Module bescheid zu wissen. Die dynamische Erweiterbarkeit wird durch Benutzung der Technik der *Code-Migration* ermöglicht. In [2] wird zwischen folgenden Modellen für Code-Migration unterschieden:

Schwache Mobilität Es werden nur das auszuführende Codesegment und, falls notwendig, Initialisierungsdaten übertragen. Ein Merkmal der schwachen Mobilität ist der Start des Programms vom Ausgangsstatus aus. Zusätzlich wird unterschieden, ob der migrierte Code vom Zielprozess oder von einem separaten, neu gestarteten Prozess ausgeführt wird. Wird er vom Zielprozess ausgeführt, so muss dieser gegen böswillige oder versehentliche Codeausführung geschützt werden.

Starke Mobilität Hierbei wird zusätzlich zu Codesegment und Inititalisierungsdaten der aktuelle Ausführungsstatus (Ausführungssegment) übertragen.

3.3.3 Mehrbenutzersysnchronisation

Die Koordination der Ausführung der IR-Prozessinstanzen wird von einer zentralen Komponente, der Messagequeue, durchgeführt. Da die IR-Module parallel ausgeführt werden und somit gleichzeitig auf die Messagequeue zugegriffen wird, kann es bei unkontrolliertem Zugriff zu Fehlern kommen, die die Konsistenz der zur Verwaltung der Ausführung notwendigen Daten gefährden können. Die Mechanismen zur Kontrolle gleichzeitiger Zugriffe werden unter dem Begriff der *Mehrbenutzersynchronisation* zusammengefasst.

Zur Vermeidung von Fehlern dieser Art muss der parallele Zugriff auf die Daten koordiniert bzw. kontrolliert werden. Zum Verständnis der Kontrollkonzepte soll zunächst der Begriff der *Transaktion* eingeführt werden.

„ Unter einer Transaktion versteht man die 'Bündelung' mehrerer Datenbankoperationen, die in einem Mehrbenutzersystem ohne unerwünschte Einflüsse durch andere Transaktionen als Einheit fehlerfrei ausgeführt werden sollen. “ [1]

Die gewünschten Eigenschaften einer Transaktion fasst das *ACID*-Akronym[1] zusammen:

Atomicity fordert die Behandlung einer Transaktion als atomare Einheit. Sie wird entweder komplett oder gar nicht ausgeführt.

Consistency besagt, dass eine Transaktion einen konsistenten Zustand der Datenbasis hinterlassen muss. Die während der atomaren Ausführung eintretenden Zwischenzustände können inkonsistent sein.

Isolation Dass sich gleichzeitig (parallel) ausgeführte Transaktionen gegenseitig nicht beeinflussen, fordert die Isolation-Eigenschaft. Eine Transaktion darf die Änderungen parallel laufender Transaktionen nicht sehen.

Durability verlangt die dauerhafte Speicherung der Auswirkungen einer erfolgreich beendeten Transaktion auch nach Systemfehlern. Das endgültige Speichern wird auch als Commit bezeichnet bzw. mit einem Commit-Befehl bestätigt.

Die Isolation-Eigenschaft steht mit der Forderung nach einer maximalen Anzahl gleichzeitig ablaufender Transaktionen im Widerspruch. Um die Änderungen einer sich in Ausführung befindlichen Transaktion T_1 vor einer weiteren sich in Ausführung befindlichen Transaktion T_2 zu verbergen, müssen die von T_1 geänderten Daten bis zu ihrem Ausführungsende gesperrt werden, so dass T_2 auf diese nicht zugreifen kann. Benötigt T_2 Zugriff auf diese gesperrten Daten, hat dies zur Folge, dass T_2 auf das Ausführungsende von T_1 warten muss und beide Transaktionen nicht parallel ablaufen können.

Oftmals muss ein Kompromiss zwischen der Maximierung der Isolation und dem Durchsatz von Transaktionen gefunden werden. Um den Grad der Übereinstimmung einer Transaktion mit der Isolation-Eigenschaft zu bestimmen, definiert der ANSI-SQL-92-Standard [5] vier Isolations-Ebenen. Sie geben Auskunft über die Wechselwirkung zwischen Datenänderungen in verschiedenen Transaktionen [10]. In [29] werden diese Ebenen wie folgt beschrieben:

Read Uncomitted ("dirty read") Dies ist die unterste Ebene. Hierbei kann eine Transaktion noch nicht mit Commit bestätigte Daten anderer Transaktionen lesen. Sie bietet den größten Durchsatz von Transaktionen, steigert die Wahrscheinlichkeit des Auftretens von Inkonsistenzen aber erheblich.

Read Committed Auf dieser Ebene kann eine Transaktion nur mit Commit bestätigte Daten anderer Transaktionen lesen. Sie ist der Regelfall.

Repeatable Read (RR) Eine Transaktion sieht lediglich die Änderungen, die vor ihrem eingenen Beginn bestätigt worden sind. Alle Leseoperationen innerhalb einer RR-Transaktion liefern dasselbe Ergebnis. RR geschieht auf Kosten der Performance, bietet aber eine konsistente Sicht der Daten.

Serializable ("full isolation") Diese Ebene ist RR erweitert um die Verhinderung des Phantom-Problems: Alle Suchoperationen innerhalb einer voll isolierten Transaktion T_1 finden stets

[1] ACID steht für *A*tomicity, *C*onsistency, *I*solation und *D*urability

diesselben Daten selbst dann, wenn eine andere Transaktion T_2 zusätzliche Daten zwischen zwei von T_1 durchgeführten Suchoperationen hinzugefügt hat.

Die Auswahl einer Isolations-Ebene muss je nach Anwendung abgeschätzt und mit der gewünschten Performance (Durchsatz von Transaktionen) aufgewogen werden. Bei der Verwaltung der zur Koordination der Ausführung eines IR-Prozesses notwendigen Daten steht die Konsistenz dieser Daten im Vordergrund. Aus diesem Grund sollte mindestens die Isolations-Ebene *Repeatable Read* verwendet werden.

Die ACID-Eigenschaften einer Transaktion können nur umgesetzt werden, wenn simultane Änderungen an der Datenbasis und somit Konflikte zwischen Transaktionen unterbunden werden. Aus diesem Grund kommt der *Isolation*-Eigenschaft eine besondere Bedeutung zu. Sie wird um das Konzept der *Sperr-* bzw. *Synchronisationsstrategien* erweitert. Diese beschreiben, wie simultan laufende Transaktionen synchronisiert werden. Es gibt zwei Sperr- bzw. Synchronisationsstrategien [1]:

optimistisch Bei der optimistischen Strategie wird davon ausgegangen, dass Konflikte selten auftreten. Sie kennt keine Sperren und etwaige Konflikte werden am Ende einer Transaktion auf Basis protokollierter Beobachtungen aufgelöst. Trat ein Konflikt auf, wird eine Transaktion zurückgesetzt. Diese Art der Synchronisation eignet sich für Anwendungen mit einer großen Anzahl lesender Transaktionen.

pessimistisch Die pessimistische Strategie geht à priori von auftretenden Konflikten aus. Vom Programmierer wird verlangt, dass er entsprechende Vorkehrungen trifft, um diese potentiellen Konflikte zu vermeiden. Die Entscheidung für die Anwendung dieser Strategie sollte getroffen werden, wenn die zu erstellende Anwendung aus einer großen Anzahl schreibender Transaktionen besteht, die sich gegenseitig *stören* könnten.

Eine in [1] beschriebene pessimistische Strategie ist die *sperrbasierte Synchronisation*. Hierbei darf eine Transaktion erst nach Erhalt einer entsprechenden Sperre auf die von ihr benötigten Daten zugreifen. Es wird zwischen zwei Sperrmodi unterschieden:

S (shared, read lock, Lesesperre) Wenn eine Transaktion eine S-Sperre für ein Datum A besitzt, kann sie dieses lesen, ohne dass Änderungen anderer Transaktionen auf diesem Datum zugelassen werden. Mehrere Transaktionen können gleichzeitig eine S-Sperre auf demselben Datum besitzen.

X (exclusive, write lock, Schreibsperre) Nur die Transaktion, welche eine X-Sperre für ein Datum besitzt, darf schreibend darauf zugreifen. Eine Schreibsperre kann zu einem Zeitpunkt von nur einer Transaktion gehalten werden.

Die sperrbasierte Synchronisation bringt jedoch das Problem von *Verklemmnungen (Deadlocks)* mit sich. Zwei Transaktionen T_1 und T_2 sind verklemmt, wenn T_1 auf die Freigabe einer Sperre durch T_2 wartet und umgekehrt T_2 auf die Freigabe einer Sperre durch T_1 wartet. Beide Transaktionen sind blockiert und können in ihrer Ausführung nicht fortfahren.

Um nun das System nicht zum Halten zu bringen, sind Methoden zur Erkennung und Auflösung bzw. Vermeidung von Deadlocks notwendig. Laut [1] können Verklemmungen auf zwei Arten erkannt und aufgelöst werden:

Time-out-(“brute force”) Bei dieser Strategie wird die Ausführung einer Transaktionen überwacht. Erzielt eine Transaktion innerhalb eines Zeitraumes keinen Fortschritt, wird sie zurückgesetzt und die von ihr gehaltenen Sperren werden freigegeben.

Wartegraph-basiert Hierbei werden die Kennungen der aktiven Transaktionen als Knoten in einem gerichteten Graph modelliert. Eine Kante vom Knoten T_i zum Knoten T_j wird dann eingefügt, wenn die entsprechende Transaktion T_i auf die Freigabe einer Sperre durch die entsprechende Transaktion T_j wartet. Eine Verklemmung liegt nur dann vor, wenn der Wartegraph einen Zyklus aufweist.

Obwohl die Anwendung der pessimistischen Strategie das Auftreten von Deadlocks erhöht, sollte sie bei der Implementierung der Messagequeue, dennoch angewendet werden. Diese Entscheidung basiert auf der Annahme, dass die Veränderung der zur Koordination der Ausführung der IR-Prozesse notwendigen Daten durch die JobServer und das Gateway sehr häufig erfolgt und somit mit einer großen Anzahl von Konflikten zu rechnen ist.

Die verlässliche Umsetzung der Anforderungen der Mehrbenutzersynchronisation nimmt einen sehr hohen Stellenwert ein, da ohne eine korrekt funktionierende Koordiantion der Ausführung der IR-Prozessinstanzen die gesamte Middleware-Plattform ihren Zweck nicht erfüllen kann. Aus diesem Grund scheint es sinnvoll auf eine diese Anforderungen bereits umsetzende Plattform zurückzugreifen. Hierfür bieten sich *Datenbankmanagementsysteme (DBMS)* an. DBMS adressieren nicht nur das Problem der Mehrbenutzersynchronisation. Sie bieten auch Lösungen für typische Probleme der Informationsverarbeitung [1]:

- Redundanz und Inkonsistenz
- Verlust von Daten
- Integritätsverletzungen
- beschränkte Möglichkeit der Verknüpfung von Daten
- Zugriffsbeschränkung

Aus diesem Grund wird die koordinierende Komponente auf Basis eines (relationalen) DBMS umgesetzt.

3.4 Simulation von UML Aktivitätsdiagrammen

In diesem Abschnitt sollen die der Koordination der Ausführung der IR-Prozessinstanzen zu Grunde liegenden Konzepte erläutert werden. Die folgenden Ausführungen orientieren sich an den in [27] dargestellten Konzepten zu formalen Ausführungssemantiken von UML-Aktivitätsdiagrammen.

Die koordinierende Komponente repräsentiert eine IR-Prozessinstanz speicherintern als einen Datenflussgraph (DFG).

Datenflussgraph Ein Datenflussgraph $DFG = (V, E)$ ist ein gerichteter Graph, der keine Zyklen aufweist. Jeder Knoten $v \epsilon V$ repräsentiert eine Operation. Jede Kante $e \epsilon E$ stellt einen Datenfluss zwischen ihren inzidenten Knoten dar und legt somit die Ausführungsabhängigkeiten fest: Eine Operation $v \epsilon V$ darf nur dann ausgeführt werden, falls alle Operationen $u \epsilon V$ mit $(u, v) \epsilon E$ bereits ausgeführt worden sind. Das UML-Klassenmodell des verwendeten DFG zeigt Abb. 3.4.

Die Regeln zur Koordination der Ausführungsabhängigkeiten werden auch als *Ausführungssemantik* bezeichnet. Um die Ausführungsabhängigkeiten zwischen den einzelnen Knoten zu verwalten, wird eine Petri-Netz-artige Markensemantik verwendet und jedem Knoten ein Zustand zugewiesen. Die Zustände und deren Übergänge sind in Anlehnung an die Zustände des Prozessmodells moderner Betriebssysteme modelliert. Das in [33] beschriebene Zustandsmodell zeigt Abb. 3.5.

Die Regeln zur Koordination der Ausführung der von den Knoten repräsentierten IR-Module sieht jedoch keinen Übergang vom Zustand *rechnend* in den Zustand *blockiert* bzw. in den Zustand *wartend* vor. Die IR-Module werden ohne Unterbrechung ausgeführt. Diese Aussage gilt auf konzeptueller Ebene. Bei der tatsächlichen Ausführung kann es natürlich dazu kommen, dass der das Modul ausführende Prozess blockiert. Abb.3.6 zeigt das zur Koordination verwendete Zustandsmodell. Wird ein Knoten erzeugt, geht er in den Zustand *blockiert* über. Er verweilt solange in diesem Zustand, bis alle gemäß der Definition eines DFG verlangten *Vorgänger*-Operationen ausgeführt worden sind. Ist dies der Fall, wird der entsprechende Knoten als *wartend* markiert. Eine Ausnahme bilden Instanzen der Klasse *StartNode*. Sie werden unmittelbar nach ihrer Erzeugung als wartend markiert. Ein sich in Ausführung befindlicher Knoten wird als *rechnend* markiert. Ist die Ausführung beendet, wird er als *terminiert* markiert.

Das UML Aktivitätsmodell sieht zwei Arten von Marken (Token) vor: Daten- und Kontrolltoken. Ein Kontrolltoken dient als eine Art Ausführungserlaubnis für einen Knoten. Trifft eine bestimmte Anzahl von Kontrolltoken in einem Knoten ein, kann dieser ausgeführt werden. Ein Datentoken transprotiert ein Datum oder eine Referenz auf ein Datum. Auch Datentoken können als Ausführungserlaubnis fungieren.
Die in dieser Arbeit umgesetzte Ausführungssemantik verwendet lediglich Kontrolltoken. Da jeder Knoten in einem DFG per Definiton über Ein- und Ausgabedaten verfügt, werden Datentoken nicht modelliert. Folglich wird auch nicht zwischen Kontroll- und Datenflusskanten unterschieden. Wie oben beschrieben ist ein Knoten ausführbar, wurden alle seine *Vorgänger*-Knoten ausgeführt. Die Anzahl der Vorgänger-Knoten wird durch das Attribut *requiredTokens* der Klasse *TokenBuffer* modelliert. Die Anzahl terminierter Vorgänger-Knoten repräsentiert der Wert des Attributes *arri-*

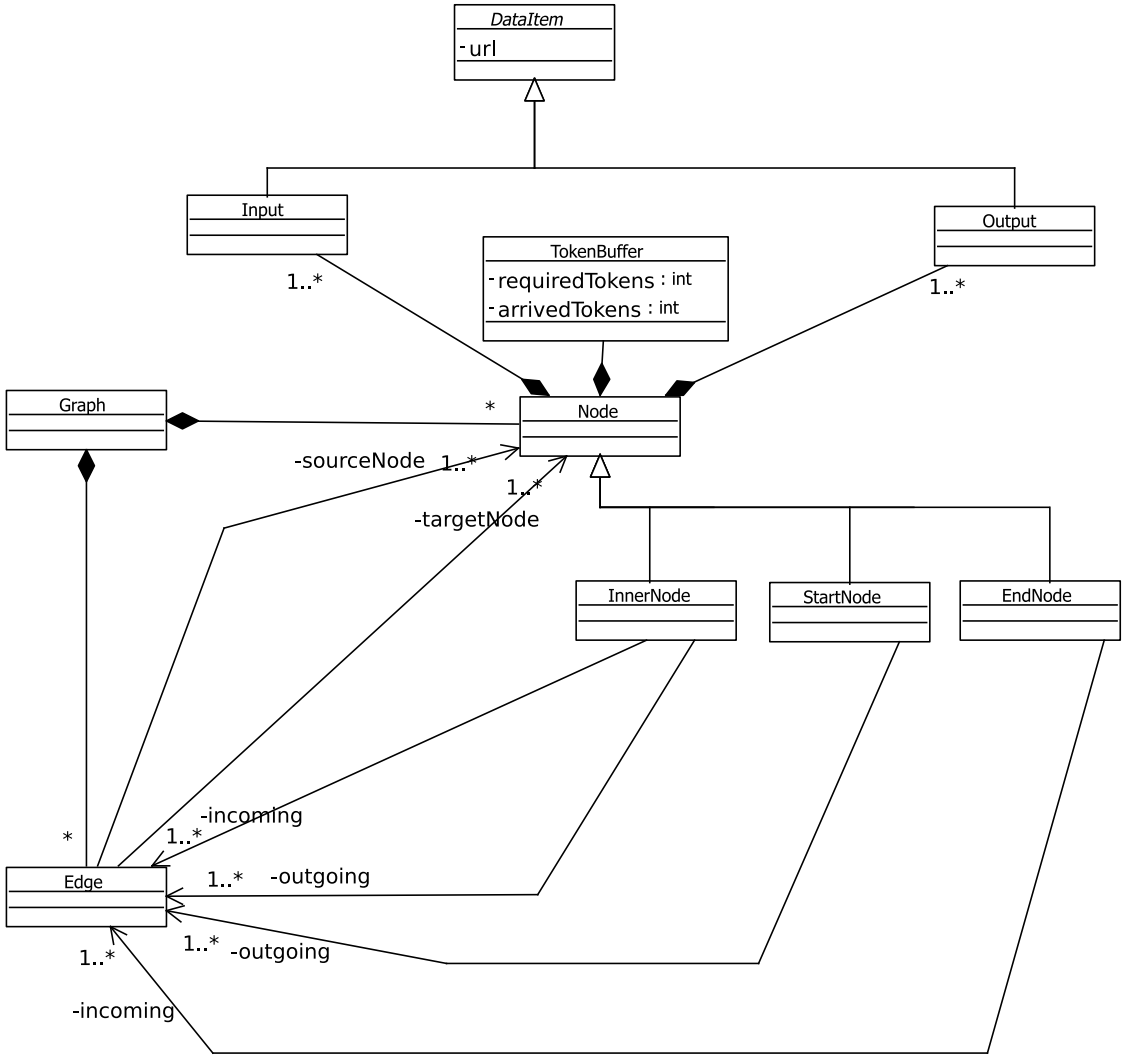

Abbildung 3.4: Klassendiagramm des verwendeten DFG

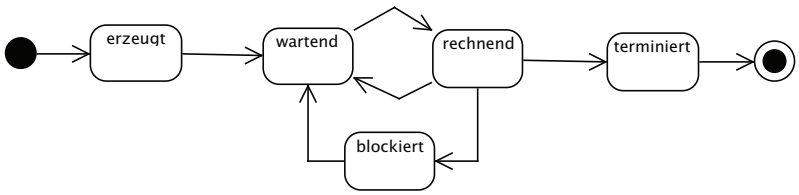

Abbildung 3.5: Zustandsmodell von Prozessen moderner Betriebssysteme [33]

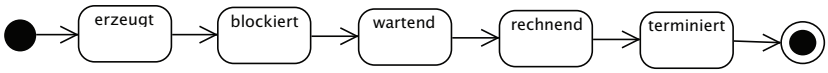

Abbildung 3.6: Zustandsmodell zur Verwaltung der Ausführungsabhängigkeiten.

vedTokens. Wird ein Vorgänger-Knoten als *terminiert* markiert, so wird in allen adjazenten Knoten dieses Attribut um eins erhöht. Anschließend wird überprüft, bei welchen Knoten der Wert des Attributes *arrivedTokens* gleich dem Wert des Attributes *requiredTokens* ist. Ist dies der Fall, wird der entsprechende Knoten als *wartend* markiert - die Ausführungserlaubnis wird erteilt.
Wird in einem Aktivitätsdiagramm ein Paralleliserungsknoten (PK) verwendet, soll damit ausgedrückt werden, dass beim Eintreffen eines Tokens die nachfolgenden Aktionen bzw. Aktivitäten parallel ausgeführt werden können. Auf Abb. 3.7 bezogen, können die Aktionen *a2* und *a3* nach dem Ausführungsende von *a1* parallel ausgeführt werden. In dieser Abbildung ist auch zu sehen, wie ein PK in einen DFG transformiert wird. Die eingehende Kante des PK wird jeweils mit einer ausgehenden Kante zusammengeführt. So entsteht die Kante *e7* durch Zusammenführen der Kanten *e1* und *e2*. Die Verwendung eines Synchronisierungsknotens (SK) soll ausdrücken, dass auf das Ausführungsende aller Aktionen gewartet wird, die dem SK vorgelagert sind. Aktion *a6* aus Abb. 3.8 kann also erst ausgeführt werden, wenn die beiden Aktionen *a4* und *a5* beendet wurden. Bei der Transformation in einen DFG werden jeweils die eingehenden Kanten mit der ausgehenden zusammengeführt.

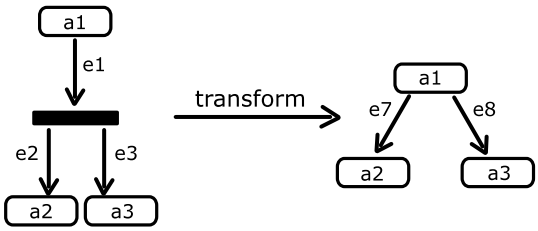

Abbildung 3.7: Transformation eines UML Parallelisierungsknoten in einen DFG

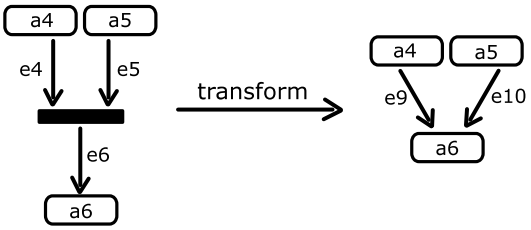

Abbildung 3.8: Transformation eines UML Synchronisierungsknoten in einen DFG

3.5 Modelltransformation

Wie einleitend erwähnt, soll aus einem Platform Independent Model (PIM) ein Platform Specific Model (PSM) generiert werden. Die Struktur des modellierten Ablaufgraphs, der dem mit der TIRA-GUI erstellten Aktivitätsdiagramm zu Grunde liegt, orientiert sich am Metamodell der UML [21]. Eine IR-Prozessintanz wird von der MQ speicherintern als ein Datenflussgraph (DFG) repräsentiert. Folglich muss der Ablaufgraph der TIRA-GUI, das PIM, in einen DFG, das PSM, transformiert werden, damit die Middleware-Plattform die Spezifikation eines IR-Prozesses auswerten kann. PIM und PSM sind MDA-spezifische Konzepte. Aus Sicht des Autors lassen sich die Transformationskonzepte in allgemeinerer Form, also mit den MDSD-Konzepten, verständlicher

darstellen als mit den Konzepten der MDA. Zur Einordnung der MDA-Konzepte PIM und PSM in den MDSD-Konzeptraum sei zunächst noch einmal auf Abb. 2.9 verwiesen.
Ziel der modellgetriebene Softwareentwicklung ist es u.a. aus Modellen automatisch ein lauffähiges System zu erstellen. Um dies zu bewerkstelligen ist die automatische Ausführung einer Reihe von Arbeitsschritten in einem *Build-Prozess* notwendig. Abb. 3.9 zeigt diese Arbeitsschritte (Workflow). Als Eingabe dienen formale Modelle, die auf Basis eines Metamodells erstellt wurden. Zunächst müssen diese eingelesen werden. Die entsprechenden Komponenten heissen *Parser* oder *Instanziatoren*. Nach diesem Schritt liegen die Modelle als Objektgraphen im Speicher vor. Der nächste Schritt beinhaltet die *Validierung* der Modelle. Es wird geprüft, ob die formalen Modelle die Regeln des Metamodells, auf dessen Basis sie erstellt wurden, befolgen. Dies dient der Erkennung von Modellierungsfehlern.

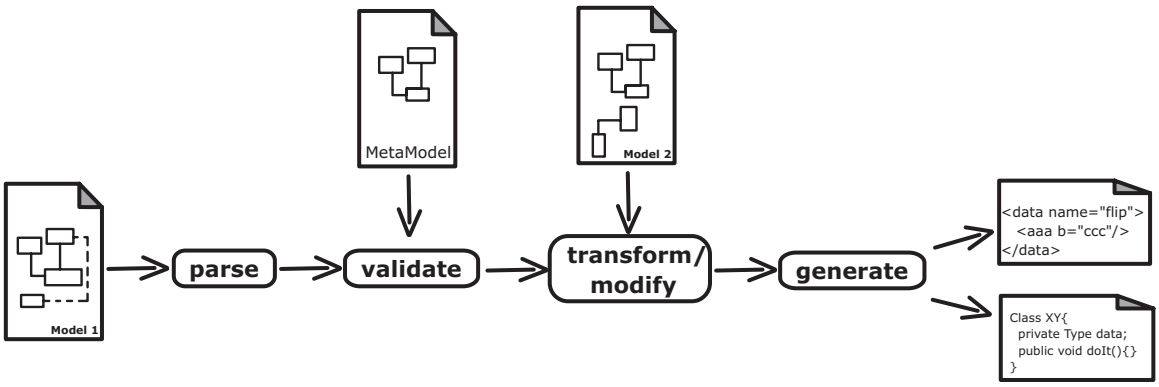

Abbildung 3.9: Workflow zur automatischen Erstellung von (Software-)Artefakten aus formalen Modellen

Der folgende Schritt ist optional. In diesem Schritt werden die Modelle so aufbereitet, wie sie die Generierungskomponente erwartet. Z.B. kann ein eingelesenes Modell um zusätzliche Informationen ergänzt werden (*Modellmodifikation*). Eine *Modelltransformation* erzeugt ausgehend von mehreren Modellen ein zusätzliches Modell, dass auf einem völlig anderen Metamodell basiert. Zu beachten ist, dass die dabei generierten Modelle keiner manuellen Änderung unterworfen sein sollten, bevor Code aus ihnen generiert wird, da der gesamte Build-Prozess sonst nicht mehr automatisch ausgeführt werden kann. Der letzte Schritt besteht nun aus der Generierung der Artefakte (z.B. Sourcecode), die notwendig sind, um das lauffähige System zu erstellen. In [17] werden verschiedene Generierungstechniken beschrieben. Die für diese Arbeit relevante Technik wird als *Templates und Filtering* bezeichnet. Dabei wird mit Hilfe von Templates über die relevanten Teile eines textuell repräsentierten Modells iteriert. In den Templates sind die Generierungsvorschriften enthalten. Sie bestimmen wie welches Modellelement auf die MDSD-Plattform abgebildet wird.
Bezüglich der Generierung ist anzumerken, dass selten der gesamte Quellcode, der sonst per Hand geschrieben würde, automatisch erstellt wird. Zur Erläuterung sei auf den Begriff der MDSD-Plattform verwiesen. Sie stellt eine Art Laufzeitumgebung für das Modell dar. Diese Laufzeitumgebung kann bereits vollständig sein und muss lediglich konfiguriert werden, oder die für ihre Vollständigkeit relevanten Artefakte werden aus dem Modell generiert und ggf. manuell ergänzt. Diese manuelle Ergänzung betrifft nicht die im *transform/modify*-Schritt erzeugten Modelle, sondern die am Ende des Build-Prozesses generierten Artefakte.

Kapitel 4

Implementierung der MOM und der Modelltransformation

In diesem Kapitel sollen die Resultate der Implementierung der MOM und der Transformation des Platform Independent Model (PIM) in das Platform Specific Model (PSM) veranschaulicht werden. Nachdem kurz erläutert wird, wie Komponenten auf der Java-Plattform konfiguriert und komponiert werden, wird die A-, T- und TI-Architektur der MOM präsentiert. Anschließend wird gezeigt, wie die Anforderungen an die Code-Migration und Mehrbenutzersynchronisation umgesetzt wurden. Mit welchen Mitteln die Modelltransformation umgesetzt wurde, erklärt der darauf folgende Abschnitt. Den Abschluss bildet ein kurzes Beispiel, das veranschaulichen soll, wie zukünftige IR-Module auf Basis der MOM entwickelt werden sollen.

4.1 Konfiguration und Komposition der Komponenten

Wie in Abschnitt 2.2.4 erläutert, müssen die Komponenten miteinander verknüpft werden. Dies ist die Aufgabe des *Kompositionsmanagers*. Als Kompostitionsmanager für die Java-Plattform wurde das *Spring*-Framework [12] verwendet. Spring basiert auf dem Prinzip der *Dependency Injection* (DI). DI ist eine Technik zum Aufbau von Objektnetzen, bei der die benötigten Objekte zur Laufzeit *injiziert* werden. D.h. dass ein Objekt die von ihm benötigten Objekte nicht selbst erzeugt, sondern diese zugewiesen bekommt. Die zu injizierenden Objekte werden anhand der set-Methoden bzw. der Konstruktoren einer Klasse oder anhand von Konfigurationsdateien automatisch ermittelt und injiziert.

4.2 A- und T-Architektur

Die Anwendungsarchitektur der Middleware-Plattform manifestiert sich in Form der Komponente *MOM*. Sie ist eine Komposition aus den fünf in Abb. 4.1 gezeigten Komponenten. In den folgenden Abschnitten werden die durch diese Komponenten zur Verfügung gestellten Dienste und die

Interaktionen zwischen diesen beschrieben.
Dienst und *Komponente* bzw. *Interaktion(en) zwischen Diensten* und *Interaktion(en) zwischen Komponenten* werden im folgenden synonym verwendet. Wird von einem *Dienst X* gesprochen, so ist die Komponente gemeint, die den durch die *Schnittstelle X* beschriebenen Dienst implementiert. Da die von einer Komponente importierten Schnittstellen die einzige Annahme über ihre Umgebung ist, werden die Interaktionen zwischen den verschiedenen Komponenten auf Basis der Schnittstellen beschrieben. Dies hat zur Folge, dass eine Lebenslinie in einem UML-Sequenzdiagramm stets eine Schnittstelle und nicht deren Implementierung durch eine Komponente repräsentiert.

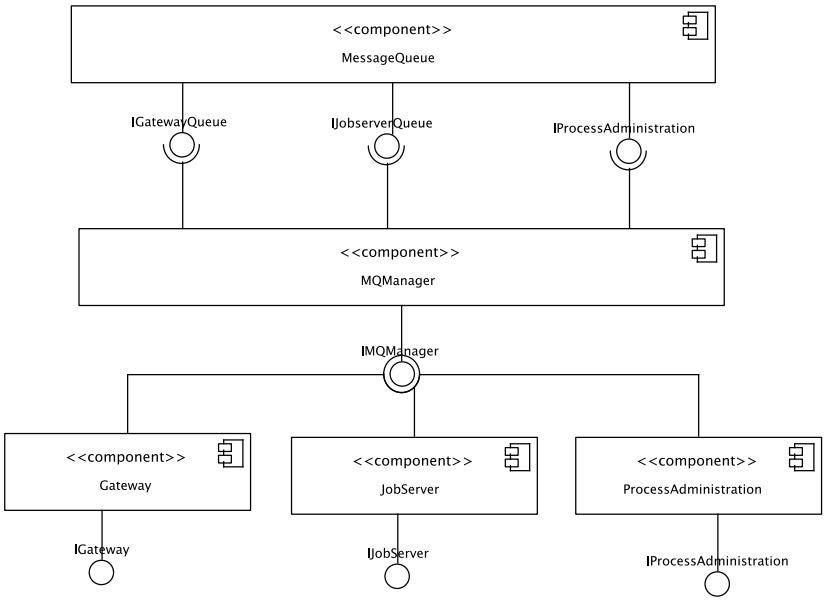

Abbildung 4.1: Die Anwendungsarchitektur der MOM

4.2.1 MessageQueue

Zentraler Baustein der MOM ist die Komponente *MessageQueue* (MQ). Sie empfängt, verarbeitet und speichert die an die MOM gestellte Anfragen. Wie in Abb. 4.2 zu sehen, bietet sie nach außen drei Dienste an:

1. Entgegennahme und Verarbeitung von Nutzeranfragen (IGatewayQueue)
2. Angebot von Jobs bzw. Entgegennahme ihrer Resultate (IJobServerQueue)
3. Verwaltung von IR-Prozessspezifikationen (IProcessAdministration)

Die Komponente MQ implementiert diese Dienste nicht selbst. Sie delegiert die Ausführung an ihre Subkomponenten *GatewayQueue*, *ProcessAdministration* und *ProcessCoordination*.

Das Datenmodell, auf dessen Basis die Komponenten *ProcessAdministration* und *ProcessCoordination* arbeiten, ist der in Abb. 3.4 modellierte DFG.

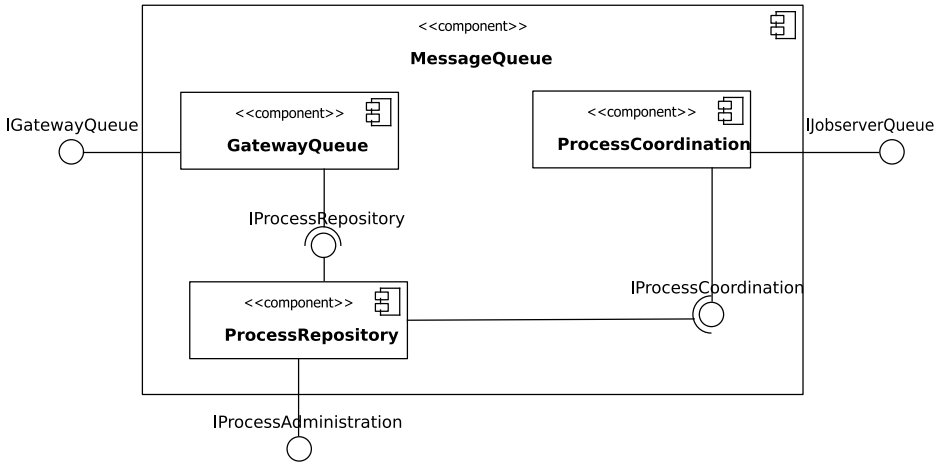

Abbildung 4.2: Die Komponente MessageQueue bestehend aus drei Subkomponenten.

4.2.1.1 ProcessCoordination

Die Komponente *ProcessCoordination* implementiert die in Abschnitt 3.4 beschriebe Logik zur Koordianation der Ausführung von IR-Prozessinstanzen. Bezogen auf die schematische Darstellung der Architektur der MOM in Abschnitt 3.2 nimmt sie die Rolle der *JobServerQueue* ein. Ist das Resultat des letzten Jobs eines Prozesses eingetroffen, so übermittelt sie die Nachricht über das Ausführungsende der Komponente *Gateway*.

4.2.1.2 ProcessAdministration

Die Schnittstelle *IProcessAdministration* beschreibt die administrativen Dienste der MOM. Sie dienen der Einspeisung und Löschung, allgemein der Verwaltung, von *Prozessspezifikationen*. Die vom Anwender spezifizierten IR-Prozesse werden nach einer zuvor stattfindenden Transformation mit Hilfe der von der Komponente *ProcessAdministration* implementierten Dienste in der MOM, genauer im *ProcessRepository*, gespeichert. Schematisch wird dieser Vorgang in Abb. 4.3 dargestellt. Ein Nutzer erstellt zunächst die Spezifikation eines IR-Prozesses mit Hilfe der TIRA-GUI. Anschließend wird diese in eine für die MOM verständliche Form transformiert (siehe Abschnitt 4.6), bevor sie schließlich im ProcessRepository hinterlegt wird.

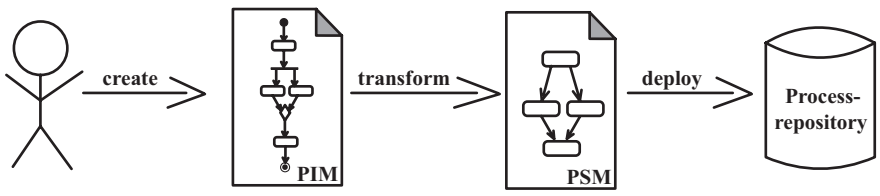

Abbildung 4.3: Ablauf zur Einspeisung einer Prozessspezifikationen in das ProcessRepository

4.2.1.3 GatewayQueue

Die Komponente *GatewayQueue* hat nicht viele Aufgaben zu erfüllen. Sie muss die Nutzeranfragen speichern und die Komponente *ProcessRepository* über neu eingetroffene Anfragen unterrichten. Die Komponente *ProcessCoordination* benutzt ihren Dienst, um Resultate zu hinterlegen.

4.2.2 JobServer

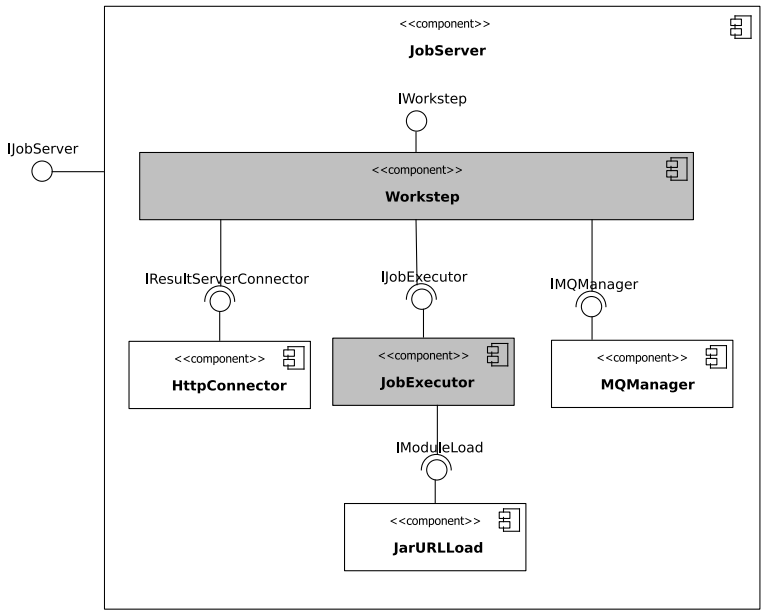

Abbildung 4.4: Die Komponente JobServer und ihre Subkomponenten. Die grau hinterlegten Komponenten sind A-Komponenten, die weiss hinterlegten sind T-Komponenten.

Die Komponente *JobServer* implementiert die Logik zum Laden bzw. zur Ausführung eines IR-Moduls. Das Datenmodell, auf dem sie operiert zeigt Abb. 4.5. Ein IR-Prozess ist eindeutig durch eine *sessionID* gekennzeichnet und besteht aus mehreren *Jobs*. Ein *Job* ist die Ausführung eins IR-Moduls. Das Attribut *moduleURL* dient der Identifizierung des auszuführenden Moduls. Die Klassen *Input* und *Result* repräsentieren sowohl Ein- und Ausgabedaten eines Jobs als auch eines gesamten Prozesses. Die Eingabedaten eines Jobs werden einer Instanz der Komponente *JobServer* nicht direkt (*per-value*) übermittelt sondern *per-reference* in Form einer URL. Das Attribut *key* der Klasse *DataItem* dient der Identifizierung der Daten innerhalb der Module (siehe Abschnitt 4.7).

Abb. 4.4 zeigt die innere Struktur der Komponente *JobServer*. Die Subkomponenten mit grauem Hintergrund sind A-Komponenten. Die Komponente *Workstep* (WS) übernimmt die Koordiantion der T-Komponenten *HTTPConnector, MQManager* und *JarURLLoad*. Das Sequenzdiagramm in Abb. 4.6 beschreibt diese Koordination. In einem ersten Schritt verbindet sich die Komponente WS mit Hilfe der Komponente MQManager (MQM) mit der Komponente ProcessCoordination (PC) (Abb. 4.2), um an die Daten eines auszuführenden Jobs zu gelangen. Anhand dieser Daten wird die Komponente HttpConnector aufgefordert, die benötigten Eingabedaten zu laden. In einem

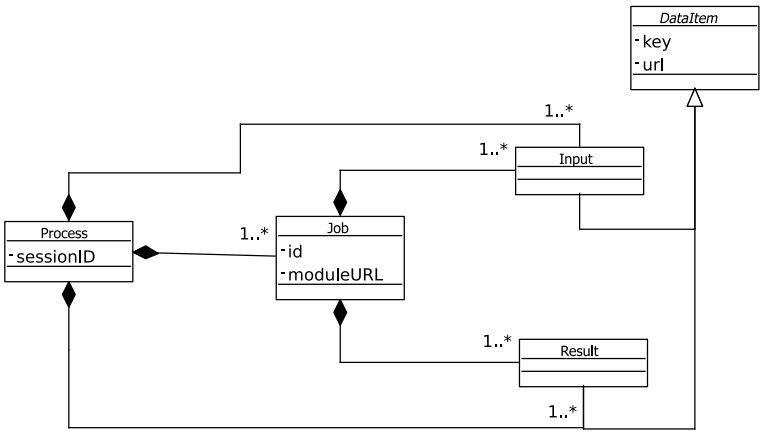

Abbildung 4.5: Das Datenmodell der Komponente JobServer.

nächsten Schritt wird die Ausführung eines IR-Moduls angestossen: Die Komponente JobExecutor veranlasst die Komponente JarURLLoad zum laden der Laufzeitinformationen, anhand derer das entsprechende Modul geladen und schließlich aufgerufen wird. Das Laden eines Moduls wird in Abschnitt 4.4 beschrieben.

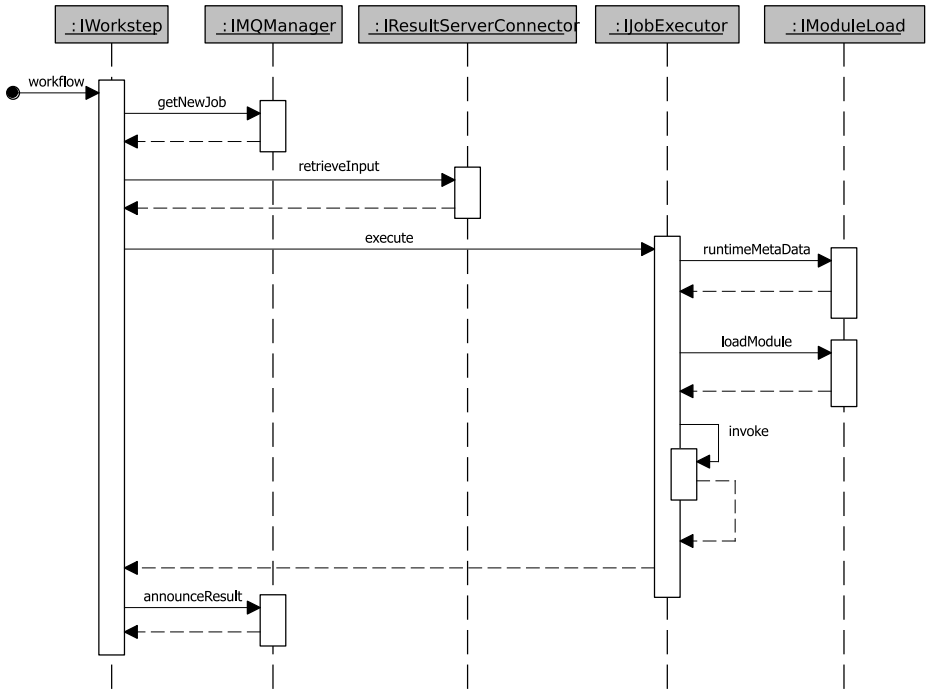

Abbildung 4.6: Der Dienst IWorkstep koordiniert die Dienste IResultServrConnector, IMQManager und IModuleLoad, um ein IR-Modul zu laden und aufzurufen.

4.2.3 Gateway

Der Dienst IGateway (GW) ist dem Nutzer am nächsten. Er beschreibt das Einspeisen der Nutzeranfragen in die MOM und das Abholen der Resultate der Ausführung der IR-Prozesse.

Das Verhalten der MOM bei Nutzung dieses Dienstes, also bei der Verarbeitung von Nutzeranfragen stellt Abb. 4.7 schematisch dar. Beispielhaft stellen drei Nutzer jeweils eine Anfrage. Zwei Nutzer fordern die Ausführung des Prozesses X und ein Nutzer die des Prozesses Y an. Zunächst werden die Anfragen (Requests) in die *GatewayQueue* eingespeist. Von hier aus wird das *ProcessRepository* veranlasst die entsprechenden Prozesse zu instantiieren. Instantiierung heisst, dass eine Arbeitskopie einer Prozessspezifikationen, die im ProcessRepository gespeichert ist, der *ProcessCoordination* übergeben wird. Hier wird die Abarbeitung, die Ausführung, der Prozessinstanzen koordiniert. Die Prozessinstanzen sind in Abb. 4.7 grau hinterlegt. Die Ausführung der Instanzen übernimmt ein oder mehrere JobServer auf Grundlage der IR-Module. Sind die Resultate der Ausführung verfügbar, wird dies der GatewayQueue von der ProcessCoordination mitgeteilt. Die Resultate können nun von GW abgeholt werden.

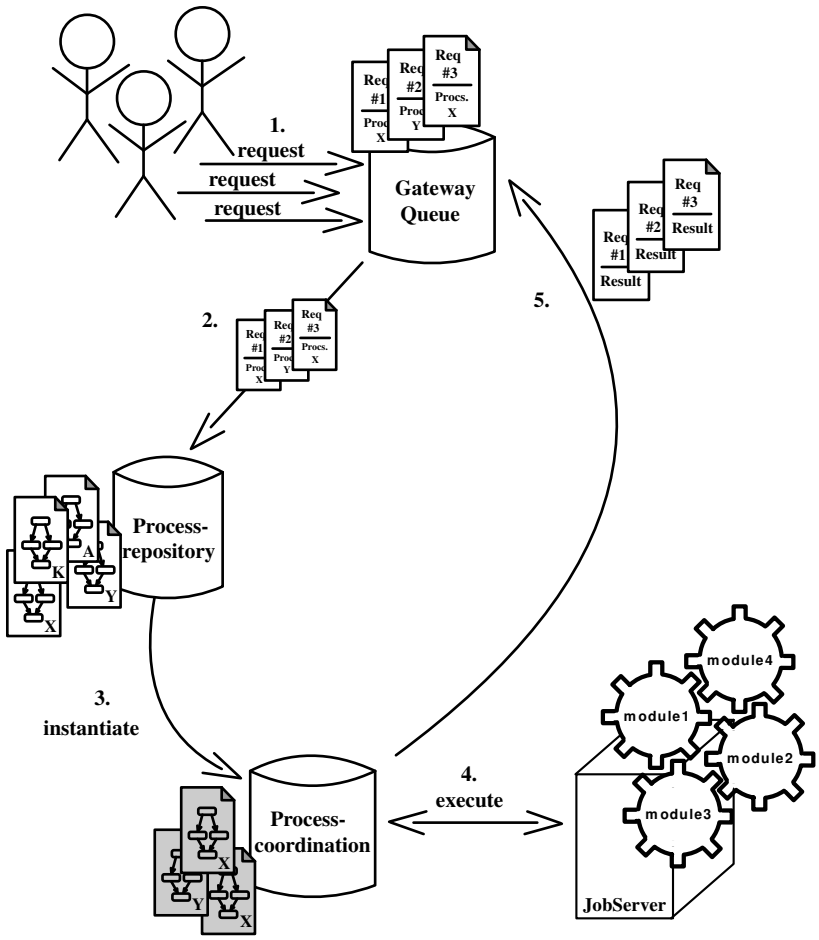

Abbildung 4.7: Schematische Darstellung der Vorgänge innerhalb der MOM beim Stellen von Nutzeranfragen

Die Interaktion zwischen den Diensten der MOM bei Eintreffen einer Nutzeranfrage veranschaulicht das Sequenzdiagramm in Abb. 4.8. Nachdem die Anfrage durch den Dienst IMQManager an die GatewayQueue übergeben wurde, ist der Dienst GW nicht weiter an die Ausführung gebunden. Dies ist durch die asynchrone Weiterleitung der Anfrage möglich. An diesem Punkt setzt die Entkopplung zukünftiger auf Basis der Middleware-Plattform erstellter Anwendungen und der Middleware-Plattform selbst ein. Nachdem eine Anwendung Anfragen in der GatewayQueue hinterlegt hat, kann sie zunächst unabhängig von der Middleware-Plattform ihre Ausführung fortsetzen. Nachdem der Abschluss der Instantiierung dem Dienst IProcessCoordination mitgeteilt wurde, stellt dieser den ersten abzuarbeitenden Job zur Verfügung.

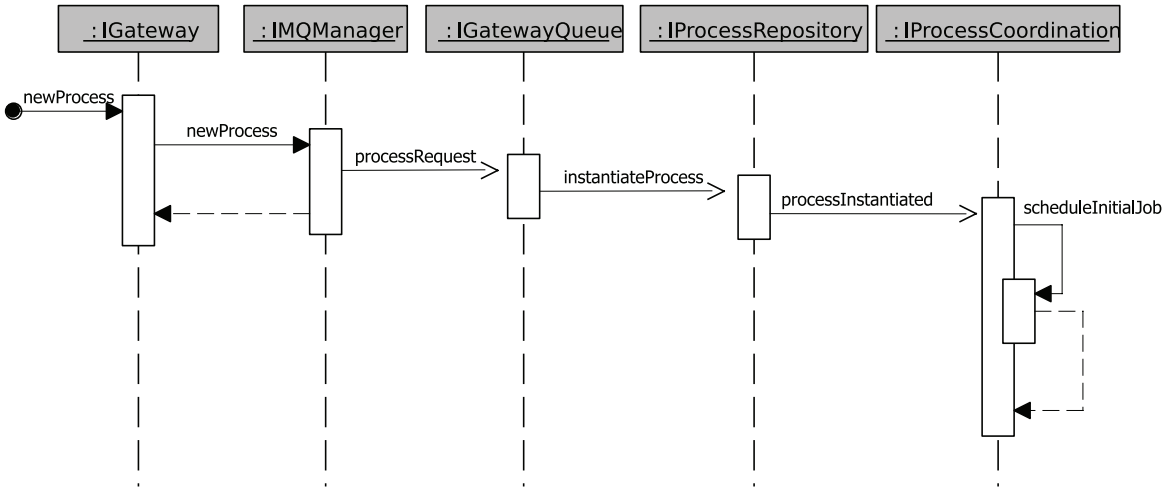

Abbildung 4.8: Interaktion der MOM-Dienste bei Eintreffen einer Nutzeranfrage. Gefüllte Pfeilspitzen stehen für synchrone, leere für asynchrone Aufrufe.

4.2.4 MQManager

Diese Komponente ist eine reine T-Komponente. Sie dient als Schnittstelle zwischen den in Java implementierten Komponenten Gateway, JobServer und Processadministration und der Komponente MessageQueue, die auf dem MySQL-Datenbank-Server (MDS) umgesetzt wurde. Um die Verbindung zum MDS aufzubauen, stützt sie sich auf das *iBatis* Framework. iBatis erleichtert die Nutzung eines Datenbankmanagementsystems (DBMS), indem es einerseits die Verwaltung der Verbindungen zum DBMS übernimmt und andereseits das Objekt-Relationale-Mapping (siehe unten) durchführt. Im Gegensatz zu anderen ORM-Frameworks erlaubt es den Aufruf von Stored Procedures (siehe Abschnitt 3.1.2). Die Abbildungsregeln werden in einer XML-Datei hinterlegt und können im Falle performancekritischer Anfragen mit selbst definierten SQL-Queries versehen werden.

4.2.4.1 Objekt-Relationales-Mapping (ORM)

ORM beschreibt die Abbildung (Mapping) hierarchischer Objektgraphen auf relationale Schemata. Der in Abb. 3.4 dargestellte DFG muss auf ein bzw. mehrere relationale Schemata abgebildet

werden, da die Komponente ProcessCoordination, deren Datenmodell der DFG ist, auf einem relationalen DBMS implementiert wurde.
Tabelle 4.1 zeigt nach welchem Verfahren Modellelemente der Objektorientierung (OO) auf relationale Schemata abgebildet werden. Die in der zweiten Spalte beschriebenen Abbildungsvorschriften beschreiben die Umsetzung eins Entity-Relationship-Modells (ERM) in relationale Schemata. Da sich die Modellelemente der OO auf die Modellelemente eines ERM abbilden lassen, können diese Abbildungsvorschriften beim ORM angewendet werden.

OO-Element	Abbildungsvorschrift
Klasse	Relationale Modellierung von Entitytypen nach [1]
Vererbung	Relationale Modellierung der Generalisierung nach [1]
Assoziation	Cross-Reference-Ansatz nach [26]
Aggregation	Relationale Modellierung schwacher Entitytypen nach [1]

Tabelle 4.1: Vorschriften zur Abbildung von OO-Elementen auf relationale Schemata

4.3 TI-Architektur

Abb. 4.9 zeigt die TI-Architektur Middleware-Plattform in Form eines UML Deploymentdiagramms. Der Übersichtlichkeit halber wurde auf die Darstellung der Verteilung der Komponenten auf die einzelnen Knoten verzichtet. Beispielhaft sind zwei JobServer dargestellt. Ihre Anzahl kann beliebig groß sein. Auf dem Knoten *MessageQueue* sind die Komponenten GatewayQueue, ProcessRepository und ProcessCoordination verteilt. Die Komponente MQManager ist auf die Knoten *JobServer1*, *JobServer2* und *Gateway* verteilt. Die Komponente JobServer findet sich auf den Knoten *JobServer1* und *JobServer2* wieder. Die Knoten Gateway, JobServer1 und JobServer2 kommunizieren über die JDBC (Java Database Connectivity) Schnittstelle mit dem Knoten MessageQueue. Die JobServer hinterlegen die Resultate der Ausführung der IR-Module in Teilen ihres lokalen Dateisystem, die durch einen Webserver für andere Knoten verfügbar gemacht werden. Der Austausch der Resultate erfolgt über das HTTP-Protokoll.

4.4 Code-Migration

Wie in Abschnitt 3.3.2 beschrieben, kann zur flexiblen Skalierung der Ausführungsgeschwindigkeit der MOM die Technik der Code-Migration verwendet werden. Die Java-Plattform bietet von sich aus Unterstützung zur *schwachen Mobilität* von Code. Auszuführende Klassen können unter Verwendung der Klasse *java.net.URLClassLoader* von einer entfernten Maschine geladen werden. Die zu ladenen Klassen können u.a. in einer *jar*-Datei gebündelt und somit *in einem Rutsch* geladen werden. Das Laden der IR-Module übernimmt die Subkomponente *JarURLLoad* der Komponente *JobServer*.

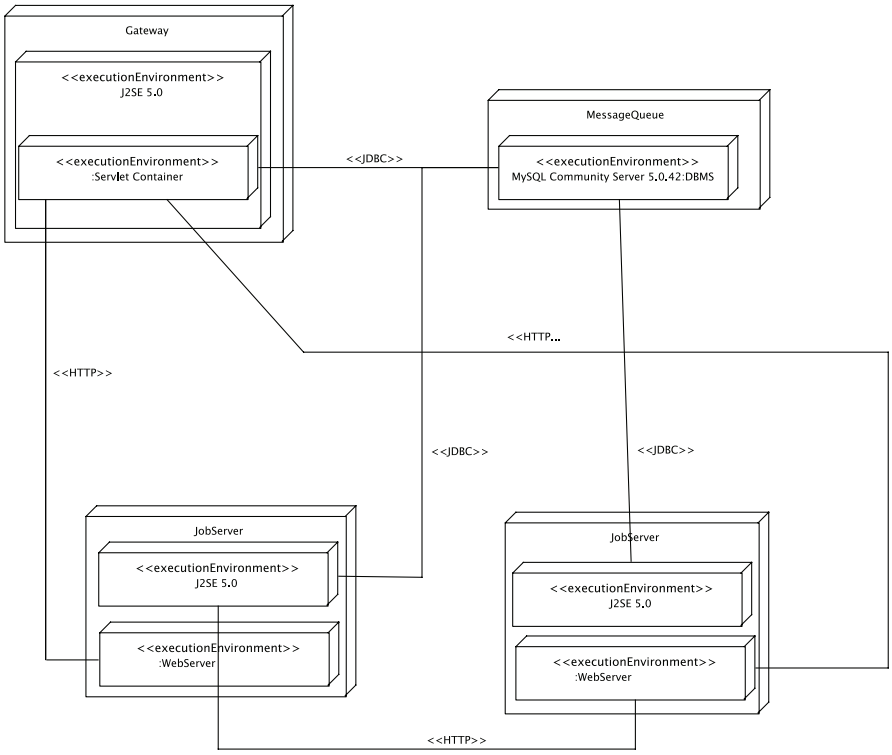

Abbildung 4.9: TI-Architektur der Middleware-Plattform

4.5 Transaktionen und Locking

Wie in Abschnitt 3.3.3 erklärt, werden Transaktionen zur Synchronisation paralleler Zugriffe (Mehrbenutzersynchronisation) verwendet. Eine Transaktion bündelt mehrere Operationen zu einer Einheit, die atomar fehlerfrei ausgeführt werden muss. Für folgende Situationen ist dies eine essentielle Anforderung an die Middleware-Plattform:

- Instanziierung, also die Erstellung einer Arbeitskopie, eines IR-Prozesses
- Übermittlung der Ein- und Ausgabedaten eines Jobs/Prozesses
- Koordination der Ausführungsabhängigkeiten (Token-Weitergabe)

Transaktionsunterstützung bietet MySQL nur für Tabellen vom Typ *InnoDB* und *BerkeleyDB*. Der am weitesten verbreitete Type ist InnoDB und wird auch in dieser Arbeit verwendet. Operationen, die mit transaktionalen Eigenschaften ausgeführt werden sollen, werden mit *START TRANSACTION* begonnen und mit *COMMIT* beendet. Genauere Beschreibungen der Transaktionsunterstützung bieten [10] und [4].
Da bei der Prozessverarbeitung vermehrt Schreiboperationen ausgeführt werden, wird eine pessimistische, sperrbasierte Strategie angewandt. Bei Verwendung der sperrbasierten Strategie ist in Verbindung mit Transaktionen darauf zu achten, dass die Sperren so kurz wie möglich gehalten werden sollten. Die Transaktionen sollten also möglichst wenige Anweisungen umfassen, da es sonst vermehrt zu *Deadlocks* kommen kann. Die verwendete Isolationsstufe (siehe Abschnitt 3.3.3) ist *Repeatable Read*.

Tritt ein Deadlock auf, so macht MySQL die kleinste, d.h. die Transaktion mit der geringsten Anzahl von ihr betroffenen Reihen rückgängig. Strategien zur Vermeidung bzw. Bewältigung von Deadlocks auf der MySQL-Plattform beschreiben [10] und [4].

4.6 Modelltransformation

Die Spezifikation eines IR-Prozesses ist eine Konfiguration der Middleware-Plattform. Ziel der Transformation eines Modells eines TIRA-GUI-Ablaufgraphs (TIRA-Modell - PIM) in ein Model eines DFG (MOM-Model - PSM) ist eine Konfiguration der Middleware-Plattform. Werden aus einem Modell nur eine geringe Anzahl nicht komplexer Artefakte, wie z.B. Konfigurationsdateien, generiert, liegen die Domäne, die modelliert wird, und die MDSD-Plattform nahe beeinander. In diesem Fall liegt, wie in Abschnitt 2.3.2 erläutert, eine reichhaltige, domänenspezifische Plattform vor. Dies hat eine Verringerung der Komplexität der Transformations- und Generierungsvorschriften zur Folge, da Domäne und Plattform sich auf der selben Abstraktionsebene (*Spezifikation und Ausführung von IR-Prozessen*) befinden. Des weiteren sind keine manuellen Eingriffe in das endgültig generierte Artefakt notwendig.
Für die Transformation wurde die *XSLT 2*-Technologie verwendet. Als XSLT-Prozessor kommt *Saxon* [14] zum Einsatz. In einem ersten Schritt wird das TIRA-Modell modifiziert, indem die Datenknoten entfernt werden. Da Informationen über die die Aktionsknoten ausführenden IR-Module im TIRA-Modell nicht vorhanden sind, wird dieses und ein Modell, das diese Modulinformationen enthält, in das MOM-Modell transformiert. Dieser Ablauf wird in Abb. 4.10 dargestellt. Eine Validierung der Eingabemodelle wird nicht vorgenommen, da die verwendete nicht kommerzielle Variante des Saxon-Prozessors dies nicht unterstützt.

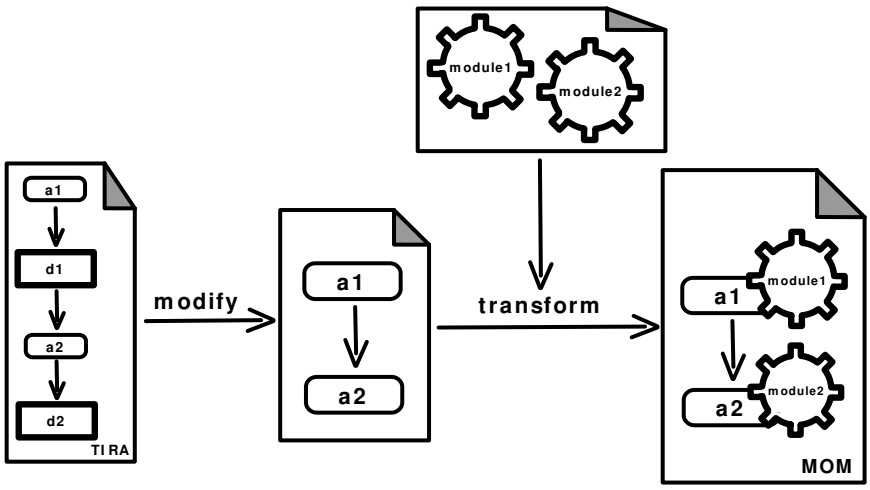

Abbildung 4.10: Der Workflow bei der Erstellung eines Modells, das die MOM auswerten und simulieren kann.

4.7 Programmiermodell

Wie in Abschnitt 2.2.1 erwähnt, soll TIRA bzw. deren Middleware-Plattform (MP) die Realisierung zukünftiger Anwendungen unterstützen. Die Frage ist nun, inwieweit sich die zukünftigen Anwendungen der MP anpassen müssen - welches Programmiermodell wird den Entwicklern auferlegt? Ein IR-Modul, muss eine strukturelle Anforderung erfüllen: Es muss eine Klasse enthalten, die eine Methode mit folgender Signatur aufweist:

```
1 public Map<String, String> methodenName (Map<String, String>)
```

Der Name der Methode ist frei wählbar. Die Werte der *Maps* des Rückgabewertes und des Parameters sind vom Typ *String*, da davon ausgegangen wurde, dass wie bei bisherigen Anwendungen auf Basis von TIRA die Daten in Form von *XML-Strings* unter den Modulen ausgetauscht werden. Die Module können ohne Beachtung der technischen Details der MP und ohne Annahmen über eine nebenläufige Ausführung implementiert werden. Dazu ein kleines Beispiel. *Modul1* führt auf Basis von Eingabedaten Berechnungen durch, deren Ergebnisse von *Module2* weiterverarbeitet werden sollen:

```
1  package modules;
2
3  public Class Module1{
4
5      public
6      Map<String, String> doThings (Map<String, String> data){
7
8           String input = data.get("module1Input");
9          String output = doCalculationWithData(input);
10         Map<String, String> result = new HashMap<String, String>();
11         result.put("module1Output", output);
12         return result;
13     }
14 }
```

Module2 , das die Ausgabedaten von *Module1* benutzt, würde folgendermaßen implementiert:

```
1  package modules;
2
3  public Class Module2{
4      public
5      Map<String, String> calculate (Map<String, String> data){
6
7          String input = data.get("module1Output");
8         String output = doIt(input);
9         Map<String, String> result = new HashMap<String, String>();
10        result.put("module2Output", output);
11        return result;
12     }
13 }
```

Die MP versorgt im Hintergrund die Module mit den benötigten Daten und kümmert sich um die Speicherung der Rückgabedaten. Die Module greifen einfach über die entsprechenden Schlüssel auf die Eingabe-*Map* zu, um an die Daten zu gelangen.
Damit die MP weiss, welche Klasse und Methode sie aufrufen soll, müssen ihr diese *Laufzeitinformationen* mitgeteilt werden. Die MP erwartet ein Modul verpackt in einer *jar*-Datei. Jede *jar*-Datei enthält standardmäßig im Ordner *META-INF* eine Datei *MANIFEST.MF*. Sie enthält Informationen, die von der Java-Plattform benötigt werden, um z.B. eine *jar*-Datei als ausführbar zu deklarieren. In dieser Datei werden zwei weitere Einträge gemacht: *Executor* und *Method*. *Executor* ist der Name der Klasse, die von der MP erzeugt und *Method* der Name ihrer Methode, die aufgerufen werden soll. Für das erste Modul (*Module1*) sieht die Datei so aus:

```
1 Manifest-Version: 1.0
2 Executor: modules.Module1
3 Method: doThings
```

Die MP liest diese Datei zur Laufzeit aus, lädt die entsprechende Klasse und ruft die deklarierte Methode auf.

Kapitel 5

Diskussion

In diesem Kapitel sollen die erzielten Resultate der Arbeit diskutiert werden. Ziel dieses Kapitels ist es, Ansatzpunkte für zukünftige Weiterentwicklungen aufzuzeigen.

5.1 Middleware-Plattform

Die Warteschlangen *GatewayQueue* und *JobServerQueue* (siehe Abschnitt 3.2) sind als unabhängige Datenbanken entworfen, aber im Zuge dieser Arbeit lediglich auf einer Maschine installiert. In Bezug auf die Verlässlichkeit des System ist dies nicht wünschenswert, da es sich bei dieser Maschine um einen *Single Point of Failure* handelt. Fällt diese Maschine aus, können Anwendungen, die auf der Middleware-Plattform aufbauen, ihren Dienst nicht zur Verfügung stellen, da Nutzeranfragen verloren gingen. Wie einleitend erwähnt, sollen die Warteschlangen bzw. Komponenten so voneinander entkoppelt werden, dass der Ausfall einer nicht die Funktionsweise der anderen beeinträchtigt. So soll eine kurzzeitige Nichterreichbarkeit der für die Ausführung der Prozesse verantwortlichen Komponenten nicht verhindern, dass Nutzer Anfragen an das System stellen können.

5.1.1 Verfügbarkeit

Verfügbarkeit beschreibt die Fähigkeit eines Systems bei Ausfällen bzw. Fehlern einzelner Komponenten seine Funktionstüchtigkeit aufrechtzuerhalten [23]. In diesem Zusammenhang wird auch von der *Up-Time* eines Systems - wie viel der Zeit steht das System den Nutzern zur Verfügung - gesprochen. Um die Up-Time eines Systems möglichst hochzuhalten, müssen einerseits die einzelnen Komponenten möglichst unabhängig voneinander funktionieren und andererseits müssen Vorkehrungen getroffen werden, die die Kompensation des Ausfalls einzelner Komponenten ermöglichen.
Eine Möglichkeit ist die redundante Speicherung der Prozessdaten auf einer weiteren Maschine. Beim Ausfall einer oder mehrerer Komponente kann dann auf diese Maschine ausgewichen werde. Dabei ist jedoch zu beachten, dass die anderen noch funktionstüchtigen Komponenten von diesem Wechsel benachrichtigt werden müssen. Für die Steigerung der Verfügbarkeit einer Anwendung

bietet MySQL zwei Technologien an: *MySQL Replication* und *MySQL Cluster*.

5.1.2 Transaktionen

Werden die verschiedenen Warteschlangen/ Komponenten verteilt, sind auch die Informationen über die sich in Ausführung befindlichen Prozesse über mehrere Maschinen verteilt. Laut [1] liegt somit eine verteilte Datenbank vor, die von einem *verteiltem Datenbankmanagementsystem (VDBMS)* kontrolliert wird. Bei der Kontrolle einer verteilten Datenbank ist zu beachten, dass die Mechanismen zur Transaktionskontrolle eines zentralisierten DBMS nicht ohne weiteres angewandt werden können. Als Beispiel sei die Instantiierung eines IR-Prozesses angebracht: Im Falle einer Verteilung der Komponenten GatewayQueue, ProcessRepository und ProcessCoordination (JobServerQueue) auf je eine Maschine erstreckt sich die die Instantiierung umspannende Transaktion über diese drei Maschinen. Diese *verteilte Transaktion* muss zunächst die Verfügbarkeit und die Fähigkeit, an der Transaktion teilzunehmen, der einzelnen Maschinen prüfen. Dies müssen die Maschinen bestätigen. Haben sie ihren Teil der Transaktion durchgeführt, bestätigen sie dies wiederrum. Man spricht von einem *Zweiphasen-Commit-Protokoll (2PC-Protokoll)*.
Verteilte Transaktionen werden in der momentanen Implementierung der Middleware-Plattform nicht berücksichtigt. Der verwendete MySQL-Server bietet aber eine Unterstützung für verteilte Transaktionen, die jedoch in dieser Arbeit nicht untersucht wurde.

5.1.3 Skalierbarkeit

Die in der MOM umgesetzten Mechanismen zur Skalierbarkeit beziehen sich lediglich auf die Verarbeitungsgeschwindigkeit einzelner Jobs. Was in dieser Arbeit nicht adressiert wurde, ist die Skalierbarkeit hinsichtlich der Verarbeitungskapazität der in das System eingestellten Nachrichten. Die Frage, was passiert, wenn die *GatewayQueue* aufgrund zu hoher Belastung keine Nutzeranfragen mehr entgegennehmen kann, kann mit der in dieser Arbeit implementierten Skalierungstechnik nicht beantwortet werden.
Um dieser Art der Überlastung begegnen zu können, könnte ein *Load-Balancing* eingeführt werden. Die Anfragelast würde dabei auf mehrere GatewayQueues verteilt. Für eine Umsetzung scheint die *MySQL Cluster*-Technologie geeignet zu sein [3]. Sie wurde in im Zuge dieser Arbeit nicht evaluiert.

5.1.4 Verwaltung von Nachrichten

In der Implementierung ist die Behandlung von nicht bearbeiteten Nachrichten nicht adressiert. Z.B. sollten Nachrichten mit einer Lebensdauer versehen werden können, auf deren Basis die MessageQueue regelmäßig nicht behandelte Nachrichten verwerfen und entsprechende Maßnahmen einleiten kann. Dazu ist eine Art *Timer*-Funktionalität notwendig, die das periodische Ausführen von Anweisungen ermöglicht. Mit den Mitteln der verwendeten Version 5.0.42 des MySQL-Servers ist dies nicht möglich. Vorstellbar wäre die Auslagerung dieser Funktionalität in die Komponente MQManager. Als Basis könnte das *Quartz-Scheduling*-Framework [22] verwendet werden. Mit der Version 5.1 des MySQL-Servers soll diese Art von Funktionalität in Form eines *Event Schedulers*

verfügbar sein.

5.1.5 Anfragehäufigkeit der JobServer

Die JobServer agieren vollkommen unabhängig von der Komponente ProcessCoordination. Sie werden nicht von dieser Komponente über zu bearbeitende Jobs benachrichtigt. Sie fragen sie selbst nach. In der momentanen Implementierung entspricht das Anfrageverhalten der Technik des *Aktiven Wartens* [36]. Die JobServer fragen ununterbrochen Jobs nach, auch wenn keine Jobs zu bearbeiten sind. In Zeiten geringer Gesamtlast der MOM verschwenden sie Ressourcen. Es ist vorstellbar, dass die JobServer ihr Anfrageverhalten selbstständig anpassen. Sie könnten aus der Bedienung bzw. Nicht-Bedienung ihrer Anfragen *lernen*. Die Generierung dieses Wissens kann mit maschinellem Lernen [39] umgesetzt werden.

5.2 Modellierung der IR-Prozesse

5.2.1 Entscheidungsknoten

Entscheidungsknoten im Sinne von UML Aktivitätsdiagrammen ermöglichen die Modellierung von alternativen Abläufen vergleichbar mit *if*-Verzweigungen in Java. Jede ausgehende Kante wird mit einer *Überwachungsbedingung* (ÜB) versehen, anhand derer entschieden wird, entlang welcher Kante ein Token weitergegeben wird. Auf Basis von Entscheidungsknoten können auch Schleifen modelliert werden.
Das Modell eines Datenflussgraphs (DFG) sieht jedoch die Auswertung solcher Bedingungen nicht vor. Die Erweiterung eines DFG um die Möglichkeit der Modellierung alternativer Abläufe heisst *Kontroll-Datenflussgraph (CDFG)*. In einem CDFG werden zusätzlich zu den Datenflusskanten *Kontrollflusskanten* (KFK) eingeführt. Eine KFK ist wie eine ausgehende Kante eines UML-Entscheidungsknotens mit einer ÜB versehen. Bei Beendigung einer von einem Knoten repräsentierten Operation werden alle ÜB der ausgehenden KFK ausgewertet. Entlang der KFK, deren ÜB als *wahr* ausgewertet worden ist, wird ein Token weitergegeben. Die ÜB der ausgehenden KFK müssen sich wechselseitig ausschließen.
Bei Erweiterung des in der MOM repräsentierten DFG zu einem CDFG ist darauf zu achten, dass die Auswertung der ÜB nicht ohne weiteres in der Komponente *ProcessCoordination* (PC) durchführbar ist. Der Grund ist die dezentrale Speicherung der Resultate der verschiedenen Jobs auf unterschiedlichen Maschinen. Da die PC auf Basis eines MySQL-Servers umgesetzt wurde, ist es für sie nicht möglich, auf die Daten via der ihr zur Verfügung stehenden URL des Resultates auf dieses zuzugreifen. Dies ist aber zwingend notwendig, da die Auswertung einer ÜB auf Basis der Resultate stattfindet.
Ein möglicher Lösungsansatz wäre die *Ausschreibung* der Auswertung einer ÜB als Job. Dieser Job würde in die Liste der anderen zu erledigenden Jobs eingereiht und entsprechend von einem JobServer ausgewertet. Ein Modul, das diese Auswertung vornimmt, müsste entsprechend des Programmiermodells implementiert werden. Das Resultat würde nicht wie bei *normalen* Jobs als Referenz in Form einer URL an die PC übertragen sondern direkt in einer für sie auswertbaren

Form.

5.2.1.1 Modelltransformation

Soll der DFG zu einem CDFG erweitert werden, muss die Transformation des Platform Independent Model (PIM) in das Platform Specific Model (PSM) angepasst werden. Hierbei stellt sich die Frage, wie Entscheidungsknoten eines UML Aktivitätsdiagramms auf einen CDFG abgebildet werden. [27] beschreibt ein mögliches Vorgehen, dass sich am Konzept der *Compound Transitions* [7] orientiert. Abb. 5.1 zeigt dieses Vorgehen. Die eingehende Kante eines Entscheidungsknotens wird mit je einer der ausgehenden Kanten zu einer neuen Kante *verschmolzen*. Die so entstandenen KFK werden jeweils mit der entsprechenden ÜB versehen.

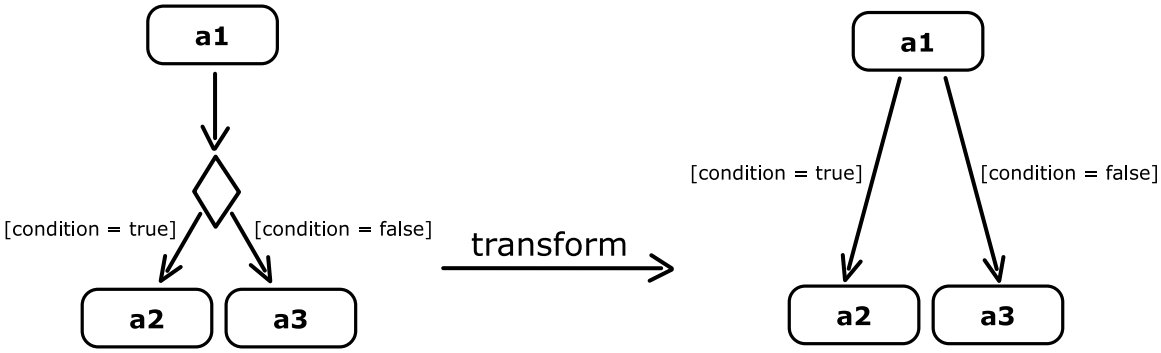

Abbildung 5.1: Abbildung (Transformation) eines Entscheidungsknotens in einem Aktivitätsdiagramm (links) auf einen Kontroll/Datenflussgraph (rechts)

5.2.2 Modellierungswerkzeuge

Zum jetzigen Zeitpunkt findet die Modellierung (Spezifikation) der IR-Prozesse mit Hilfe der TIRA-GUI statt. Diese unterstützt jedoch nicht die Modellierung von Parallelisierungs-, Synchronisierungs- und Entscheidungsknoten. Da ihre Erweiterung zusätzlichen Aufwand bedeutet, sollte überlegt werden, ob nicht andere Modellierungswerkzeuge verwendet werden sollten. Die MOM macht keine Annahmen über das verwendete Modellierungswerkzeug. Das Eingabeformat eines modellierten IR-Prozesses muss dem des verwendeten DFG bzw. bei erfolgter Erweiterung dem des CDFG entsprechen. Würde ein anderes Modellierungswerkzeug verwendet, muss lediglich eine neue Modelltransformation vom Ausgabeformat des Modellierungswerkzeugs hin zum (C)DFG implementiert werden. Die Modelltransformation ist eine Umsetzung des *Adapter*-Patterns [35]. Als Modellierungswerkzeuge kommen z.B. UML Editoren in Frage, die über ein XML-Ausgabeformat verfügen wie z.B. [34].

Literaturverzeichnis

[1] A. Eickler A. Kemper. *Datenbanksysteme*. Oldenbourg, 2004.

[2] M.v. Steen A. Tanenbaum. *Verteilte Systeme*. Pearson Studium, 2003.

[3] MySQL AB. High availability, scalability, and drbd, 2007. Online verfügbar unter `http://dev.mysql.com/doc/refman/5.0/en/ha-overview.html`; [letzter Zugriff am 5. August 2007].

[4] MySQL AB. Mysql 5.0 reference manual, 2007. Online verfügbar unter `http://dev.mysql.com/doc/refman/5.0/en/index.html`; [letzter Zugriff am 5. August 2007].

[5] American National Standards Institute (ANSI). Database language sql. Technical Report X3.135, American National Standards Institute (ANSI), 1992.

[6] C. Bock. Uml2 activity and action models. *Journal of Object Technology*, 2(4):43–53, July-August 2003.

[7] A. Naamad D. Harel. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[8] IBM Developerworks. Business process execution language for web services version 1.1, 2007. Online verfügbar unter `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`; [letzter Zugriff am 29. August 2007].

[9] D. Frankel. *Model Driven Architecture*. John Wiley & Sons Ltd., 2003.

[10] S. Feuerstein G. Harrison. *MySQL Stored Procedure Programming*. O'Reilly, 2006.

[11] Google. Google, 2007. Online verfügbar unter `http://www.google.de`; [letzter Zugriff am 29. August 2007].

[12] Interface21. Spring framework, 2007. Online verfügbar unter `http://www.springframework.com/`; [letzter Zugriff am 29. August 2007].

[13] E. Denert J. Siedersleben. Wie baut man informationssysteme? *Informatik Spektrum*, 23(4):247–257, August 2000.

[14] Michael H. Kay. Saxon the xslt and xquery processor, 2007. Online verfügbar unter http://saxon.sourceforge.net/; [letzter Zugriff am 30. August 2007].

[15] J. Siedersleben M. Broy. Objektorientierte programmierung und softwareentwicklung eine kritische einschätzung. *Informatik Spektrum*, 25(1):3–11, Februar 2003.

[16] G. Kappel E. Kappsammer W. Retschitzegger M. Hitz. *UML @ Work*. dpunkt.verlag, 2005.

[17] M. Stahl M. Völter. *Modellgetriebene Softwareentwicklung*. dpunkt.verlag, 2005.

[18] Sven Meyer zu Eissen and Benno Stein. Service-orientierte architekturen für information retrieval. In *Proceedings of the Workshop Information Retrieval 2006 of the Special Interest Group Information Retrieval (FGIR) in conjunction with Lernen – Wissensentdeckung – Adaptivität 2006 (LWA '06)*, pages 77–83, 2006.

[19] Object Management Group (OMG). Mda guide version 1.0.1, 2003. Online verfügbar unter http://www.omg.org/docs/omg/03-06-01.pdf; [letzter Zugriff am 5. August 2007].

[20] Object Management Group (OMG). Uml superstructure, v2.0, 2005. Online verfügbar unter http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf; [letzter Zugriff am 29. August 2007].

[21] Object Management Group (OMG). Uml superstructure, v2.1.1, 2007. Online verfügbar unter http://www.omg.org/cgi-bin/apps/doc?formal/07-02-05.pdf; [letzter Zugriff am 5. August 2007].

[22] OpenSymphony. Quartz enterprise job scheduler, 2007. Online verfügbar unter http://www.opensymphony.com/quartz/; [letzter Zugriff am 5. August 2007].

[23] A. Longshaw P. Dyson. *Architecting Enterprise Solutions - Patterns for High-Capability Internet-Based Systems*. John Wiley & Sons Ltd., 2004.

[24] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Bonn, 1962.

[25] TIRA Projektgruppe. Tira text-based information retrieval architecture, 2006. Online verfügbar unter http://www.uni-weimar.de/medien/webis/research/aitools/wiki/doku.php?id=tira; [letzter Zugriff am 30. August 2007].

[26] S.B. Navathe R. Elmasri. *Fundamentals of Database Systems*. Addison Wesley, 2006.

[27] R. Wieringa R. Eshius. A formal semantics for uml activity diagrams - formalising workflow models. Technical Report ISSN 13813625, University of Twente, 2001.

[28] Bernd Schalbe. Skript zur vorlesung konkurrente systeme, 2004.

[29] J. Siedersleben. *Moderne Softwarearchitektur*. dpunkt.verlag, 2006.

[30] Inc. Sun Microsystems. Data access object, 2002. Online verfügbar unter http://java.sun.com/blueprints/patterns/DAO.html; [letzter Zugriff am 5. August 2007].

[31] Inc. Sun Microsystems. Java se downloads, 2007. Online verfügbar unter http://java.sun.com/javase/downloads/index.jsp; [letzter Zugriff am 29. August 2007].

[32] C. Szypersky. *Component Software - Beyond Object-Oriented Programming*. Addison Wesley, 1999.

[33] A. Tanenbaum. *Moderne Betriebssysteme*. Pearson Studium, 2002.

[34] TOPCASED. Topcased homepage, 2007. Online verfügbar unter `http://www.topcased.org/`; [letzter Zugriff am 5. August 2007].

[35] E. Gamma R. Helm R. Johnson J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Longman Publishing Co., Inc., 1998.

[36] Wikipedia. Aktives warten, 2007. Online verfügbar unter `http://de.wikipedia.org/wiki/Aktives_Warten`; [letzter Zugriff am 29. August 2007].

[37] Wikipedia. Design by contract, 2007. Online verfügbar unter `http://de.wikipedia.org/wiki/Design_By_Contract`; [letzter Zugriff am 5. August 2007].

[38] Wikipedia. Informationssystem (informatik), 2007. Online verfügbar unter `http://de.wikipedia.org/wiki/Informationssystem_%28Informatik%29`; [letzter Zugriff am 5. August 2007].

[39] Wikipedia. Maschinelles lernen, 2007. Online verfügbar unter `http://de.wikipedia.org/wiki/Maschinelles_Lernen`; [letzter Zugriff am 29. August 2007].