Diplomarbeit

Praktische Evaluierung der aspektorientierten Programmierung zur Stärkung der Wandlungsfähigkeit eines ERP-Systems

Oliver Neumann <beast@cs.tu-berlin.de>

Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik
Prof. Dr. Ing. Stefan Jähnichen

Betreut durch:

Dr. Stephan Herrmann
&
Dipl.-Inform. Anne Lämmer
Lehrstuhl für Wirtschaftsinformatik und Electronic Government
Universität Potsdam

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den / Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Motivation

Das Aufkommen der globalen Marktwirtschaft in den 60'er Jahren des letzten Jahrhunderts eröffnete den Unternehmen völlig neue Märkte und Möglichkeiten des Wachstums. Diese, noch immer anhaltende, Entwicklung führte zwangsläufig zu einer stärkeren Konzentration an Konkurrenzunternehmen, wobei diese den Druck auf ein Unternehmen erhöhen. Innovationen, Produktionskosten, Kontakt zum Kunden oder die geographische Nähe zu den Absatzmärkten sind heute wichtige Faktoren im ökonomischen Wettstreit der Unternehmen.

Seit den Anfängen der Globalisation werden Methodiken zur Planung oder Optimierung der Prozessketten verwendet, um in diesem Wettstreit konkurrenzfähig zu bleiben. Diese Entwicklung schreitet bis zum heutigen Tage voran. Die Einführung einer computergestützten Planung und Kommunikation führte zu einer weiteren Effizienzsteigerung und stellte damit einen Wettbewerbsvorteil dar, der die Entwicklung der sogenannten Unternehmenssoftware förderte. Der Begriff Unternehmenssoftware bezeichnet dabei eine Software, welche typischerweise verteilt lauffähig ist und eine oder mehrere betriebliche Bereiche abdeckt. Eine Gruppierung solcher Unternehmenssoftware bildet der Bereich zur Planung von Ressourcen, sei diese nun in menschlicher, materieller oder auch finanzieller Form. Allgemein bezeichnet man diese als *Enterprise Ressource Planning*-Software.

Eine Software, welche in die betrieblichen Prozesse eingebunden ist, erfordert einen möglichst hohen Grad an *Wandlungsfähigkeit*, um sich schnell und effizient den dynamischen Prozessen anpassen zu können. Eine Erhöhung der Wandlungsfähigkeit von ERP-Systemen ist die Bestrebung zahlreicher Forschungsinstitute und ERP-System-Anbieter.

Dem gegenüber steht in der Softwareentwicklung der Wunsch nach mehr Flexibilität und Modularität. Eine Entwicklung die dieses Ziel verfolgt, ist die der aspektorientierten Programmierung (*AOSD - Aspect Oriented Software Development*). Diese Arbeit stellt den Versuch dar, die beiden genannten Elemente vorteilhaft zu verknüpfen. Dazu wird die Wandlungsfähigkeit eines Open-Source-ERP-Systems untersucht, um anschließend die positiven Einflüsse bei der Verwendung von Aspektorientierung im Sinne der Wandlungsfähigkeit aufzuzeigen. Als Grundlage dient dabei das Java-basierende Open-Source-ERP-System *Compiere*. Unter Verwendung des aspektorientierten Programmiermodells *ObjectTeams* wird dieses im weiteren Verlauf dieser Arbeit mit Aspekten angereichert werden. Die Erweiterung zielt hierbei auf eine Stärkung einzelner Kriterien der Wandlungsfähigkeit ab, um dadurch eine Steigerung der Gesamtwandlungsfähigkeit zu erreichen.

1.2 Struktur

Die vorliegende Arbeit ist unterteilt in die Kapitel:

- *Kapitel 2 - Enterprise Resource Planning*

  Dieses Kapitel gibt einen Überblick über den Begriff des Enterprise Resource Planning und erläutert dessen Aufbau und Bedeutung innerhalb eines Unternehmens.

- *Kapitel 3 - Wandlungsfähigkeit*

  Der Begriff der Wandlungsfähigkeit im Kontext von ERP-Systemen wird in diesem Kapitel näher erläutert.

- *Kapitel 4 - Compiere*

  Das Kapitel 4 gibt einen Einblick in den Aufbau und der Funktionsweise des ERP-Systems Compiere. Ebenso werden hier die verwendeten Technologien von Compiere, wie dem Applikations-Server JBoss, kurz erläutert.

- *Kapitel 5 - Aspektorientierte Softwareentwicklung*

  Hier werden die Konzepte der Aspektorientierung allgemein und speziell die für diese Arbeit relevanten, aspektorientierten Programmiermodelle vorgestellt und näher erläutert.

- *Kapitel 6 - Compiere Monitor*

  Kapitel 6 behandelt die Implementierung des Compiere Monitor. Dieser nutzt verschiedene Techniken der vorhergehenden Kapitel zur Einbettung unterschiedlicher Funktionalitäten in Compiere über die Verwendung von Aspekten.

- *Kapitel 7 - Ergebnis und Zusammenfassung*

  Abschließend gibt Kapitel 7 eine Übersicht der erzielten Steigerung der Gesamtwandlungsfähigkeit des ERP-Systems Compiere und zeigt die dabei einwirkenden Vorteile der Verwendung von Aspektorientierung.

2 Enterprise Resource Planning

2.1 Begriffsklärung und Aufgabendefinition

Die Anforderungen heutiger Unternehmen an Bereiche wie Logistik, Buchhaltung oder Verkauf sind vielschichtig, wobei die Kernanforderungen bei allen in der Planung von Ressourcen, sei es in personeller, materieller oder finanzieller Form, liegt. Eine möglichst effiziente Verteilung der vorhandenen Ressourcen auf die Geschäftsprozesse wird somit zu einer wichtigen Aufgabe innerhalb eines Unternehmens, dem *Enterprise Resource Planning* (ERP). Für den Begriff ERP existieren in der einschlägigen Literatur verschiedenste Definitionen. Aus diesem Grund soll der Begriff für die vorliegende Arbeit explizit festgelegt werden. Dazu wird im Folgenden die Klärung der Aufgaben und Funktionen vorgenommen.

Die Aufgaben und Eigenschaften des ERP können zusammenfassend beschrieben werden als [WK01]:

- eine Sammlung unternehmensweiter Programme zum Management von Ressourcen im Zyklus von Angebot und Nachfrage
- die Fähigkeit den Kunden und Anbieter in einer kompletten Lieferkette miteinander zu verbinden
- die Verwendung bewährter Unternehmensprozesse zur Entscheidungsfindung zu unterstützen
- die Eigenschaft einen hohen Grad an Inter-funktioneller Integration zwischen den Bereichen Verkauf, Marketing, Herstellung, Betrieb, Logistik, Einkauf, Finanzen, Produktentwicklung und menschlicher Ressourcen zu bieten
- der Unterstützung des unternehmerischen Betriebs, speziell den Bereichen Kundendienst und Produktivität, auf einem hohen Niveau bei gleichzeitig geringeren Kosten und nötigen Lagerkapazitäten
- der Bereitstellung der Basis eines effektiven e-Commerce

In [Gro04] wird des Weiteren als Wesensmerkmal von ERP-Systemen die Integration verschiedener Funktionen, Aufgaben und Daten in ein gemeinsames Informationssystem genannt, wobei der wesentliche Vorteil in der Automatisierung von Abläufen und in einer Standardisierung von Prozessen liegt. Der Begriff Integration umfasst in diesem

Zusammenhang zwei Dimensionen (Abb. 2.1). Zum einen gibt es die horizontale Integration. Sie beschreibt die abteilungs- bzw. funktionsübergreifenden Prozesse, wie z.B. einen gemeinsamen Prozess aus den Bereichen Vertrieb und Lagerhaltung. Senkrecht dazu entsteht die zweite Dimension, die der vertikalen Integration. Sie umfasst alle Ebenen der Aufgaben, welche auf einen Bereich angewandt werden können, so z.B. die Ebenen der Administration oder der Planung des gemeinsamen Wirkbereichs Vertrieb.

Der Umfang einer vollzogenen Integration wird von einem Integrationsgrad beschrieben. Dieser gibt an, wie viele betriebliche Funktionen in dem ERP-System vereinigt wurden. Typischerweise ist der Integrationsgrad abhängig von den betrachteten Bereichen, d.h. Bereiche wie die Logistik oder Produktion bilden Bereiche besonders hoher integrativer Möglichkeiten. Für diese Bereiche, in [Gro04] Integrationsinseln genannt, entstehen in Unternehmen oft eigenständige ERP-Systeme. Nur wenn alle diese Integrationsinseln in einem gemeinsamen ERP-System zusammengefasst sind, spricht man von einem *voll integrierten ERP-System.*

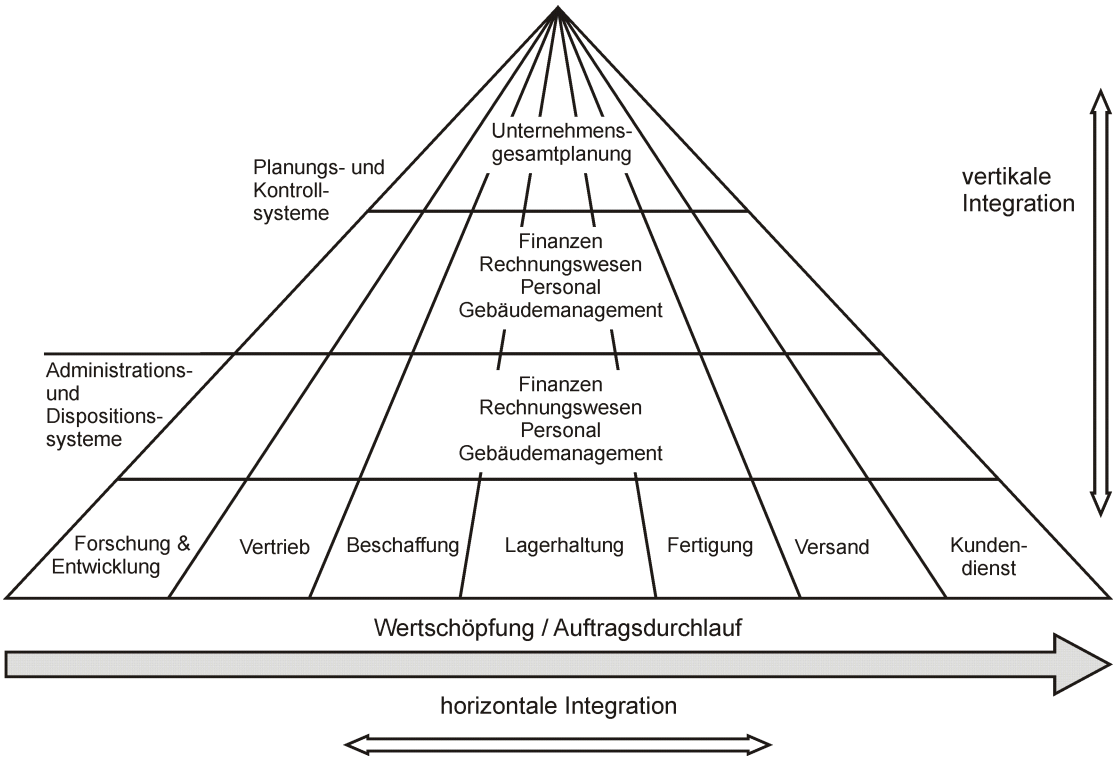

Abbildung 2.1: Horizontale und vertikale Integration betrieblicher Anwendungssysteme (nach [Gro04])

Die Einbettung von ERP innerhalb der von Unternehmen eingesetzten betrieblichen Informationssysteme zeigt Abbildung 2.2. Im Zentrum steht dabei die ERP-Komponente, wobei das *Supply Chain Management* (SCM) und das *Customer Relationship Management* (CRM) die Verbindung in die Bereiche ausserhalb des Unternehmens darstellen. Sie stellen die Verknüpfung vom innerbetrieblichen ERP-System zum ausserbetriebli-

chen Lieferanten bzw. Kunden dar. Diese beiden Begrifflichkeiten werden im Folgenden nun kurz beschrieben werden.

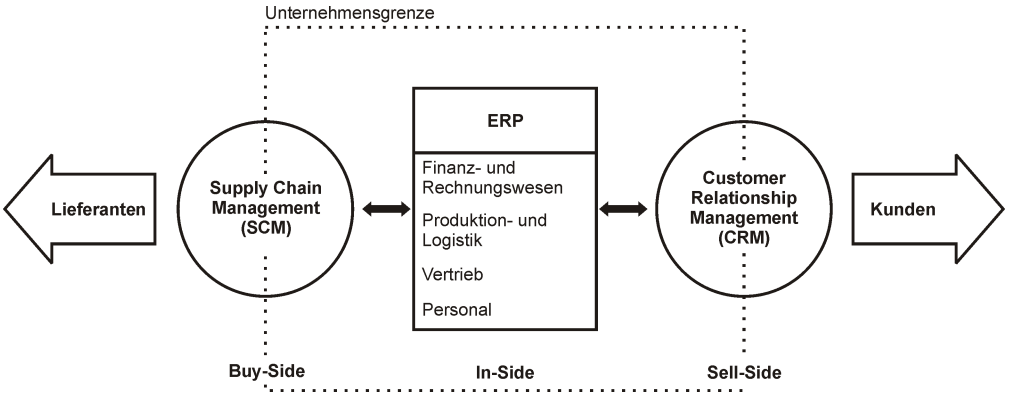

Abbildung 2.2: Aufgabenverteilung innerhalb betrieblicher Informationssysteme

Supply Chain Management

„Nothing Entirely New...Just a Significant Evolution“ [Hug03]

Der Begriff Supply Chain Management beschreibt eine Lieferkette, Versorgungskette oder auch eine unternehmensübergreifende Wertschöpfungskette. Diese Kette erstreckt sich dabei typischerweise vom Lieferanten bis hinüber zum Kunden und kann auch Querverbindungen zu verschiedenen Organisationen beinhalten. Sie dient der unternehmensübergreifenden Koordination der Material- und Informationsflüsse über den gesamten Wertschöpfungsprozess hinweg, d.h. von der Rohstoffgewinnung über die einzelnen Veredelungsstufen bis hin zum Kunden. Ziel ist hierbei, den Gesamtprozess sowohl zeit- als auch kostenoptimal zu gestalten. [BD04][SRJ99]

Customer Relationship Management

Die Ausrichtung auf den Kunden wird zunehmend zu einem kritischen Erfolgsfaktor innerhalb eines Unternehmens. Durch eine geringe Loyalität des Kunden gegenüber den Unternehmen sind diese aufgefordert, die Wünsche des Kunden zu erkennen und in ihrem Handeln zu berücksichtigen. Das Customer Relationship Management, im Weiteren auch als CRM bezeichnet, soll bei dieser Aufgabe helfen. Es umfasst die gesamte Interaktion eines Unternehmens mit bestehenden und zukünftigen Kunden während des gesamten Prozesses der Kaufentscheidung und des Besitzzeitraums. Das CRM beinhaltet damit die Planung, Durchführung, Kontrolle und Anpassung aller Unternehmensaktivitäten, wie Marketing oder Service, mit dem Ziel die Kundenbeziehungen zu optimieren. Um die Isolierung der einzelnen Unternehmensbereiche zu lockern, wird durch CRM eine globale Kundendatenbank etabliert, welche jedem Bereich Zugriff auf dieselben Kundendaten ermöglicht. Dies verringert die Informationsverluste bei seperaten Kundendaten für jeden

Bereich und erhöht damit die Effizienz des Kundenkontaktes. Typische Funktionalität eines CRM ist daneben unter anderem das Bereitstellen eines Helpdesk, Kundendienstes oder die Möglichkeit einer Kundenwertanalyse. [Gro04][HS00]

Letztendlich kann aber auch CRM allein keine Kundenfreundlichkeit gewährleisten, sondern dient nur zur Unterstützung derselbigen. Dieser Umstand wird in [Bre03] wie folgt dargelegt:

> „...spielt die Technik im Gesamtkontext des Customer Relationship Managements keineswegs die dominierende Rolle. Denn eines ist klar: Beziehungen werden nicht von Computern aufgebaut und gepflegt, sondern einzig und allein von Menschen... Allerdings hängt der Erfolg neben Mitarbeitern und Technik auch von der organisatorischen Gesamtausrichtung eines Unternehmens ab. Kundenfeindlich strukturierte Geschäftsprozesse werden allein durch guten Willen und moderne Technik nicht zu kundenfreundlichen."

2.2 Entstehungsgeschichte

Das Bedürfniss nach besseren Methoden zur Material- und Komponentenbestellung führte um 1960 zur Entwicklung des *Material Requirement Planning (MRP)*. Primär ging es dabei um die Erfüllung der grundlegenden Fragen bei der Herstellung eines Produkts [WK01]:

- Was möchte ich herstellen?
- Was benötige ich für die Herstellung?
- Was haben wir auf Lager?
- Was müssen wir bestellen?

Sumner lokalisiert die Einführung des MRP jedoch 10 Jahre später, d.h. um 1970 [Sum05]. Das Planungssystem in den 60'ern bezeichnet sie dabei als *Reorder point system*. Dabei legt sie den Schwerpunkt der Unterscheidung auf den jeweiligen Fokus der beiden Systeme. Resultierend daraus stehen Marketing und eine höhere Integration der Produktionsplanung bei MRP einem Kostenfokus auf Seiten des Reorder point system gegenüber.

Der Effizienzvorteil durch MRP säte den Keim für den Erfolg von ERP. Auf dem Weg zu dem umfassenderen ERP machte MRP noch einige Evolutionsschritte durch. Der erste vollzog sich durch die Hinzunahme von Priorisierungen, dem *priority planning*, und Komponenten zur Einschätzung des zukünftigen Kapazitätsbedarfes, dem *capacity planning*, innerhalb des MRP. Aus diesen Verbesserungen ging das *closed-loop MRP* hervor. An der dritten Evolutionsstufe steht dann das *Manufacturing Resource Planning (MRP II)*, welches in [Sum05] um 1980 lokalisiert wird und zusätzlich folgende Elemente beinhaltete:

- Planung von Verkauf & Betrieb zur Anpassung der Produktion an die bestehende Nachfrage und einer besseren Kontrolle der damit verbundenen betrieblichen Aspekte

- Transformation von betrieblichen Einheiten, wie Anzahl oder Gewicht, in finanzielle, wie Euro oder Dollar

- Simulation von Aktionen zur Bestimmung der Auswirkungen und Seiteneffekten

Besonders das Element zur Simulation war in der Stufe der Evolution noch unterentwickelt, sollte sich dann aber bis zur folgenden und aktuellen Stufe des ERP bzw. ERP II weiter ausbauen. Gronau nennt an dieser Stelle in [Gro04] zusätzlich noch zwei weitere Funktionen die als Evolutionselemente von MRP zu MRP II genannt werden:

- Beschaffung (Einkauf)

- Zeitwirtschaft als Erweiterung der mengenorientierten Materialwirtschaft

Als Erweiterung von MRP II ist ERP, aufgekommen in den späten 90'ern, effizienter geworden und bietet neben einem erweiterten Anwendungsbereich auch eine verstärkte Ingeration der Geschäftsprozesse. ERP kann als unternehmensweite Ansammlung von Programmen zur Vorhersage, Planung und Terminierung verstanden werden und bietet eine Vielzahl von Vorteilen, welche in Kapitel 2.1 beschrieben wurden. Eine weitere Stufe des ERP zeichnet sich mit ERP II ab, bei welchem der Bedarf an offenen Schnittstellen zum Internet im Mittelpunkt steht. Einen kurzen Überblick über die Entwicklung der genannten Planungssysteme soll mit Abbildung 2.3 gegeben werden.

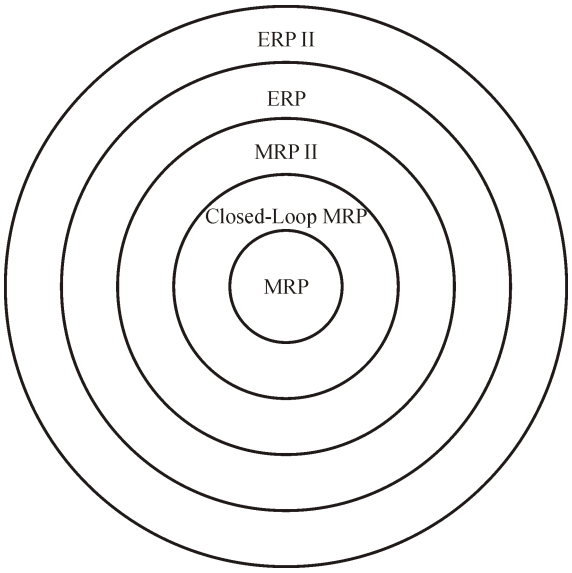

Abbildung 2.3: Evolutionsschritte des ERP

Mit dem Aufkommen der ersten ERP-Systeme geht auch das Erscheinen des heutigen Marktführers SAP[1] im Jahre 1972 einher. Ziel dieser 3-Mann-Firma aus Mannheim war anfangs die bessere Integration von Geschäftsprozessen und Verknüpfung von Daten zur Effizienzsteigerung des Geschäftbetriebs. Der Erfolg von SAP gab der Idee von ERP Recht und war der Anfang einer Entwicklung, die bis in die heutigen Tage reicht. Von dem anfänglichen Schwerpunkt auf den Bereich größerer, globaler Unternehmen bildeten sich im Folgenden ERP-Systeme heraus, welche für *kleine- und mittelständische Unternehmen (KMUs)* konzipiert sind. Ein Beispiel solch eines ERP-Systems ist *Compiere*, das in Kapitel 4 beschrieben wird und welches als Grundlage späterer Betrachtungen in dieser Arbeit dienen wird.

2.3 Enterprise Systeme

Die Realisierung einer effizienten Ressourcenplanung in einem Unternehmen wird mit zunehmender Anzahl der zu planenden Ressourcen immer aufwendiger und damit kostenintensiver. Um dem entgegenzuwirken, stehen Tools zur Verfügung, welche die Aufgaben des ERP unterstützen können. Sie werden als *Enterprise Software* (ES) bezeichnet und in [Dav00] wie folgt beschrieben:

> *"...packages of computer applications that support many, even most, aspects of a company's information needs."*

Jedoch dient lediglich eine Untermenge der vorhanden *Enterprise Software* zur Realisierung der Aufgaben der Ressourcenplanung (Abb. 2.4). Um eine maximale Abdeckung der von ES unterstützten ERP-Prozesse zu gewährleisten, bedarf es einer auf diese *ERP-Prozesse* spezialisierten Enterprise Software - der *ERP-Software*, oder im Folgenden auch als ERP-Lösung bezeichnet. [WK01]

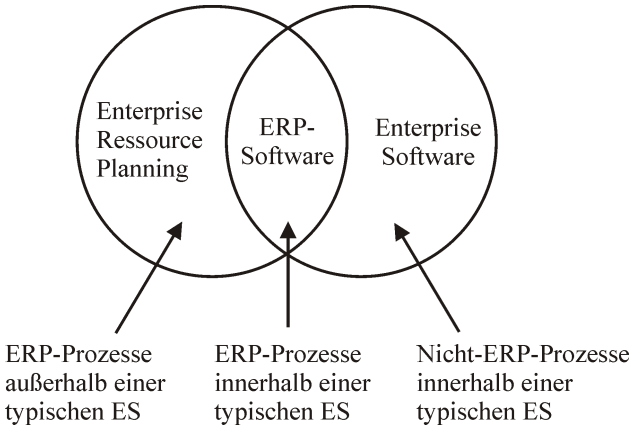

Abbildung 2.4: ERP-Prozesse in Enterprise Software

[1] Mit einem mehr als klaren Marktanteil von 54,8% Im Jahre 1999 ist SAP auch zum heutigen Tage Marktführer unter den ERP-System-Anbietern. [tusT]

Das Ausrollen einer ERP-Lösung innerhalb eines Unternehmens bedeutet jedoch nicht zwingend einen sofortigen Vorteil gegenüber der Konkurrenz. Es kann sogar zu einem anfänglichen Nachteil durch entstehende Kosten und Einfluss auf die laufenden Prozesse kommen. Die Benutzung von ERP-Lösungen bietet im Gegenzug aber das Potential zur Verbesserung der Prozesse, Beschleunigung der Zyklen und der Informationsflüsse allgemein sowie zu einem besseren Finanzmanagement. [Dav00]

Das erst spät einsetzende Wirken der Vorteile bei der Nutzung von ERP-Lösungen, stellt einen Hemmfaktor bei der Entscheidung für eine ERP-Lösung dar. Daneben spielen auch noch die Kosten/Nutzen-Faktoren für Unternehmen eine wichtige Rolle. Besonders die Software- und Consultingkosten bilden den Hauptanteil der Kosten eines ERP-Systems. So wird in [Sum05] davon ausgegangen, dass eine Kostenersparniss durch die Einführung eines ERP-Systems erst ab dem Zeitpunkt einsetzt, an dem Mitarbeitertraining und Implementierung abgeschlossen sind. Dies sind Faktoren, aus welcher eine Lebensdauer der ERP-Lösungen von 10 Jahren und mehr resultiert.

Die dynamischen Anforderungen des Unternehmens während dieses Zeitraums werden somit zu einer ständigen Herausforderung für eine Lösung. Um dem gerecht zu werden, müssen die Spezifikationen der ERP-Lösung an den Grad der benötigten Flexibilität angepasst werden. Neben den funktionalen Eigenschaften treten somit technische Charakteristika in den Vordergrund der Entwicklung neuer ERP-Lösungen. Einige der technischen Merkmale treten hierbei besonders hervor. Dies sind zum Beispiel die Zukunftsaussichten der Produkte, auf welche die Lösung aufbaut, sowie der Grad der Anpass- und Erweiterbarkeit der Lösung selbst. Sie sind es, die im hohen Maße etwas über die Fähigkeiten einer Lösung aussagen können, . Eine solche Fähigkeiten ist zum Beispiel, sich an dynamische ERP-Prozesse anpassen zu können um so eine stetig hohe Abdeckung der ERP-Prozesse, welche durch ES unterstützt werden können, gewährleisten. Die dafür nötigen Architekturen und Techniken werden Bestandteil späterer Kapitel sein.

2.4 Aufbau eines ERP-Systems

Unabhängig von den jeweiligen spezifischen Charakteristika eines ERP-Systems weisen alle ERP-Systeme einen mehrstufigen Aufbau auf. (Abb. 2.5)

Die Abstufung findet dabei über 4 vertikale Schichten statt:

1. *Datenhaltungsschicht*
2. *Applikationsschicht*
3. *Adaptionsschicht*
4. *Präsentationsschicht*

Jede dieser Schichten wird im Folgenden kurz beschrieben werden.

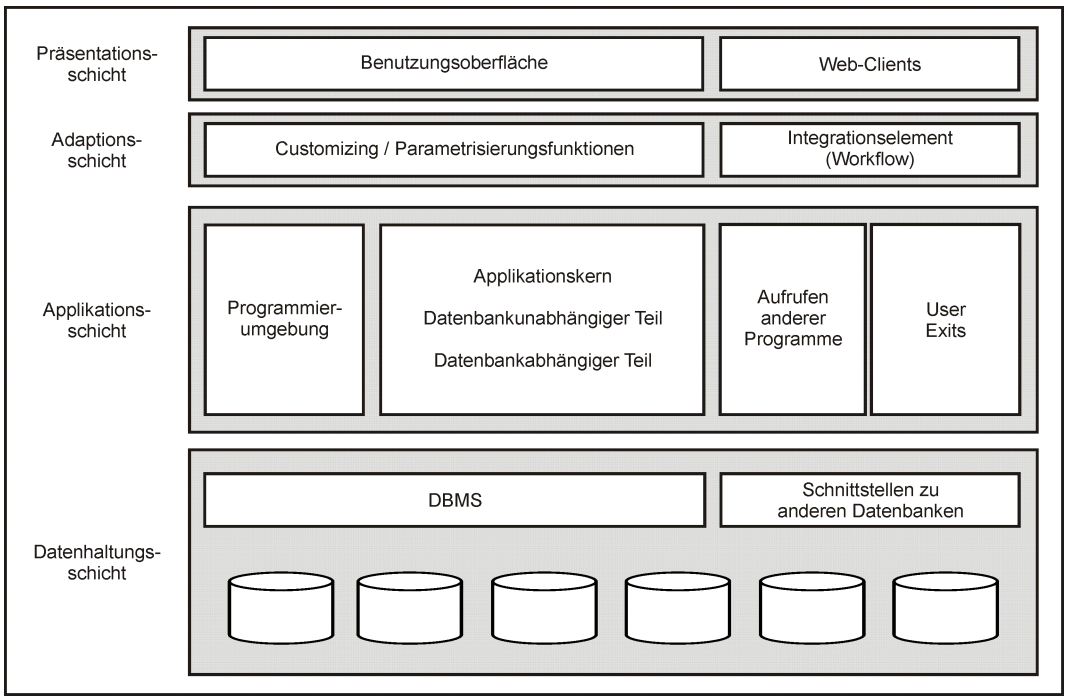

Abbildung 2.5: Aufbau eines ERP-Systems (nach [Gro04])

Datenhaltungsschicht Diese unterste Schicht bildet das Fundament auf das alle anderen Schichten aufbauen. Sie dient der Datenhaltung und besteht aus Datenbanken, die über ein *Datenbankmanagementsystem (DBMS)*, wie Oracle Database oder MySQL, verwaltet und angesprochen werden. Sie ermöglicht einen transparenten Zugriff höhererliegender Schichten auf diese Datenbanken. Zusätzlich kann diese Schicht auch Schnittstellen zu anderen Datenbanken, welche nicht zwingend zum eigenen Informationssystem gehören müssen, beinhalten.

Applikationsschicht Der funktionale Kern des gesamten ERP-Systems mitsamt der dazugehörigen Business-Logik befindet sich in dieser Schicht. Aus diesem Grund gehört zu dieser Schicht auch typischerweise eine Programmierumgebung, mit welcher Funktionalitäten bis zu einem spezifischen Grad erweitert oder ergänzt werden können. Desweiteren existiert in dieser Schicht eine Trennung zwischen dem datenbankabhängigen und datenbankunabhängigen Teil. Dies erlaubt eine individuelle Ausrichtung des datenbankabhängigen Teils auf die spezifischen Merkmale der jeweiligen Datenbank. Daneben ist es möglich über *Remote Procedure Calls* (RPC) andere Programme aufzurufen, um deren Funktionalitäten zu nutzen. Die Möglichkeit der Integration von Programmbausteinen, die in anderen Programmiersprachen geschrieben sind, wird über sogenannte *User Exits* ermöglicht.

Adaptionsschicht Eine Anpassung der Funktionalität und des Datenmodells des ERP-Systems an die jeweils abgebildeten betrieblichen Prozesse und Datenstrukturen ist Teil

der Adaptionsschicht. Zusätzlich sind in dieser Schicht typischerweise Integrationselemente vertreten. Sie erlauben über Workflows eine Abbildung von Prozessketten, dessen einzelne Prozesse auch unterschiedliche Informationssysteme nutzen können.

Präsentationsschicht Als oberste Schicht bildet die Präsentationsschicht die Schnittstelle zum Benutzer. Sie kann die typischen Elemente, wie eine grafische Benutzeroberfläche (*GUI*) oder Internet-Browser, umfassen. Die von der Schnittstelle abgebildeten Funktionalitäten können dabei variieren. So bietet z.B. für gewöhnlich ein Internet-Browser weniger Funktionalität als eine grafische Benutzeroberfläche.

2.5 Open-Source in unternehmensweiten Anwendungen

Der enorme Vormarsch von OpenSource-basierenden Projekten und Anwendungen in den vergangenen Jahren mag verschiedene Gründe haben. Für Unternehmen sind jedoch bei der Entscheidungsfindung zwischen kommerzieller, proprietärer und OpenSource-Lösungen bestimmte Kriterien besonders entscheidend. Diese leiten sich im Falle von ERP-Lösungen direkt aus den, in Kapitel 2.3 genannten Anforderungen an eine ERP-Software ab. Insbesondere die Eigenschaft der Zukunftssicherheit einer ERP-Lösung spricht für eine Verwendung von Software auf Basis einer OpenSource-Lizenz [2].

Problematisch ist das Auslaufen und letztendliche Versiegen der Produktpflege seitens des Herstellers einer ERP-Lösung, sei es durch Fusionen, Konkurs oder anderer Einflüsse. Es führt nicht zwingend zu einem Konflikt bei Änderungen oder Evolutionen der eigenen Prozesse, stellt jedoch einen Risikofaktor dar, welcher die Entscheidung für eine OpenSource-Lösung bestärkt. Der offene Quellcode und die weltweite Gruppierung der unterstützenden Programmierer bewirkt hier eine Entkopplung von Abhängigkeiten im Zuge von Produktevolutionen und Änderungen. Bei einem Versiegen der Gruppierungen der freien Programmierer bleibt einem Unternehmen hier weiterhin noch die Möglichkeit, den offenen Code durch finanzierte Programmierer erweitern und ändern zu lassen. Ein gänzliches Verschwinden der Erweiterungsmöglichkeiten ist bei solch einer Form der ERP-Lösung nicht möglich.

Im Folgenden sollen einige Kriterien zur Entscheidungsfindung bei der Auswahl von kommerzieller oder OpenSource-Software genannt werden [GW05]:

- Transparency: Die Beurteilung der Softwarequalität, ohne eine tiefere Analyse und Bewertung des zugrunde liegenden Programmcodes, bleibt auf die nach außen sichtbaren Komponenten der Software begrenzt. Typischerweise werden bei kommerzieller Unternehmenssoftware Programmfehler auch nicht in den öffentlichen Raum, wie es z.B. Internetforen sind, getragen, um Hilfe zu ersuchen. Dies

[2] Beispiel einer, an wirtschaftliche Interessen angepassten, OpenSource-Lizenz ist die Mozilla Public Licence [Moz00]

resultiert in einem begrenzten Wissenshorizont, was die Fehler und Schwächen des Programms betreffen. Die Möglichkeiten zur Evaluierung der Softwarequalität ist bei OpenSource-Software somit ungleich zahlreicher.

- The Sales Process: Zur Vermarktung kommerzieller Unternehmenssoftware stehen überwiegend professionelle und vom Hersteller bezahlte Verkäufer zur Verfügung, welche auf Fragen bezüglich des Produktes Antworten geben können. Bei OpenSource-Software ist dies nicht der Fall - hier sind die Fragen an die Gemeinschaft der Programmierer bzw. Anwender zu richten. Eine Bewertung der gegebenen Antworten obliegt in beiden Fällen dem Fragesteller, d.h. einem Selbst. Einen Anspruch auf unbedingte Glaubwürdigkeit haben beide Bezugsquellen der Antworten nicht.

- Flexibility: Individuelle Änderungswünsche, z.B. das Hinzufügen einer Funktionalität, sind bei kommerziellen Softwareprodukten an den Hersteller zu richten. Die Entscheidung, wann dieser die gewünschte Änderung vornimmt, obliegt ihm allein. Eine erhöhte Prioritätsstufe lässt sich hier oft nur durch eine gesteigerte Anfragezahl anderer Nutzer hinsichtlich derselben Änderung oder durch finanzielle Mittel erreichen. Die Zeitdauer bis eine Änderung vorgenommen wird, kann so mitunter Jahre in Anspruch nehmen. Bei OpenSource-Software gibt es diese Hindernisse nicht. Es bedarf nur dem Wissen, der Zeit und dem Aufwand um eine Änderung selbstständig durchzuführen bzw. durch die Gemeinde durchführen zu lassen.

- Risk of Quality: Die Entwicklung neuer Funktionalitäten innerhalb einer kommerziellen Software geschieht für den Benutzer bzw. dem Unternehmen im Verborgenen. Die Beurteilung der korrekten Funktionsweise der neuen Funktion beruht auf Vertrauen gegenüber dem Hersteller und wird nur mittels Trial & Fail, d.h. dem austesten der neuen Funktion, bestärkt. OpenSource-Software bietet hier den Vorteil der öffentlichen Diskussion über eventuelle Nebeneffekte der neuen Funktion oder der korrekten Implementierung der Funktion selber. Das Risiko eines unentdeckten Nebeneffektes oder Fehlers wird dadurch verringert.

- Risk of Productization: Ein Bereich, in welchem OpenSource gegenüber kommerzieller Software oft Schwächen aufzeigt, ist der des Produktrahmen. Damit sind Elemente gemeint sind, welche das Softwarepaket abrunden, wie z.B. ausführliche Dokumentationen, administrative Unterstützungstools oder Tools die das Installieren und Ausrollen der Software im Unternehmen erleichtern.

- Risk of Failure: Das Risiko des Scheiterns eines Softwareherstellers besteht ebenso für OpenSource-Projekte, bei denen die Gemeinde der Programmierer versiegt. Dieser Verlust wiegt bei jedoch kommerzieller Software schwerer, da hier nur bei vorher getätigtem Abschluss entsprechender Verträge die Herausgabe des Quellcode erwirkt werden kann. OpenSource-Projekte bieten diese Möglichkeit um ihrer Selbst willen und reduzieren damit das Risiko für die Unternehmen im Falle eines Scheiterns der Softwarehersteller.

- Risk of Takeover: Neben dem Risiko des Scheiterns des Softwareherstellers gibt es noch die Gefahr der Übernahme des Herstellers durch einen Konkurrenten, welcher ein vergleichbares Produkt anbietet. Auflösung des Supports des ursprünglichen Herstellers und erzwungene Migrationen zum Konkurrenzprodukt sind durchaus übliche Szenarien und erhöhen das Risiko für das eigene Unternehmen. OpenSource-Projekte sind gegen diese Art von Übernahmen weitgehend geschützt.

- Support: Der Support der Unternehmenssoftware ist ein wichtiges Entscheidungskriterium, welches durch kommerzielle Anbieter überwiegend durch spezialisierte Supportteams gewährleistet wird. Sie gewährleisten eine Unterstützung auf hohem Level, welche die jährlich anfallenden Gebühren für die Unternehmen rechtfertigen. OpenSource-Projekte bieten im Gegensatz dazu größtenteils nur Internet-Foren und andere Formen der schriftlichen Veröffentlichung von Fragen. Die Qualität und Wartezeit der Antwort ist hier variabel, d.h. eine Instanz bei dringenden Fragen in Notsituationen gibt es nicht. An diese Punkt setzen einige Firmen an, die ihr Wissen über die OpenSource-Software in Form eines Supportdienstes anbieten. Sie füllen die Lücke, welche OpenSource hin zu kommerzieller Software im Bereich des Supports aufweist.

Die Verwendung eines OpenSource-Produktes bringt also verschiedene Vorteile für Unternehmen. Besonders die genannten Punkte *The Sales Process*, *Risk of Failure/Takeover* und *Support* bilden für Unternehmen häufig die stärksten Argumente bei der Entscheidungsfindung. Sie bilden die Kriterien, welche in hohem Maße eine reibungslose und dauerhafte Nutzung des Produkts sicherstellen. Trotz der aufgezeigten Vorteile von OpenSource-Produkten existieren parallel dazu auch Nachteile, die sich besonders im Bereich des Support negativ auswirken können. Nichtsdestotrotz ist die Entscheidungsfindung nicht generell zu Gunsten von OpenSource-Lösungen zu treffen, sondern unterliegt immer auch den jeweiligen Anforderungen der Unternehmen.

3 Wandlungsfähigkeit

Im vorangegangenem Kapitel wurden die Kernaufgaben eines ERB-Systems näher erläutert und spezifiziert. Sie unterstützen ein Unternehmen bei der Koordinierung und Verbesserung ihrer Arbeitsprozesse. Diese Arbeitsprozesse sind jedoch nicht statisch. Das Aufkommen der globalen Marktwirtschaft hat nicht zuletzt auch den Änderungsdruck auf Unternehmen und damit auch ihrer Arbeitsprozesse erhöht. Im Zuge dieser Veränderungen des Marktes entwickelten sich Trends, welche in [Wil06] unter anderem wie folgt benannt werden:

- Anhaltender Preis- und Wettbewerbsdruck
- Erhöhte Anforderung an Lieferzeiten und Termintreue
- Steigende Nachfrage kundenindividueller Produkte
- Sinkende Produktlebenszyklen und Time-to-Market-Zeiten
- Hoher Innovationsanspruch in der Produktentwicklung
- Wechsel zu System- und Modullieferanten sowie Globalisierung von Produkten und Beschaffung

Die Anpassung an diese Trends durch die sich ändernden Unternehmensbereiche, seien dies nun z.B. geänderte Geschäftsprozesse oder Zuliefererketten, wird damit zu einem wesentlichen Erfolgskriterium eines Unternehmen. In diesem Kapitel soll nun der Begriff der *Wandlungsfähigkeit* näher erläutert werden und die daraus resultierende Bedeutung für ERP-Systeme betrachtet werden.

3.1 Begriffsklärung

Der Begriff der Wandlungsfähigkeit wird in vielen Quellen unterschiedlich definiert. Aus diesem Grund soll an dieser Stelle eine kurze Erläuterung der in dieser Arbeit verwendeten Begriffsdefinition, angelehnt an [AG05], folgen:

> Als Wandlungsfähigkeit bezeichnet man die Fähigkeit, sich bzw. etwas zu verändern, um sich an Veränderungen anzupassen. Im Kontext von IT-Systemen konkretisiert sich dies auf eine Veränderung mit dem Ziel, sich an wechselnde Umgebungen oder Gegebenheiten anzupassen. Daraus resultiert die Anforderung an ein Systems, Änderungen idealerweise selbst zu erkennen, um sich

anschließend mittels passender Lösungsalternativen an die Änderung anpassen zu können.

Der Umfang des nötigen Aufwands, um ein System wandlungsfähig zu gestalten, hängt damit von der Systemumgebung ab und ist so individuell für jeden Systemtyp. Im Folgenden soll nun ein Einblick in den betrachteten Systemtyp dieser Arbeit, d.h. der Thematik wandlungsfähiger ERP-Systeme, gegeben werden.

3.2 Wandlungsfähige ERP-Systeme

Am Anfang dieses Kapitels fanden Trends Erwähnung, welche heutige Unternehmen dazu drängen, ihre Systeme und hier im speziellen ihre IT-Systeme wandlungsfähig zu gestalten. Ziel ist ein unternehmensweites IT-System, welches sich an Änderungen innerhalb seiner Umgebung, den sogenannten *Umweltturbulenzen*, anpassen kann. Umweltturbulenzen, wie sich ändernde Geschäftsprozesse, finden häufig im laufenden Systembetrieb statt. Die Anpassungsfähigkeit bestehender ERP-Systeme ist hier meist nur ungenügend, d.h. die veränderten Geschäftsprozesse lassen sich nur unvollständig oder ineffizient abbilden. Ziel der Entwicklung bzw. Verbesserung wandlungsfähiger ERP-Systeme ist eine eigenständige, schnelle und effiziente Anpassung an veränderte Anforderungen.

3.2.1 Kriterien der Wandlungsfähigkeit

Für eine Evaluierung der Wandlungsfähigkeit von Informationssystemen werden geeignete Kriterien benötigt, welche in diesem Abschnitt vorgestellt werden.

Die Kriterien der Wandlungsfähigkeit finden sich auf zwei Dimensionen wieder. Die Dimension der technischen Ausprägung definiert die Kriterien zur Bestimmung des Potentials eines Softwaresystems, sich wandlungsfähig zu verhalten. Dem gegenüber steht die Dimension der unternehmensindividuellen Ausprägung. Ihre Kriterien bestimmen wie wandlungsfähig eine unternehmensindividuelle Informationsarchitektur ist.

Die Kriterien der Wandlungsfähigkeit werden im Folgenden genannt [GLA06][ALG06]:

- ***Skalierbarkeit***

  Die Skalierbarkeit eines Informationssystem bezeichnet die quantitative Kapazität eines Systems. Dieser Indikator fordert die effiziente Anpassung der Kapazitätsmenge sowohl nach oben als auch nach unten. Die Techniken zur Erfüllung dieses Kriteriums können dabei sowohl Hardware- als auch Software-basierend sein. Eine Hardwarelösung könnte hier z.B. das automatische Entfernen, Freigeben oder Hinzufügen von Ressourcen beinhalten, sobald der problemlose Betrieb des Systems gefährdet ist. Eine skalierbare Softwarelösung muss selbst den Anforderungen entsprechen und mit ihnen wachsen können. Dies kann z.B. die Bildung von weiteren Systemgruppen bei steigender Anwenderzahl sein.

- ***Modularität***

  Als Modularität bezeichnet man die Aufteilung eines Gesamtsystems in kleine Subsysteme, den Modulen. Ein Modul wird dabei definiert als ein Konstrukt aus einem Modulrumpf und einer Modulschnittstelle. Der Modulrumpf implementiert dabei Definitionen, welche durch die Modulschnittstelle gegeben werden. Die Schnittstelle spezifiziert damit die Eigenschaften eines Moduls, welche für seine Modulumgebung von Bedeutung sind. Die Möglichkeit der damit ermöglichten Wiederverwendung und Kombination von Modulen kann eine schnelle und effiziente Anpassung des Systems unterstützen und damit zur Erhöhung der Wandlungsfähigkeit beitragen.

- ***Verfügbarkeit***

  Die Verfügbarkeit ist das Kriterium der uneingeschränkten zeitlichen und räumlichen Zugriffsmöglichkeit auf das System. Dies bedeutet, dass ein System von jedem Ort aus und zu jeder Zeit für jeden verfügbar sein muss.

- ***Unabhängigkeit***

  Ein als unabhängig definiertes System agiert von anderen Systemen getrennt. Der Ausfall des Systems darf des Weiteren auch keinen Einfluss auf andere Systeme haben. Dies bedeutet, dass ein System unabhängig von der Umgebung, wie dem Betriebssystem oder der System-Hardware, agieren können muss. Daneben zählen dazu auch alle wichtigen Subsysteme. Sie müssen vor Ausfällen bewahrt werden, wie z.B. durch geeignete Backup- und Redundanzstrategien.

- ***Interoperabilität***

  Als Interoperabilität bezeichnet man die Fähigkeit eines Systems mit anderen Systemen zu interagieren. Diese muss dabei unabhängig von der zugrundeliegenden Architektur, d.h. der Hardware, dem Betriebssystem, der Netzwerktopologie oder der Implementierung, sein. Durch diese Charakteristika lässt sich der Indikator dieses Kriteriums auch als Maß der Systemkompatibilität verstehen. Die Entwicklung und Nutzung von Standards kann dabei helfen den Grad der erreichten Systeminteroperabilität zu erhöhen. Eine steigende Interoperabilität erhöht die Integrationsfähigkeit des Systems und unterstützt damit z.B. die Verteilung eines Geschäftsprozesses auf unterschiedliche Anwendungssysteme. Diese Eigenschaft der steigenden Integrationsfähigkeit räumt diesem Kriterium eine zentrale Rolle unter den Kriterien der Wandlungsfähigkeit ein. Sie stärkt die Kommunikationsfähigkeit über System- und damit auch Unternehmensgrenzen hinweg und kann sich so auch auf Eigenschaften der *Verfügbarkeit* und *Unabhängigkeit* auswirken und diese positiv beeinflussen.

- ***Selbstorganisation***

  Die Selbstorganisation von Systemen bezeichnet deren Fähigkeit ihre Systemstruktur mittels selbstregulierender Mechanismen selbst zu ermitteln, um mit diesen Informationen den problemlosen Betrieb zu gewährleisten. Dabei sammelt jedes der

Subsysteme Informationen über ihre Umwelt und ihrer eigenen Wechselwirkungen mit ihr. Diese Informationen werden genutzt, um ein Modell zu erstellen, nachdem das Subsystem dann handeln kann. Umweltveränderungen eines dieser Subsysteme führt zu einer Änderung dieses Modells und damit des Verhaltens dieses Subsystems selbst. Das Kriterium der Selbstorganisation von Informationssystemarchitekturen verlangt vom System demnach die Fähigkeit, seine innere Struktur bzw. Architektur selbst ganz oder teilweise zu bestimmen.

- ***Selbstähnlichkeit***

  Als Selbstähnlichkeit wird die Eigenschaft eines Systems bezeichnet, auf unterschiedlichen Abstraktionsebenen immer wieder dieselben Muster vorzufinden. Diese Skaleninvarianz hilft bei der Komplexitätsreduzierung eines Systems und unterstützt damit auch die Selbstorganisation. Daneben bietet sie noch weitere Vorteile, wie die leichtere Erlernbarkeit. So ist z.B. ein selbstähnlicher Quellcode schneller zu analysieren als Code mit ständig wechselnden Patterns oder Stilen.

- ***Automatisierungsgrad:*** Neben dem Customizing durch eine Parametrisierung ist hiermit auch die Anpassung des Systems durch Änderung seiner Implementierung gemeint. Dadurch wirkt sich dieser Punkt insbesondere bei quelloffenen Systemen aus.

- ***Systemwissen:*** Änderungen an einem System erfordern Wissen über die jeweiligen Eigenheiten und Details dieses Systems. Je besser das vorhandene und je geringer das nötige Wissen ist, desto effizienter ist die Umsetzung und daher auch die Wandlungsfähigkeit des Systems.

3.2.2 Wandlungsfähige Architekturmodelle

Im Zuge der Entwicklung der in diesem Kapitel genannten Kriterien der Wandlungsfähigkeit eines ERP-Systems, wurde ein Referenzmodell für wandlungsfähige Systemarchitekturen entwickelt (Abb. 3.1). Da die einzelnen Schichten dieses Modells den kontextuellen Rahmen für die Kritieren der technischen Ausprägung bildet, soll dieses Modell an dieser Stelle kurz vorgestellt werden.

- ***Infrastrukturschicht***

  Die Infrastruktur bildet die unterste Schicht des Modells und beinhaltet Informationen über die Verteilung des Systems sowie der verwendeten Topologie

- ***Datenschicht***

  Das Datenbankmanagementsystem mitsamt den Datenbanken wird in der Datenschicht zusammengefasst.

- ***Applikationsschicht***

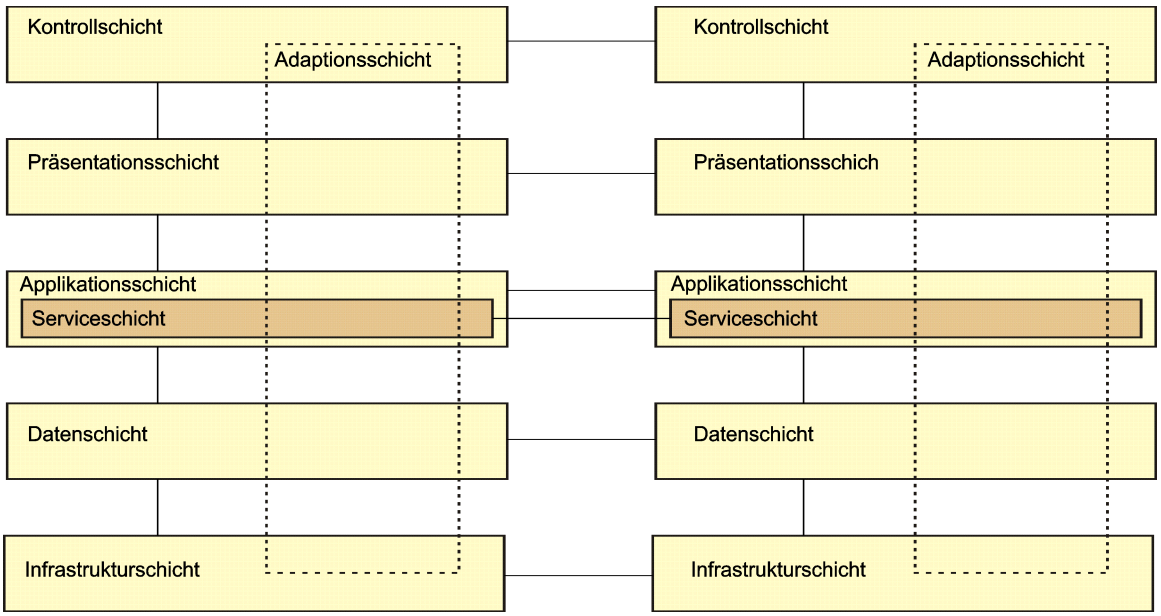

Abbildung 3.1: Architekturmodell für wandlungsfähige ERP-Systeme (nach [GLA06])

Die Applikationsschicht hilft bei der Strukturierung der einzelnen Systemkomponenten. Es beinhaltet den sogenannten *Servicelayer*, welcher Ressourcen und Transaktionen verwaltet.

- ***Präsentationsschicht***

  Die Präsentationsschicht dient der Interaktion zwischen Benutzer und dem System. Es stellt damit das Benutzerinterface des Systems dar.

- ***Kontrollschicht***

  Diese Schicht dient der Modellierung von Geschäftsprozessen. Änderung an dem Modell werden von dieser Schicht an alle ihre Unterschichten kommuniziert und somit konsistent gehalten.

Durch alle diese Schichten zieht sich die vertikale Adaptionsschicht. Sie beinhaltet alle wandlungsfähigen Elemente der jeweiligen Schicht und bildet damit ein Maß zur Evaluierung des erreichten Grades der Wandlungsfähigkeit aller Schichten, d.h. des gesamten ERP-Systems.

3.2.3 Ermittlung der Wandlungsfähigkeit

Mit Hilfe der im vorangegangenem Abschnitt vorgestellten Kriterien kann nun ein aussagekräftiger Wert für den Grad der Wandlungsfähigkeit ermittelt werden. Das Vorgehensmodell hierfür beinhaltet ein stufenweises Vorgehen innerhalb einer Hierarchie von Bewertungen. Die Reihenfolge erfolgt dabei bottom-up, d.h. man beginnt bei der untersten Hierarchiestufe und folgt ihr schrittweise nach oben bis zum Erreichen der Wurzel.

Diese Hierarchie soll im Folgenden kurz erläutert werden. Auf oberster Stufe findet in ihr zuallererst eine Aufteilung in zwei Typen der Wandlungsfähigkeit statt. Die Typen und ihr weiterer hierarchischer Aufbau werden nun kurz erläutert werden.

1. ***Technische Wandlungsfähigkeit***

   Diese Art der Wandlungsfähigkeit hat als Fokus das eigentliche Anwendungssystem an sich. Es gliedert sich in jedes der Schichten des wandlungsfähigen Architekturmodells aus Abbildung 3.1 und betrachtet dabei pro Schicht jeweils die genannten Kriterien aus 3.2.1. Jedem dieser Kriterien ist dabei für die jeweilige Modellschicht ein passender Fragenkatalog zugeordnet, mit welchem ein Punktwert für die Wandlungsfähigkeit in dieser Schicht und unter dem jeweiligen Kriterium ermittelt werden kann. Aus den Ergebnissen der Fragebögen der einzelnen Kriterien lässt sich anschließend ein Wert zur Wandlungsfähigkeit für die Gesamtschicht ermitteln. Je höher der erreichte Wert eines Kriteriums oder einer Schicht ist, desto mehr Komponenten liegen innerhalb der vertikalen Adaptionsschicht des Architekturmodells und umso wandlungsfähiger ist damit das Gesamtsystem.

2. ***Geschäftsspezifische Wandlungsfähigkeit***

   Die zweite Ebene der Betrachtung ist die der unternehmensindividuellen Wandlungsfähigkeit, d.h. der Beurteilung, inwieweit das System Geschäftsprozesse zur Laufzeit des Systems umsetzen und ändern kann. Zu diesem Zweck gibt es einen Fragebogen, welcher sich auf die 4 Reorganisationsansätze von Gronau in [Gro06] bezieht:

   - ***Subsystembildung:*** Zuordnung oder Aufspaltung der Aufgabenbearbeitung zu einzelnen Untersystemen
   - ***Prozessorientierung:*** Ausrichtung und Anpassung der Prozesse entlang der Wertschöpfungskette
   - ***Kontinuierlicher Verbesserungsprozess:*** Kontinuierliche Betrachtung des Unternehmens über einen Zeitabschnitt hinweg zur Bildung von Subsystemen und Ausrichtung der Prozesse an der Wertschöpfungskette
   - ***Auflösung der Unternehmensgrenzen:*** Ausweitung der Wertschöpfungskette über die eigenen Systemgrenzen hinaus

   Jedes dieser Reorganisationansätze beinhaltet jeweils eine Reihe von Teilprozessen. Sie werden in Form von Fragen formuliert und erlauben die Beantwortung über die beiden Kritieren *Automatisierungsgrad* und *Systemwissen*. Diese werden mittels einer fünf-stufigen Skala[1] für den betrachteten Teilprozess bewertet. Alle Bewertungen zusammen geben Aufschluss darüber, inwieweit der jeweilige Reorganisationsansatz im betrachteten System realisierbar ist und dienen so der Ermittlung der geschäftsspezifischen Wandlungsfähigkeit.

[1] 0 = Keine Anpassung möglich, 1 = Anpassung durch Add-Ons, 2 = Anpassung durch Modifikationen, 3 = Anpassung durch Parametrisierung, 4 = autom. Selbstkonfiguration

Die einzelnen Elemente dieser Hierarchie können einen verschieden grossen Einfluss auf die reale Wandlungsfähigkeit ausüben. Diesem wird man gerecht, in dem jedes Element zusätzlich eine Gewichtung bekommt. Sie bestimmt bei der späteren Berechnung, wie stark der ermittelte Wert zur Wandlungsfähigkeit eines Elements, z.B. eines Kriteriums oder einer Schicht, auf das Endergebnis Einfluss nimmt. Die Gewichtung wird hierbei mit dem *Analytical Hierarchy Process* von Saaty [SV00] ermittelt. Auf eine tiefergehende Betrachtung des mathematischen Modells zur Berechnung der Wandlungsfähigkeit wird an dieser Stelle aber verzichtet.

Abbildung 3.2 verdeutlicht die dargelegte Hierarchie und zeigt die Beziehung der Ebenen untereinander. Zu erkennen sind ebenfalls die Gewichte für jede der Ebenen. Alle Elemente derselben Ebene bilden dabei zusammen das Gewicht 1.0. Elemente, welche auf den Fragenkatalog zurückgreifen, sind durch eine Verbindungslinie zu diesem gekennzeichnet.

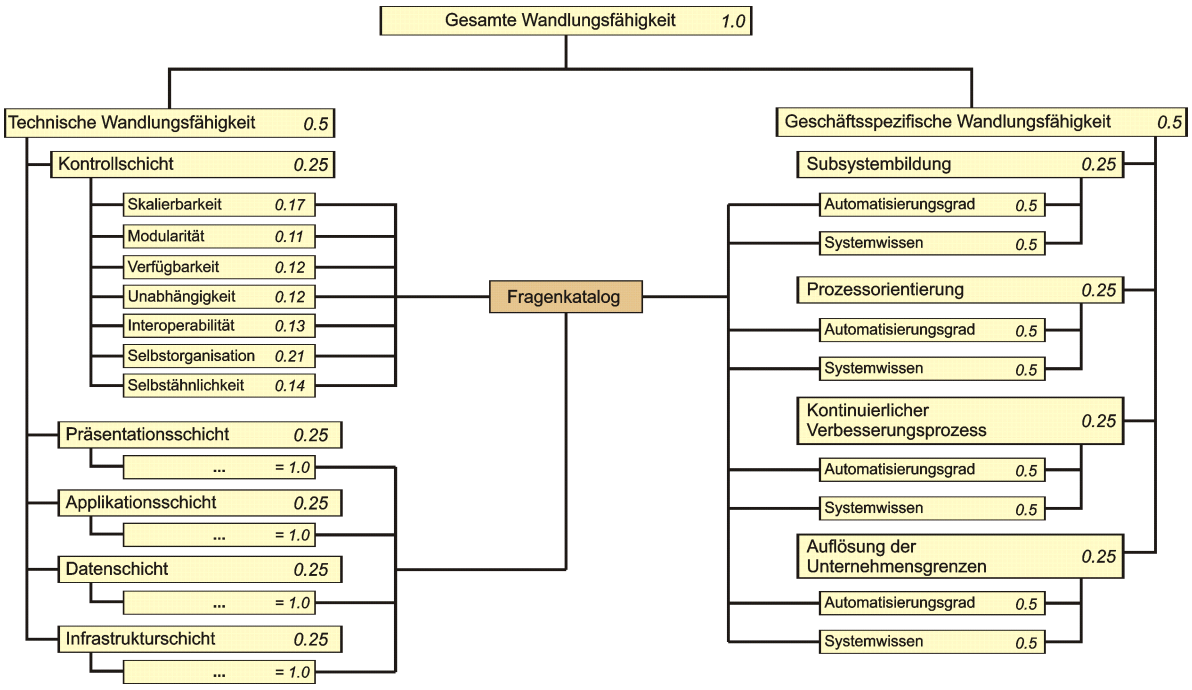

Abbildung 3.2: Hierarchie zur Ermittlung der Wandlungsfähigkeit eines ERP-Systems

Die Fragenkataloge jedes Kriteriums sowie das gesamte Vorgebensmodell wurden innerhalb des Forschungsprojektes *CHANGE*[2] am *Lehrstuhl für Wirtschaftsinformatik und Electronic Government* der Universität Potsdam[3] entwickelt. Sie finden innerhalb eines Werkzeuges zur Evaluierung der Wandlungsfähigkeit eines ERP-Systems Anwendung - dem *Adaptability Analyzer*[4]. Diese Java-Applikation wurde im Rahmen einer Diplomarbeit am genannten Lehrstuhl entwickelt und unterstützt den Benutzer bei der Beantwortung aller Fragen jedes Kriteriums. Aus den Antworten wird daraufhin das, im Anschluss

[2] http://www.change-projekt.de
[3] http://wi.uni-potsdam.de
[4] http://wi.uni-potsdam.de/projekte/ceronline.nsf?Open&ID=AEC3CD875292223AC125710000798BF3

näher beschriebene Bewertungsportfolio erstellt. Das Ergebnis der Bewertung des *Adaptability Analyzer* dient als Qualitätsmerkmal eines ERP-Systems und wird in Form einer Zertifizierung durch das *Center for Enterprise Research*[5] käuflich vertrieben. Aus diesem Grund ist der Fragenkatalog nicht öffentlich einzusehen und wird in den abschließenden Bewertungen (Siehe 7.1) nur Auszugsweise genannt werden. Des Weiteren dient der *Adaptability Analyzer* in späteren Kapiteln lediglich der Bewertung der Wandlungsfähigkeit von Compiere und bedarf damit an dieser Stelle selbst keiner detaillierteren Betrachtung.

3.2.4 Auswertung der Wandlungsfähigkeit

Im Verlaufe dieses Kapitels wurde das Vorgehen zur Evaluierung der Wandlungsfähigkeit von ERP-Systemen erläutert. Durch die Auswertung der Antworten auf die Fragen der einzelnen Hierarchieebenenen kann ein Wert sowohl für die technische- als auch die geschäftliche Wandlungsfähigkeit ermittelt werden. Diese beiden Werte dienen der Einordnung des Systems in ein Portfolio, in welchem sich die technische- und geschäftsspezifische Wandlungsfähigkeit gegenüberstehen (Abb. 3.3).

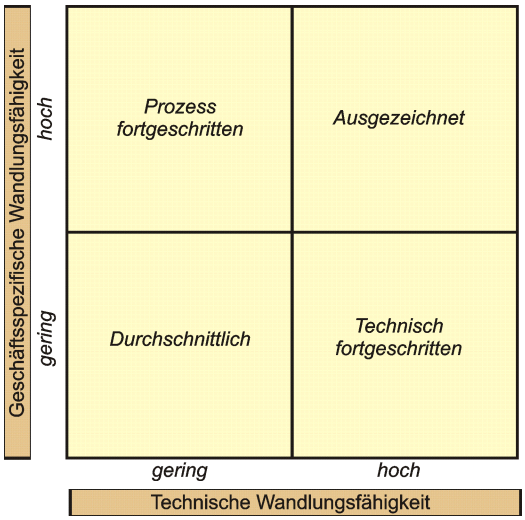

Abbildung 3.3: Ergebnisportfolio für ERP-Systeme bezüglich Wandlungsfähigkeit (nach [ALG06])

Dieses Portfolio ordnet die untersuchten Systeme in 4 Quadranten ein, wobei jedes der Quadranten unterschiedliche Aussagen zulässt. Der Quadrant „Durchschnittlich“ beinhaltet Systeme, welche sowohl in der technischen als auch in der geschäftsspezifischen Wandlungsfähigkeit eher geringer Ausprägung sind. Als „Technisch fortgeschritten“ wird der Quadrant betitelt, welcher die Systeme umfasst, die technisch sehr wandlungsfähig sind, jedoch nur eine mangelhafte Unterstützung von sich ändernden Geschäftsprozessen bieten. Ein weiterer Quadrant ist „Prozess fortgeschritten“. Ein System, welches in diesem

[5] http://www.erp-research.de

Quadranten liegt, erlaubt eine sehr gute Abbildung von Geschäftsprozessen im System, ohne jedoch über eine vergleichbare technische Wandlungsfähigkeit zu verfügen. Dies trifft oft bei der Betrachtung von spezialisierten oder standardisierten Geschäftsprozessen zu, auf die ein System zugeschnitten sein kann. Als „Ausgezeichnet“ ist der Quadrant bezeichnet, der eine hohe Ausprägung auf beiden Ebenen der Wandlungsfähigkeit bietet. Er ist der Bereich, in dem ein System im Sinne der Wandlungsfähigkeit idealerweise liegen soll.

Die Gegenüberstellung der beiden Ebenen der Wandlungsfähigkeit erlaubt eine Beurteilung des betrachteten Systems und auch einen objektiven Vergleich mit anderen Systemen. Die durch diese Auswertung gewonnenen Resultate können sowohl den Entwicklern als auch den Kunden dienen. Aus Entwicklersicht dienen sie dem Erkennen von Schwachstellen im System und dem Finden neuer Entwicklungsansätze. Der Kunde hat mit ihnen ebenfalls die Möglichkeit, Schwächen eines Systems zu erkennen. Er nutzt sie jedoch zur frühzeitigen Beurteilung eines Systems und die damit verbundene Entscheidungshilfe.

3.2.5 Bedeutung der Wandlungsfähigkeit für ERP-Systeme

Allgemein benötigt man für eine aussagekräftige Gegenüberstellung von Dingen ein Bewertungsmass, welches für alle Vergleichselemente dieselben Metriken und Methoden beinhaltet. Auf die Frage nach einer Bewertung der Wandlungsfähigkeit eines ERP-Systems haben dessen Entwickler in der Vergangenheit häufig subjektiv entschieden und damit die Nutzung einer Vergleichsmatrix erschwert [AKS05]. In diesem Kapitel wurde nun eine einheitliche Methodik zur Bewertung der Wandlungsfähigkeit eines ERP-Systems vorgestellt. Sie erlaubt anhand ihrer Ergebnise und deren Einordnung in das vorgestellte Portfolio eine Gegenüberstellung verschiedener ERP-Systeme. Mit ihr lassen sich für jedes der ERP-Systeme Schwachstellen bezüglich der Wandlungsfähigkeit lokalisieren.

Eine gesteigerte Wandlungsfähigkeit dient dabei zuallererst dem Kunden. Diesem wird durch ein wandlungsfähiges System erlaubt, seine Geschäftsprozesse zur Laufzeit des Systems anzupassen und im Idealfall das System selbst nach den eigenen Wünschen zu gestalten. Zum Grossteil wirkt dies jedoch entgegen den Interessen der Systementwickler. Wartungsverträge und der Verkauf von Erweiterungsmodulen bilden einen Hauptpfeiler des Tagesgeschäfts vieler Anbieter. Ein System mit stark ausgeprägter Wandlungsfähigkeit würde diese Geschäftsbereiche einschränken. Nichtsdestotrotz kommt dem Thema Wandlungsfähigkeit auch von Entwicklerseite ein reges Interesse zu. So bewerten bereits viele Entwickler die Flexibilität des ERP-Systems als entscheidenden Punkt für zukünftige Weiterentwicklungen [AKS05]. Nicht zuletzt, weil das Prädikat der hohen Wandlungsfähigkeit die Attraktivität eines ERP-System für den Kunden steigert. Wie so oft wird sich hier aber ein Konsens der Interessen bilden und die Wandlungsfähigkeit auch in Zukunft ein Impulsgeber für die Weiterentwicklung von ERP-Systemen bleiben.

4 Compiere

compiere [kompjere] ital. für erreichen, erfüllen, abliefern

Das seit 1999 entwickelte OpenSource-Projekt Compiere®[1] stellt eine ERP-Lösung mit dem Schwerpunkt auf kleine und mittelständische Unternehmen dar. Der Name Compiere ist ein registriertes Markenzeichen der ComPiere Inc., wird im weiteren Verlauf dieser Arbeit aber nicht weiterführend als solches gekennzeichnet. Als Produkt mit dem funktionalen Kern im Bereich der Ressourcenplanung tritt Compiere in Konkurrenz zu kommerziellen Produkten von Microsoft, SAP und anderen. Die in 2.5 genannten Vorteile von OpenSource ermöglichten dem Produkt, sich auf dem Markt zu behaupten. Wirtschaftlich trägt sich dieses Projekt, welches zum Zeitpunkt der Ausarbeitung über 55 Mitarbeiter in der Entwicklung beschäftigt, durch die Ausrichtung des Geschäftmodells rund um das Produkt. So bilden Support in Form von Wartungsverträgen, Training und das Bereitstellen von Dokumentationen das Kerngeschäft der Compiere Inc.

Das in diesem Kapitel Betrachtung findende Compiere entspricht dem *Release 2.5.3a*.

4.1 Funktionalitäten

Anders als bei einigen anderen Entwicklungen, liegt die funktionale Hauptausrichtung im Design von Compiere in der Abbildung von Geschäftsprozessen. Eine zentrale Aufgabe kommt dabei dem *Ressource-Management* zu. Dieser verknüpft die abgebildeten Geschäftsprozesse mit den Ressourcen, wie Material, Kapital oder Personal und erlaubt damit die Integration verschiedenster Ressourceneigenschaften in die abgebildeten Geschäftsprozesse. Ein vorstellbares Anwendungsbeispiel wäre hier die Anpassung des Geschäftprozesses *Verkauf* an die vorhandenen Ressourcen. Das Unterbreiten eines Angebotes ist nur sinnvoll, wenn die nötigen Ressourcen vorhanden bzw. lieferbar sind. In Abbildung 4.1 sind diese Relationen zwischen den Geschäftprozessen und den Ressourcen abgebildet.

Im Folgenden soll nun noch eingehender auf die von Compiere implementierten Funktionalitäten und abgebildeten Geschäftsprozesse eingegangen werden. [Com]

4.1.1 Quote-to-Cash

Quote-to-Cash bezeichnet den Geschäftsprozess *Verkauf* und beinhaltet die folgenden Elemente:

[1] http://www.compiere.org - ComPiere Inc., Portland, USA

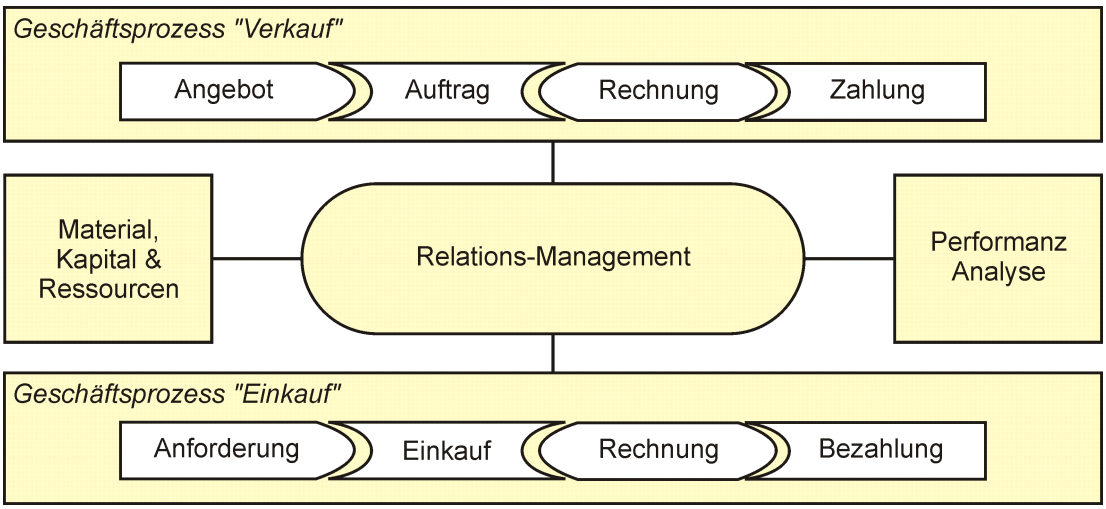

Abbildung 4.1: Basisfunktionalitäten von Compiere (Nach [Com])

- *Angebot*

  Erstellung und Druck eines Angebotes für potentielle bzw. reale Kunden. Dieses basiert auf allgemeinen oder kundenbezogenen Preislisten und kann bindend wirken, d.h. Ressourcen werden für dieses Angebot reserviert. Angebote können jederzeit geändert und zu Aufträgen konvertiert werden.

- *Auftrag*

  Das Management der getätigten Bestellungen umfasst die Auslieferung und die Erstellung von Rechnungen. Ein Auftrag kann dabei verschiedene Aktionen implizieren, welche über den Auftragtypus geregelt werden. Hierbei werden die folgenden Aufträge unterschieden:

  - *Standard:* Erstellung eines Auftrages mit anschließender Reservierung der benötigten Ressourcen. Die Rechnung wird sofort oder nach erfolgter Lieferung erstellt.
  - *POS-Auftrag:* Sofortiges Einleiten aller Auftragselemente, d.h. Erstellen des Auftrages, Auslieferung, Drucken der Rechnung und Empfang der Bezahlung.
  - *Kredit-Auftrag:* Erstellung des Auftrages mit sofort erfolgender Lieferung und Rechnungsstellung, aber mit optionaler Bezahlung. Diese kann hier zu einem späteren Zeitpunkt erfolgen.
  - *Warenhaus-Auftrag:* Sofortige Erstellung und Lieferung des Auftrages. Das Ausstellen der Rechnung und die Bezahlung können zu einem späteren Zeitpunkt erfolgen.
  - *Vorkassen-Auftrag:* Nach Erstellen des Auftrages und der Rechnung wird auf die eingehende Bezahlung gewartet, bevor die Lieferung eingeleitet wird.

  - *Reklamation (RMA):* Umfasst das Empfangen von schon gesandten Artikeln im Zuge einer Reklamation.

- *Lieferung*

  Basierend auf dem Auftrag werden eine oder mehrere Lieferungen eingeleitet, welche sofort oder automatisch bei Vorhandensein der benötigten Ressourcen getätigt werden können.

- *Rechnung*

  Rechnungen werden entweder sofort nach der Auftragsgenerierung oder nach erfolgter Lieferung erstellt. Dabei können auch Verzögerungen erzwungen werden, wie sie zum Beispiel bei monatlichen Sammelrechnungen nötig sind.

- *Bezahlung & Quittungen*

  Die automatische Generierung von Quittungen wird über sogenannte *Payment-rules* gewährleistet, wobei diese eine Bezahlung über verschiedene Arten, wie Kreditkarte oder Scheck, erlauben.

Die genannten Elemente bilden den gesamten Prozess des Verkaufes ab und sind stark in das Supply-Chain-Management (Abschnitt 4.1.5) und dem Customer-Relationship-Management (Abschnitt 4.1.3) integriert. Dies erhöht die Anpassbarkeit der Angebote an die verfügbaren Ressourcen und einen genaueren Zuschnitt an den jeweiligen Kunden.

4.1.2 Requisition-to-Pay

Die Befriedigung der Anforderungen an vorhandene Ressourcen zum laufenden Betrieb ist ein essentieller Bestandteil eines ERP-Systems. Der Geschäftsprozess *Requisition-to-Pay* dient der Akquirierung von Ressourcen als Unterstützung zur Erfüllung dieser Anforderungen. Er deckt den Ablauf von der Requirierung bis zu dessen Bezahlung ab. Dabei wird basierend auf der Anforderung eine Einkaufsbestellung generiert, welcher eine dazugehörige Rechnung und Bezahlung folgt. Aufgrund der Verknüpfung zur Lieferantenkette und dem Lieferanten als Kunden, findet in diesem Prozess eine starke Integration des Supply-Chain-Management und des Customer-Realtionship-Management statt.

Auf die Elemente dieses Geschäftsprozesses soll an dieser Stelle nicht näher eingegangen werden, da diese analog zu dem Prozess *Quote-to-Cash* stehen.

4.1.3 Customer-Relations-Management (CRM)

Das in Kapitel 2.1 erläuterte Konzept des *Customer-Relations-Management* dient der Verbesserung des Kundenkontaktes durch die Bereitstellung eine möglichst hohen Vielzahl an Kundeninformationen. Anders als die typischen ERP-Systeme ist das CRM nicht als gesondertes Modul in Compiere implementiert, sondern ist selbst integraler Bestandteil der Business Prozesse. Der Grund dafür liegt wohl auch in dem Umstand, dass das

CRM in Compiere sehr rudimentär ausgebildet ist und ein gesondertes Modul überflüssig werden lässt.

So bietet Compiere beispielsweise keine Kundenservice-Center Funktionalitäten oder eine Kundenwertanalyse wie sie in Abschnitt 2.1 dargelegt wurden. Nichtsdestotrotz gereicht der Integrationsumfang zur Verbesserung des Kundenkontaktes und erreicht damit ein wesentliches Ziel des CRM.

4.1.4 Partner-Relations-Management

Eine Erweiterung der CRM-Funktionalitäten bildet das sogenannte *Partner-Relations-Management* von Compiere. Es erlaubt die Nutzung von CRM-Funktionalitäten über die Grenzen von Compiereklienten und -firmen hinweg. Es verbindet verschiedene Systeme und erlaubt daneben den Zugriff über Webschnittstellen für Partner, welche nicht dem Verbund angehören. Die Funktionen sind wie folgt zusammenzufassen:

- *Server-übergreifendes Relationship-Management*

  Für nichtverbundene Partner bietet das Partner-Relations-Management eine Webschnittstelle zum Informationszugriff an. Systeme, welche dem Verbund angehören, kommunizieren untereinander mittels Anfragen, den *Requests*. Neben der Verbesserung der direkten Interaktion der Partner untereinander, wie das Einsehen und Begleichen von Rechnungen, liegt die für wachsende Unternehmen wichtigste Funktionalität in dem Aufteilen und Auslagern von Aufgabenbereichen.

  Dabei werden verschiedene Compiere-Systeme in einen logischen Verbund gebracht, wobei einigen Führungsrollen in bestimmten Aufgabenbereichen zugeteilt werden. Verschiedene Aufgaben können so ausgelagert und von privilegierten Systemen ausgeführt werden. Denkbar wäre hier zum Beispiel das Auslagern der Versandfunktionalitäten auf ein System, welches geographisch in einer Versandabteilung steht. Diese führenden Systeme können von allen anderen Systemen überwacht werden, um so den aktuellen Stand einer Aufgabe oder deren Ergebnis dem Verbund zur Verfügung zu stellen.

- *Geteilte Dienste*

  Dienstanbieter, wie zum Beispiel der Helpdesk oder die Lagerverwaltung, erhalten Zugriff auf Informationen, die für den jeweiligen Dienst notwendig sind. Diese Informationen können dabei auch über Organisations- und Systemgrenzen hinausgehen. Durch diese erweiterte Informationsdichte können die Dienste effizienter gestaltet werden.

- *Zentralisierte Informationen*

  Das Auslagern von Informationen in eine zentralisierte Datenbank erlaubt den Zugriff auf denselben Datenstamm durch verschiedene Partner. Ein Anwendungsbeispiel wäre unter anderem das zentrale Speichern von Produkten, welche bei beiden Partnern Verwendung finden. Dies erleichtert die Erhaltung der Datenkonsistenz

durch Aussparung der, bei verteilten Datenstämmen erfolderlichen Synchronisierungen.

Abbildung 4.2 dient an dieser Stelle zur erneuten Veranschaulichung der genannten Zentralisierungselemente des Partner-Relations-Management.

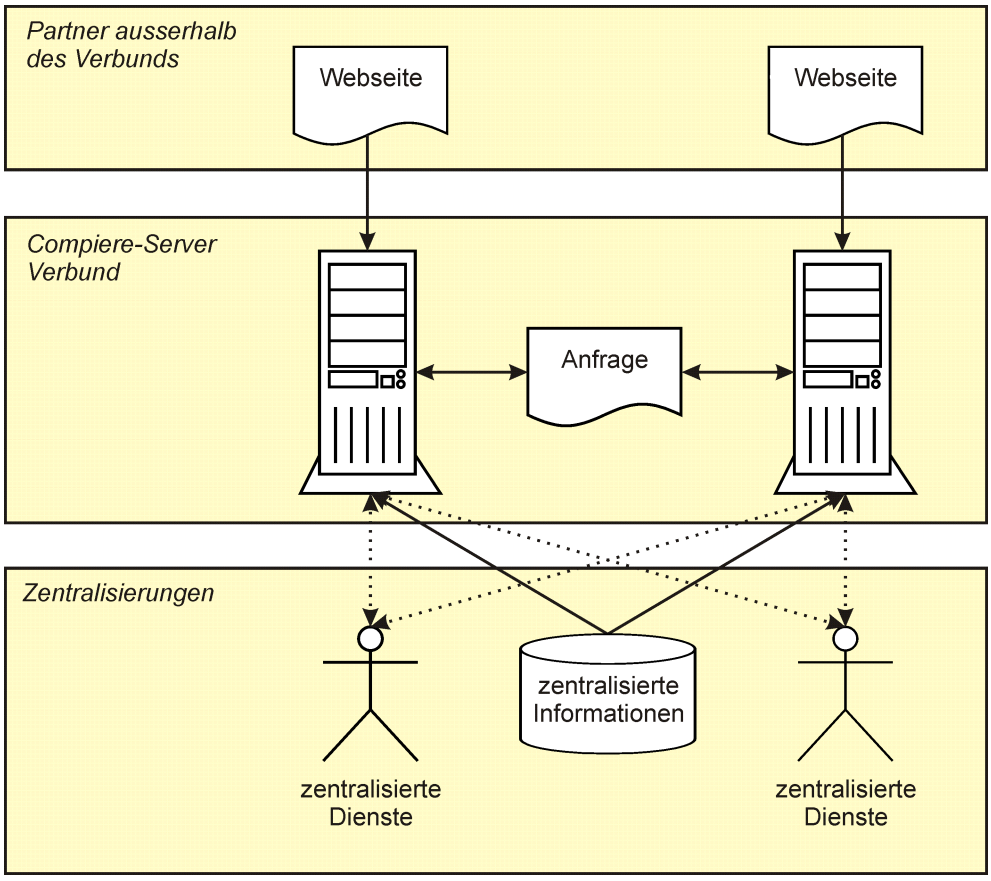

Abbildung 4.2: Zentralisierungen in einem Compiere-Systemverbund (Nach [Com])

4.1.5 Supply-Chain-Management (SCM)

Das schon in Kapitel 2.1 erläuterte Supply-Chain-Management deckt alle Aktivitäten innerhalb des Materialmanagements ab. Damit verbunden sind unter anderem das Ausstellen von Bestandsbelegen, das Einleiten von Lieferungen und die Zählung des Warenbestandes.

4.1.6 Performanz Analyse

Diese Komponente umfasst die Kostenabrechnung sowie die Buchhaltung. Es lassen sich zu diesem Zweck *Reports* von jedem Dokument innerhalb des Compiere-Systems erstellen. Die generische Generierung lässt sich dabei durch Anpassungen spezialisieren und

so auf die besonderen Bedürfnisse zuschneiden. Sie können erstellt und genutzt werden während einer Neudarstellung, Neubewertung oder Kontierung vorhandener Daten.

4.1.7 Web-Store

Ein integraler Bestandteil der Compiere-Server-Umgebung ist der JBoss-Applikationsserver (Siehe 4.2.1). Er beinhaltet einen Container für die Nutzung von Internetinhalten in Form eines Webserver. Compiere nutzt diesen, um neben administrativen Bestandteilen auch einen rudimentären Web-Store anzubieten. Während der Generierung von dynamischen Inhalten wird hier direkt auf der zentralen Datenbank gearbeitet. Da der Compiere-Client ebenfalls direkt auf derselben Datenbank arbeitet, entfällt somit die Anforderung einer passenden Synchronisation.

Der Web-Store erlaubt unter anderem die Ansicht des Produktkataloges, wobei für registrierte Geschäftspartner Zusatzinformationen angezeigt werden können. Daneben besitzt er des Weiteren Warenkorbmodule für eine Auswahl von Produkten zur späteren Bestellung. Er bietet damit die Grundfunktionalitäten eines Web-Shop-Systems. Darauf aufbauend lässt er sich erweitern und an die individuellen Umgebungen und Bedürfnisse anpassen.

4.1.8 Workflows

Compiere bietet die Möglichkeit der Nutzung von `Workflows`. Dabei bestehen diese aus einer Kette von empfohlenden Arbeitsschritten, wobei ein Knoten aus dieser Kette unterschiedlichen Typs sein kann. Ein Typ kann hier z.B. ein `Form` oder ein `Window` sein. Sie werden in Abschnitt 4.2.5 noch eingehende Erläuterung finden. Der Workflow selbst ist nicht bindend, d.h. der Benutzer hat stets die Wahl der empfohlenen Kette zu folgen oder nicht. Da Unternehmen bestrebt sind die Workflows optimal zu gestalten, erwächst daraus die Anforderung die jeweiligen Abweichungen der Benutzer von der empfohlenen Kette zu erkennen. Mit diesen Informationen liessen sich die Schwachstellen aktueller Workflows erkennen, welche anschließend zu einer Optimierung derselben beitragen können. Die Unterstützung dieser Anforderung wird im weiteren Verlauf dieser Arbeit noch weiter Thema bleiben. Sie ist eines der Ziele bei der Entwicklung geeigneter Aspekte zur Stärkung der Wandlungsfähigkeit von Compiere.

4.1.9 Zusammenfassung

Wie in den letzten Abschnitten gezeigt wurde, besitzt Compiere eine Vielzahl an Funktionalitäten, die durch Lagerhaltung, Finanzbuchhaltung, Anlagenverwaltung sowie eine Preis- und Stücklistenverwaltung abgerundet werden. Weitere positive Eigenschaften sind die Benutzerverwaltung auf Rollenbasis, die zahlreich verfügbaren Sprachmodule, die Unterstützung diverser Währungen und Rechenlegungssysteme sowie die Fähigkeit mehrere Mandanten zu unterstützen.

Durch das Fehlen einer Absatzplanung sind jedoch Defizite in der Materialbedarfsplanung (*Material Requirement Planning*)zu erkennen, wodurch das Produkt minder für

Handelskonzerne geeignet ist. Daneben sind *Workflows* durch die einfache Darstellung der Prozesskette nur rudimentär implementiert. Die Ausrichtung der Finanzbuchhaltung an das amerikanische Finanzrecht, macht diese Komponente zudem nur bedingt einsetzbar[2] für andere Länder und speziell für Deutschland [Hat06]. Dies sind Faktoren, welche die Benutzung von Compiere für eine umfassendere Palette von Unternehmen noch einschränkt.

Zusammenfassend lässt sich feststellen, dass der Schwerpunkt eher auf den zentralen Geschäftsprozessen des Mittelstandes liegt. Daher empfiehlt sich Compiere speziell für die kleinen- und mittelständischen Unternehmen mit bis zu 200.000 Euro Umsatz im Jahr. [Jeh05]

4.2 Aufbau und Struktur

In diesem Kapitel sollen die technischen Aspekte von Compiere nähere Betrachtung finden. Hierbei wird im Folgenden ein kurzer Überblick über die technischen Anforderungen gegeben werden, um im Anschluss mit einem tieferen Einblick in den internen Aufbau fortzufahren. Der Schwerpunkt liegt dabei auf Seiten der Softwaretechnik, da der infrastrukturelle Aufbau einer Compiere-Server-Umgebung für diese Arbeit nur von sekundärer Bedeutung ist.

4.2.1 Technische Anforderungen

Wie andere ERP-Lösungen ist auch Compiere an spezifische Umgebungen gebunden. Trotz der Bestrebungen, diese zu lockern, ist man zum Zeitpunkt der Ausarbeitung noch stellenweise auf eine fest vorgegebene Infrastruktur angewiesen. Diese soll an dieser Stelle am Release 2.5.3a kurz skizziert werden:

- Betriebssystem

  Durch die Verwendung von Java ist man hier ungebunden und kann alle von der Java-Virtual-Machine unterstützten Plattformen nutzen.

- Laufzeitumgebung

  Compiere ist seit den Anfängen ein Open-Source-Projekt, basierend auf Java. Ab dem Release 2.5.3a finden unter anderem auch Java-Generics Anwendung, was die Verwendung von Java 1.5 oder höher vorraussetzt.

- Datenbank

  Als zugrundeliegende Datenbank wird die proprietäre *Oracle Database 10g*[3] verwendet. Da diese nur bei privater Nutzung kostenfrei ist, gibt es Bestrebungen

[2] Schnittstellen zur elektronischen Steuererklärung (ELSTER) und für das Homebanking (Home-Banking Computer Interface) fehlen

[3] http://www.oracle.com

eine Adaption für andere, freie Datenbanken vorzunehmen. *Compiere Libero*[4] ist ein Projekt, welches dieses Ziel in Kombination mit der Open-Source-Datenbank *PostgreSQL*[5] verfolgt. Auch die Compiere Inc. hat sich dies als ein Ziel für zukünftige Weiterentwicklungen gesetzt und ab dem Release 2.5.2a eine Datenbankunabhängigkeit[6] für Compiere proklamiert. Diese befindet sich jedoch bis dato noch im Betastadium und wird daher in dieser Arbeit keine weitere Betrachtung finden.

- Applikations-Server

  Compiere besitzt eine geringe Anzahl an Komponenten, die in einem Java-Applikations-Server laufen müssen. Dies sind vorwiegend Komponenten der Webschnittstellen welche die Form eines Online-Shop-System sowie einer Administrationsplattform haben. Der funktionale Kern von Compiere ist jedoch auch ohne diese Komponenten lauffähig. Der Applikationsserver ist somit optional zu starten und bildet keine Voraussetzung für die Benutzung des Compiere-Systems. Im Paket von Compiere wird der frei verfügbare Applikations-Server *JBoss*[7] mitgeliefert. Mit dem Compiere-Release 2.5.3a wurde von der JBoss-Version 3.2 auf 4.0 gewechselt. Ein Wechsel zu anderen Applikations-Servern ist möglich, solange diese die J2EE-Spezifikationen erfüllen. Daneben sind Entwicklungsbestrebungen zur Anpassung der Compiere-Pakete an speziellere J2EE-Applikations-Server, wie dem *IBM WebSphere Applikations-Server*[8], geplant. Diese erlauben dann die Nutzung der spezifischen Funktionen des jeweiligen Applikations-Server, die von den J2EE-Spezifikationen abweichen können. Weitere Details zu der Verwendung des JBoss-Applikations-Server innerhalb von Compiere finden sich im Anschluss an diesen Abschnitt.

4.2.2 Applikationsstruktur

Wie im Kapitel 2 beschrieben, liegt die Hauptaufgabe eines ERP-Systems in der Planung und Verwaltung von Ressourcen innerhalb eines Unternehmens. Diese Anforderung impliziert das Vorhandensein einer steten Verbindung zu den benötigten Daten. Typischerweise werden diese Daten dazu innerhalb eines *Datenbankmanagementsystem* (Siehe 2.4) organisiert.

So ist auch der Aufbau des ERP-Systems Compiere in hohem Maße an das verwendete DBMS ausgerichtet. Der Zugriff auf die Daten findet dabei über SQL-Abfragen statt deren Ausprägungen dem verwendeten DBMS entsprechen. Sie sind entweder direkt innerhalb des Programmcodes definiert oder werden durch sogenannte *Persistenzobjekte* generiert. Hilfsobjekte führen diese SQL-Abfragen mittels eines JDBC[9]-Treibers aus,

[4] http://www.e-evolution.com.mx/postgre.html
[5] http://www.postgresql.org
[6] http://www.compiere.org/news/0502-R252.html
[7] http://www.jboss.com
[8] http://www-306.ibm.com/software/webservers/appserv/was
[9] *Java-Database-Connectivity* - ein von Sun Microsystems entwickeltes Java-Paket zur Anbindung von Datenbanken innerhalb von Java-Anwendungen

wobei die Ergebnisse durch die Persistenzobjekte ausgewertet und durch ihre Wrapper-Funktionalitäten zugänglich gemacht werden. Die dabei verwendete Technik wird im Abschnitt 4.2.4 nähere Erläuterung finden.

Aus Sicht des Clients unterteilt sich Compiere in zwei Kernkomponenten, welche den Zugriff auf Compiere-Funktionalitäten ermöglichen - dem JBoss-Applikations-Server und der Java-Client-Anwendung. Sie sollen im Folgenden kurz beschrieben und anhand der Abbildung 4.3 weiter veranschaulicht werden.

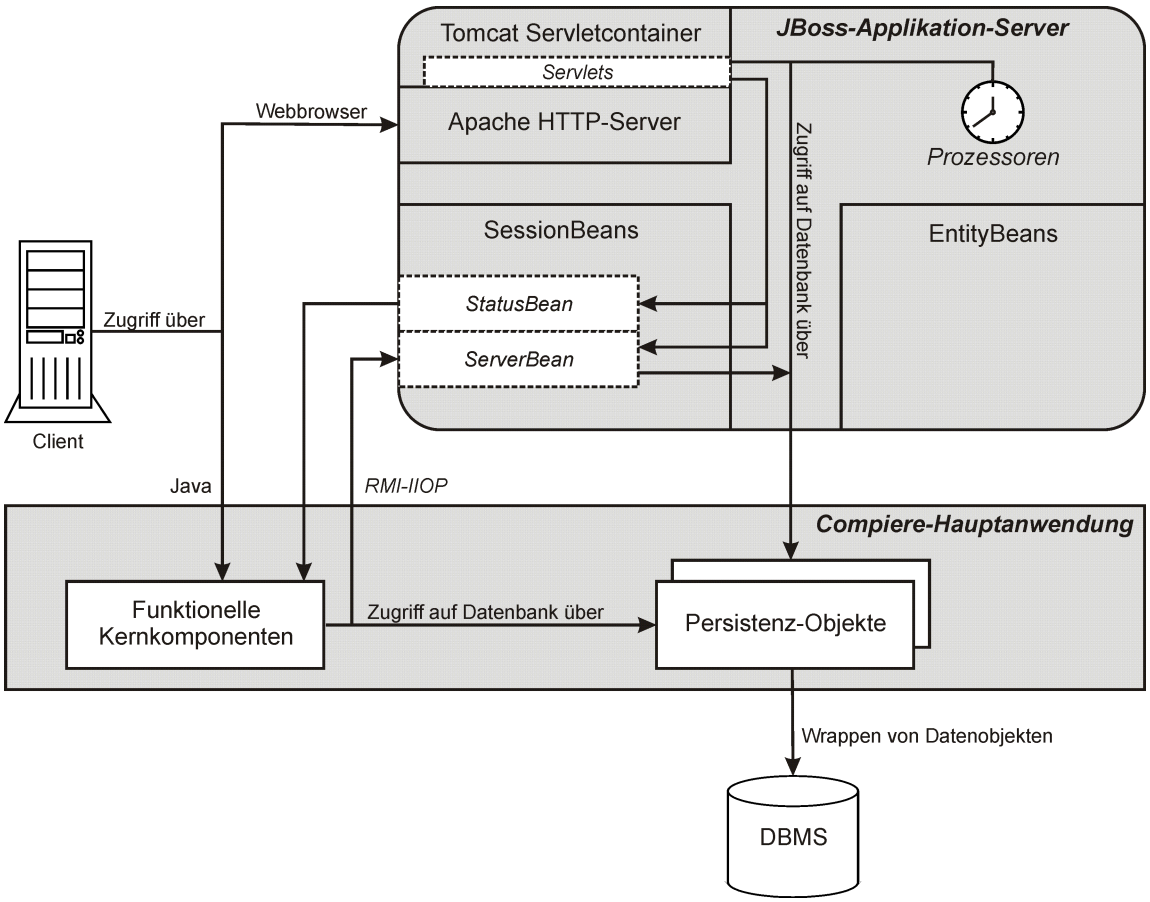

Abbildung 4.3: Zugriffsebenen der Compiere-Architektur

JBoss-Applikations-Server

JBoss findet innerhalb der Compiere-Applikations-Architektur nur rudimentär Verwendung. Die potentiellen Möglichkeiten eines J2EE-Applikations-Server, wie der Nutzung persistenter EntityBeans anstelle der selbst generierten Persistenzobjekte, werden durch Compiere nur marginal eingesetzt. Hauptaugenmerk bei der Benutzung des JBoss-Applikations-Server liegt bei dem Compiere-System auf der Verwendung des integrierten *Apache*

HTTP-Server[10] in Kopplung mit dem *Tomcat Servletcontainer*[11].

Neben dem statischen Inhalt innerhalb des Apache HTTP-Servers werden eine Reihe von Java-Servlets eingesetzt, welche dynamische Webseiten ermöglichen. Da es innerhalb des Compiere-Deployments keine EntityBeans gibt, greifen die Servlets ebenfalls auf die Persistenzobjekte zurück, wenn diese Zugriff auf die Datenbank benötigen. Die Servlets selbst dienen der Implementierung des Compiere-WebStore, einem rudimentären Online-Shop und einer Managementumgebung. Das Management umfasst hierbei die Anzeige von Statusinformationen zur Datenbank sowie die Möglichkeit, die Prozessoren, welche im Anschluss in Abschnitt 4.2.2 noch eingehender erläutert werden, zu verwalten. Zum Zweck der Statusübersicht greifen die Servlets unter anderem auch auf die SessionBeans zurück. Der Client-Zugriff auf die Servlets erfolgt über einen herkömmlichen Internetbrowser, der auf den Apache HTTP-Server zugreift. Anforderungen dynamischer Inhalte werden von dort aus an die Apache Tomcat-Komponente weitergeleitet, die daraus statische Ausgaben generiert, welcher der Übergabe an den Client durch den Apache HTTP-Server dienen.

Neben der Nutzung der HTTP/Servlet-Komponenten werden zusätzlich *Enterprise Java Beans* (EJB) verwendet, die jedoch nur minimal ausgebildet sind. Neben den SessionBeans `ServerBean` und `StatusBean` werden innerhalb des Applikations-Servers durch Compiere keine weiteren Entity- oder Session-Beans registriert. Darüber hinaus wird lediglich das SessionBean `ServerBean` durch die Compiere-Hauptanwendung, technisch durch die Verwendung von *RMI-IIOP (Remote-Method-Invocations over the Internet Inter-ORB Protocol)*, genutzt. An ihr werden alle Datenbankabfragen des Clients geschickt. Erst wenn der Applikations-Server nicht erreichbar ist, werden die Datenbankabfragen durch die *funktionellen Kernkomponenten* direkt an die Persistenzobjekte gesendet. Das `ServerBean` dient dabei in erster Linie als Relay. Es sendet die eingehenden SQL-Statements direkt an die Persistenzobjekte weiter.

Zusammenfassend ist die Verwendung des Applikationsserver für Compiere nur eingeschränkt von Nutzen. Lediglich die Integration der im Anschluss betrachteten *Prozessoren* und der Web-Shop-Komponente, machen den funktionellen Nutzen des JBoss-Applikationsserver für Compiere aus. Neben der Stärkung einer sonst nicht vorhandenen Client-Server-Architektir, bietet es jedoch auch zahlreiches Potential für weiterführende Verwendungszwecke. Die Entwicklung zukünftiger Releases wird hier eventuell ansetzen und diese Technik noch weiter ausbauen und so den J2EE-Applikations-Server verstärkt in die Struktur von Compiere einbinden.

Prozessoren

Der schon erwähnte Umstand, dass Compiere keine wirkliche Client-Server-Architektur besitzt, liegt in dem Fehlen eines expliziten Servers begründet. Die Zugriffsmechanismen, d.h. die Client-Anwendung und der Webbrowser, bilden jeweils eine eigenständig agierende Anwendungsebene, welche die Datenbank ändern kann. Einzig und allein die Persistenzobjekte sichern einen konsistenten Datenbestand während der asynchronen

[10] http://httpd.apache.org
[11] http://tomcat.apache.org

Zugriffsoperationen, wie dem Speichern eines Tabelleneintrags, ab. Die Eigenständigkeit jedes dieser agierenden Clients unterbindet die Nutzung eines globalen Prozesses, der eine permanente Aktivität seinerseits sicherstellt. Dem wurde durch die zusätzliche Nutzung des JBoss-Applikations-Servers abgeholfen.

Im JBoss-Applikations-Server werden zu diesem Zweck sogenannte *Prozessor*-Objekte eingerichtet. Als Adaptionen der Java-Klasse *Thread* laufen sie parallel als Objekte innerhalb der Prozesshierarchie. Verwaltet werden diese Prozessoren durch einen Verwaltungsdienst, dem `CompiereServerMgr`-Objekt. Dieses zentrale und auf Serverebene einmalig instanziierte Objekt, erstellt mit Hilfe der entsprechenden Informationen aus der Datenbank alle Prozessor-Objekte und startet bzw. stoppt diese bei Bedarf. Alle Prozesse besitzt dazu innerhalb der Datenbank einen eigenen Eintrag, der über die Persistenzobjekte ausgelesen wird. Jeder dieser speziellen Persistenzobjekte adaptiert dabei ein generelles Persistenzobjekt (PO) der einen generischen Zugriff auf Datenbankelemente ermöglicht. (Abb. 4.4) Eine umfassendere Beschreibung des Datenbankzugriffes mittels der Persistenzobjekte vermittelt der Abschnitt 4.2.4.

Die Prozessoren werden genutzt, um typische Serveraufgaben innerhalb der Compiere-Umgebung zu übernehmen. Der Kern liegt dabei auf der Überwachung von eintretenden Ereignissen innerhalb des Compiere-System und dem Ausführen von daraus resultierenden Aufgaben. Als Beispiel soll an dieser Stelle der `AlertProcessor` dienen. Dieser prüft in zyklischen Abständen den Bedarf zum Versand von EMail-Adressen, welche als Hinweis bei Erreichen eines Warnzustandes dienen. Dazu liest dieser in jedem dieser Zyklen alle vorhandenen Alarm-Einträge in der Datenbank aus und prüft deren Bedingungen. Sind diese Bedingungen erfüllt, erfolgt der Versand von Warnhinweisen per EMail an die jeweils in der Datenbank definierten Empfänger. Zusammen mit diesem Prozessor zur Überwachung von Warnzuständen gibt es in dem betrachteten Compiere-Release 2.5.3a vier weitere Prozessoren:

- `AcctProcessor`
- `AlertProcessor`
- `RequestProcessor`
- `Scheduler`
- `WorkflowProcessor`

Sie dienen der Überwachung und Unterstützung von unternehmensweiten Elementen, wie den Aufträgen oder Workflows, und werden damit zu kritischen Komponenten innerhalb der ERP-Lösung. Eine Einbettung innerhalb zertifizierter J2EE-Applikations-Server, wie dem verwendeten JBoss-Applikations-Server, wirkt hier zusätzlich unterstützend auf den Grad der Zuverlässigkeit der Gesamtlösung. Ein J2EE-Applikationsserver stellt jedoch eine sehr umfangreiche Technologie dar. Inwieweit die Nutzung eines solchen zum Zwecke der Integration der *Prozessoren* und des *Web-Store* zu rechtfertigen ist, bleibt offen.

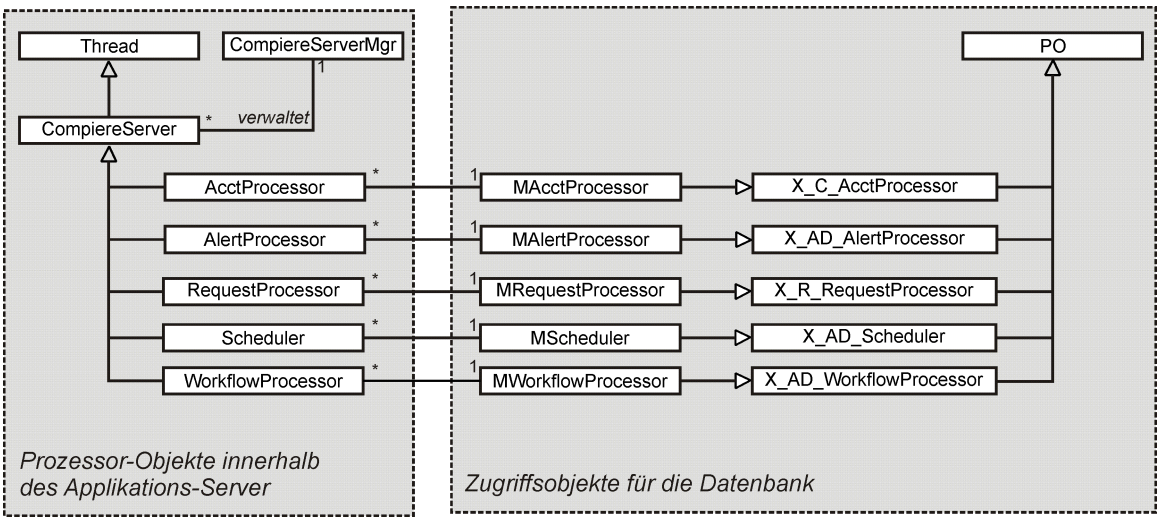

Abbildung 4.4: Architektur der Compiere-Prozessor-Implementierung

Compiere-Hauptanwendung

Neben dem Zugriff über einen Webbrowser bietet Compiere eine Java-basierende Client-Anwendung. Anders als die Webschnittstelle deckt sie alle funktionalen Kernelemente des gesamten Systems ab und wird damit in diesem Kontext als *Hauptanwendung* bezeichnet. Sie unterteilt sich in die Bereiche der *funktionalen Kernkomponenten* und die der *Persistenzobjekte*.

Die Compiere-Hauptanwendung dient der Darstellung und Bearbeitung der in der Datenbank gespeicherten Daten. Sie ist äußerst stark auf den Inhalt der Datenbank ausgerichtet. So beinhaltet die Datenbank zum Beispiel sämtliche Daten über die, in der Hauptanwendung dynamisch darzustellenden Fenster und deren Inhalte. Diese Aufteilung ermöglicht eine grafische Oberfläche, die grösstenteils generisch erzeugt wird und Änderungen an ihr erleichtert. Die Datenbank wird damit zur maßgeblichen Komponente des gesamten Systems. Kapitel 4.2.5 verdeutlicht die Beziehung zwischen darstellender Ebene und Datenbankinhalten noch eingehender.

4.2.3 Datenbankstruktur

In den vorangegangenen Kapiteln wurde bereits die starke Datenbankbindung von Compiere durch die grafische Oberfläche erwähnt. Zusätzlich findet die Datenbank auch Verwendung bei der Speicherung anwendungsbezogener Daten, wie Produktdaten oder Ähnlichem. Zur Verbesserung der Übersicht über alle Compiere-Tabellen, werden diese mittels ihrer funktionellen Zuordnungen gruppiert. Dazu erhalten sie jeweils einen Gruppenpräfix im Tabellennamen. Sie sollen im Folgenden kurz genannt sein. Aufgrund der besonderen Rolle des *Application Directory* für Compiere, werden die zugehörigen Tabellen in detaillierterer Form beschrieben.

- `AD_ (Application dictionary)`

Beispiel: `AD_WINDOW, AD_WORKFLOW, AD_ALERTPROCESSOR`

Tabellen mit diesem Präfix bilden Kernkomponenten der Compiere-Applikations-Architektur ab. Im Gegensatz zu allen anderen Tabellen, die Teil der Funktionalitäten sind, dienen diese Tabellen der Bereitstellung der Funktionalitäten selbst. Sie sind damit wesentlicher Bestandteil der Systemarchitektur und können als infrastrukturelle Komponenten aufgefasst werden. Diese Komponenten können unter anderem Prozessoren (Siehe 4.2.2), Benutzerrollen oder Workflows sein. Sie alle finden in den jeweiligen Tabellen ihre Repräsentation innerhalb der Datenbankstruktur.

Im Gegensatz zu anderen Tabellen, müssen Tabellen mit dem Präfix `AD_` permanent vorhanden sein. Sie bilden damit die theoretische Grundmenge aus der Menge aller vorhandenen Tabellen, auch wenn diese in der Praxis aufgrund einzelner hartcodierter Abfragen auf nicht-permanente Tabellen abweicht. Alle anderen Tabellen sind erst in dem Moment von Nöten, wenn dynamische Elemente aus den permanenten Tabellen heraus generiert werden, wie z.B. ein `Window` (Siehe 4.2.5) über einen Eintrag in der Tabelle `AD_WINDOW`. Aus diesem Grund sind diese Tabellen von besonderer Bedeutung. Einige von ihnen werden nun einer kurzen Betrachtung unterzogen:

- `AD_TABLE`

  Diese Tabelle dient der Speicherung von Metadaten über alle Compiere-Tabellen innerhalb der Datenbank. Dies umfasst Daten wie z.B. Beschreibungen der Tabelle selbst oder die benötigten Zugriffsrechte.

- `AD_COLUMN`

  Metadaten über jede Spalte einer jeden Compiere-Tabelle werden innerhalb dieser Tabelle gespeichert. Der Standardwert einer Spalte oder der Callout-Eintrag (Siehe 4.2.5) sind Beispiele solcher Metadaten.

- `AD_WINDOW`

  Diese Tabelle speichert alle dynamischen *Window*-Elemente (Siehe 4.2.5) und umfasst Daten über das zu erstellende *Window*, wie z.B. dessen Fenstersymbol oder initiale Fenstergrösse.

- `AD_TAB`

  Jedes *Window* beinhaltet eine Anzahl von Tab-Feldern, welche in dieser Tabelle gespeichert werden. Sie beinhaltet Informationen über jeden einzelnen Tab, so z.B. den Namen oder ob Änderungen am Datensatz in diesem Tab erlaubt sind.

- `AD_FIELD`

  Ein Tab gruppiert Spalten einer Tabelle zur späteren Nutzung in einem grafischen Element des Typs *Window*. Die Spalten sind dabei durch Einträge

der Tabelle AD_COLUMN-Tabelle mit Metdadaten angereichert und werden durch ein Field repräsentiert. Ein Field-Eintrag dient der weiteren Anreicherung mit Metadaten, welche zur Darstellung innerhalb des *Window* nötig sind. Dies sind zum Beispiel Informationen über die Anordnung des Field innerhalb eines Tab oder sprachliche Beschreibungen zum Field selber.

– `AD_ELEMENT`

Jeder Eintrag der Tabelle `AD_COLUMN` verfügt über eine Referenz zu einem Elementtyp, der in dieser Tabelle gespeichert wird. Er definiert den Datentyp der durch `AD_COLUMN` repräsentierten Spalte und dient der Anzeige des passenden grafischen Elements innerhalb des Fields.

- `A_` (Asset Management)
- `C_` (Core Functionality)
- `GL_` (General Ledger)
- `I_` (Import)
- `K_` (Knowledge Base)
- `M_` (Material Management)
- `PA_` (Performance Analysis)
- `R_` (Request)
- `RV_` (Report Viewer)
- `S_` (Service Management)
- `T_` (Temporary)
- `W_` (Webstore)

Die Nutzung der genannten Präfixe bei der Benennung der Tabellen dient der semantische Unterscheidung der Tabellen. Es werden so Gruppen gebildet, die hierarchisch oder funktional zusammengehörig sind und dadurch die Lesbarkeit des Tabellenschemas erhöhen. Die hierarchische Trennung findet hier primär zwischen den permanenten Tabellen des Application Directory, d.h. den Tabellen mit dem Präfix `AD_`, und den restlichen dynamischen Tabellen statt. Die dynamischen Tabellen gruppieren sich dagegen untereinander hauptsächlich durch funktionale Eigenschaften. So bilden zum Beispiel alle Tabellen, welche für den Webstore relevant sind, die entsprechende Gruppierung mit dem Präfix `W_`.

Im Folgenden soll nun die Zugriffstechnik auf die genannten Tabellen erläutert werden.

4.2.4 Datenbankzugriff

Die Benutzung der in den vorangegangenen Kapiteln erläuterten Datenbankstruktur wird mittels einer Vielzahl von Objekten ermöglicht. Diese werden in diesem Abschnitt näher erläutert. Abbildung 4.5 legt den hier beschriebenen Sachverhalt grafisch dar. Sie dient als Grundlage der nun folgenden Betrachtung, welche unterteilt ist in die Sichtweise der Anwendungsebene und die Datenbank-Kommunikationsebene.

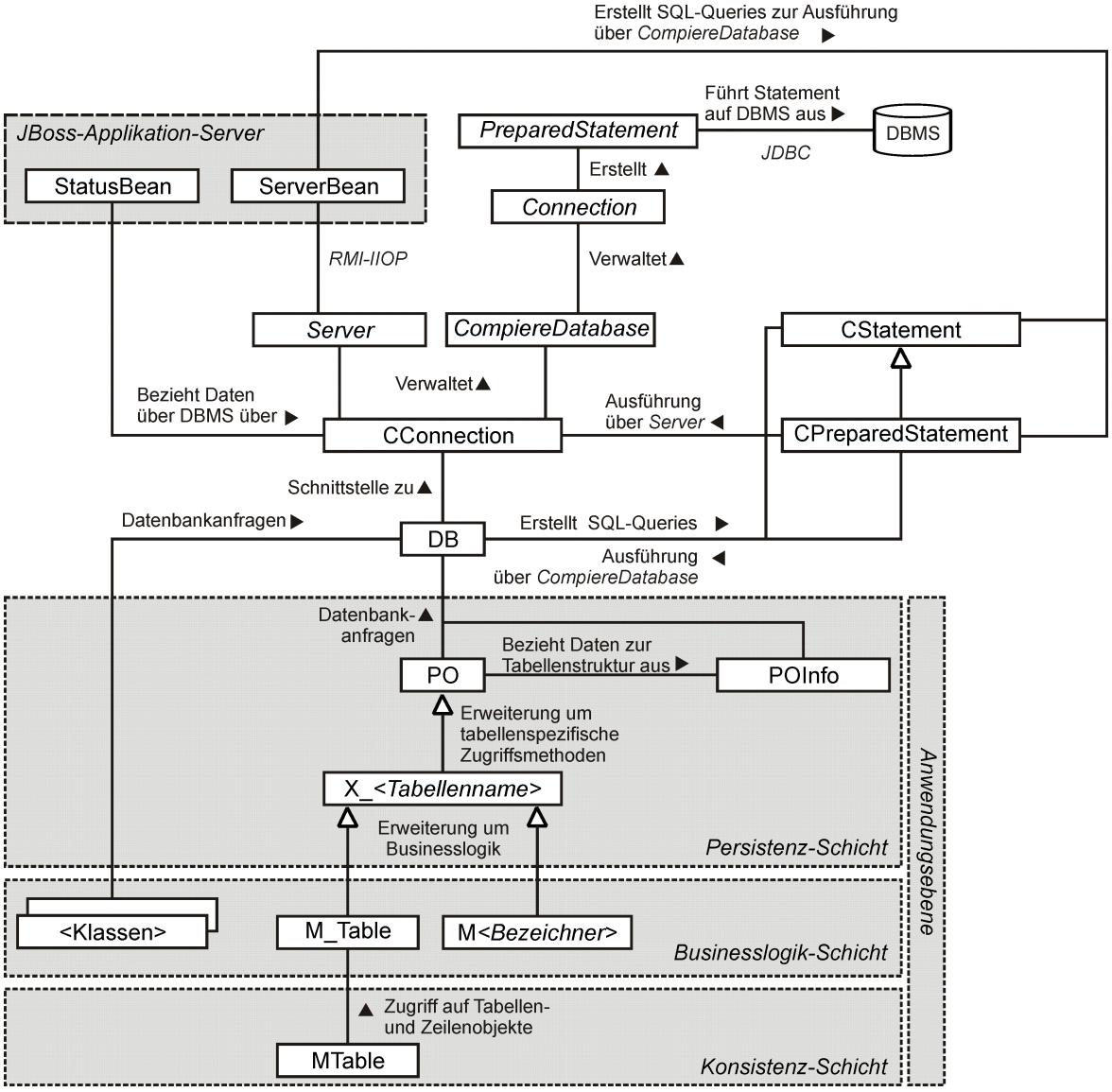

Abbildung 4.5: Klassendiagramm der beim Datenbankzugriff beteiligten Objekte

Zugriff durch Objekte der Anwendungsebene

Als zentrales Element der gesamten Zugriff- und Kommunikationsebene steht die Klasse *DB*. Sie ermöglicht für alle Objekte der Anwendungsebene einen uniformen Zugriff auf das Datenbank-Management-System (DBMS). Als Objekte der Anwendungsebene werden hierbei alle Elemente aus den Ebenen *Persistenz*, *Businesslogik* sowie *Konsistenz* bezeichnet. Die Klasse DB bietet zum Zweck dieses Zugriffs eine Vielzahl von Methoden, welche die Ausführung von Datenbank-Operationen auf dem DBMS ermöglichen. All diese Methoden sind dabei statisch implementiert, was den Zugriff auf ihre Funktionalitäten erleichtert. Der Aufgabenschwerpunkt dieser Klasse liegt auf der Ausführung von SQL-Statements, welche durch Persistenzobjekte bzw. anderen Objekten der Anwendungsebene an die entsprechenden Methoden der DB-Klasse übergeben werden.

Diese SQL-Statements sind bei nicht-Persistenzobjekten, in der Grafik als «Klassen» innerhalb der *Businesslogik-Schicht* symbolisiert, fest im Quellcode definiert und legen damit die Nutzung von permanenten Tabellen, wie den Tabellen des *Application Directory*, nahe. Da sich bei dynamischen Tabellen die Bezeichner jedoch ändern können, findet der Zugriff hier über Persistenzobjekte statt. Jede Tabelle verfügt zu diesem Zweck über je eine Persistenzklasse, die als Bezeichner den Tabellennamen mit einem vorangestellten `X_` trägt. In der Grafik werden diese als `X_<Tabellenname>` gekennzeichnet. Sie können mit Hilfe eines kleinen Hilfsprogramm aus dem Compiere-Paket automatisch für eine Tabelle generiert werden. Dazu wird die entsprechende Tabelle ausgelesen und zu jeder Spalte eine, zum Elementtyp der Spalte passende, Getter- und Setter-Methode erzeugt.

Jede dieser Persistenzklassen erbt von dem Persistenzobjekt `PO`, das Methoden für einen generischen Zugriff auf Tabellenobjekte bietet. Ein solches `PO`-Objekt spiegelt hierbei eine Zeile der durch sie abgebildeten Tabelle wieder. Sie wird von den Getter/Setter-Methoden der Persistenzklassen `X_<Tabellenname>` verwendet, um die entsprechenden Spaltenwerte für diese Zeile auszulesen. Die spezifischen Eigenschaften jeder Tabelle werden durch diese von `PO` erbende Klasse und dem Objekt `POInfo` implementiert. `POInfo` ermittelt so zum Beispiel die Spaltennamen der abgebildeten Tabelle und stellt diese `PO` bereit. In Kombination mit Daten aus dem Objekt `POInfo` und den generischen Methoden aus `PO`, ist die Klasse `X_<Tabellenname>` damit in der Lage tabellenspezifische SQL-Statements zu generieren und diese an das Objekt DB zur Ausführung zu senden.

Daneben besitzen einige der `X_<Tabellenname>`-Persistenzklassen eine weitere Spezialisierung. Sie sind durch ein vorangestellten `M` gekennzeichnet und tragen in der Grafik den Namen `M<Bezeichner>`. Sie erweitern die strikte Abbildung der Tabellenstruktur mittels der Klassen `X_<Tabellenname>` um weitere Funktionalitäten der Business-Schicht. Durch die Definition hardcodierter SQL-Statements können sie um die reine Nutzung der Getter/Setter-Methoden der Superklasse hinausgehen und ermöglichen damit die Implementierung weiterführender Funktionalitäten.

Damit ist es den Objekten der Business-Schicht möglich, alle Elemente der Datenbank zu verändern. Aufgrund dieser Offenheit gegenüber Änderungen an allen Datenbankobjekten, wird durch die Elemente der Persistenz- und Businesslogik-Schicht keine *Datenkonsistenz* sichergestellt. Eine zwingende Aufrechterhaltung der Konsistenz würde

in diesem Fall ein Zwischenspeichern und Überprüfen aller Daten bei jedem Speichervorgang erfordern und den Aufwand nur bedingt rechtfertigen. Der schreibende Zugriff auf Objekte über die Nutzung der PO-Subklassen erfolgt also ohne weitere Konsistenzprüfung der zu schreibenden Daten.

Eine Aussnahme bildet die Klasse `MTable`. Sie bietet eine Konsistenzprüfung und ist selbst Bestandteil der grafischen Darstellung einer Tabelle mittels eines *Tab* in einem *Window* der Compiere-Hauptanwendung (Siehe 4.2.5). Innerhalb des Speichervorganges eines solchen *Window* werden für jede Tabelle, die über *Tabs* dargestellt sind, Konsistenzprüfungen vorgenommen. Diese beziehen sich immer nur auf die jeweils in der Tabelle vorkommenden Elemente und erfordern damit nicht so viel Aufwand wie es eine Konsistzenzprüfung auf Ebene der Businesslogik-Schicht würde. Die Prüfung wird dabei über einen Vergleich der Originaldaten, d.h. der zwischengespeicherten Daten des letzten Einlesevorgangs mit den derzeit in der Datenbank vorliegenden Daten getätigt. Sie erlaubt das Erkennen von zwischenzeitlichen Änderungen am Datensatz der Datenbank und verhindert so Änderungskollisionen.

Zum Einholen der nötigen Informationen benutzt `MTable` dabei die `X_AD_Table`-Subklasse `M_Table`. Sie erlaubt den generischen Zugriff auf jede Tabelle anhand der jeweiligen *AD_Table_ID*. Für den Zugriff auf eine Zeile der jeweiligen Tabelle bietet `M_Table` des Weiteren Factory-Methoden, die ein entsprechendes `PO`-Subklassen-Objekt, wie z.B. `X_C_BPartner`, anhand ihrer Primärschlüssel erstellen. Erweiternd veranschaulicht dazu die Abbildung 4.8 die erwähnten Konsistenzobjekte.

Der in diesem Abschnitt aufgezeigte Zusammenhang zwischen den einzelnen Persistenzobjekten und der Spezialisierung durch die Objekte der Business-Schicht wird durch einen Auszug des Quellcode im Listing 4.1 näher verdeutlicht. Die Klasse `MProduct` ist dabei eine Erweiterung der Persistenzklasse `X_M_Product` um weitere Businessfunktionalitäten. Sie benutzt dabei Getter- und Setter-Methoden der Persistenzklasse (Zeile 6). Diese wiederum greifen auf die generischen Getter- und Setter-Methoden des Persistenzobjektes `PO` zurück (Zeile 16), welches Inhalte der Tabelle in Form von Arrays zwischenspeichert. Ein Zugriff auf die entsprechenden Einträge wird über den Index der jeweiligen Spalte ermöglicht, der mit Hilfe die Klasse `POInfo` in Erfahrung gebracht wird (Zeile 28).

Listing 4.1: Zugriff auf Tabelleninhalte durch Objekte der Persistenz- und Businessschicht am Beispiel der Tabelle *M_Product*

```
1  public class MProduct extends X_M_Product{
2     ...
3     public int getA_Asset_Group_ID()
4     {
5        ...
6        int i = getM_Product_Category_ID();
7        ...
8     }
9  }
10
```

```
11 public class X_M_Product extends PO{
12   ...
13   public int getM_Product_Category_ID()
14   {
15     Integer ii =(Integer)get_Value("M_Product_Category_ID");
16     if (ii == null) return 0;
17     return ii.intValue();
18   }
19 }
20
21 public abstract class PO ...{
22   protected POInfo p_info;
23   ...
24   public final Object get_Value (String columnName){
25     ...
26     return m_oldValues[p_info.getColumnIndex(columnName)];
27   }
28 }
29
30 public class POInfo implements Serializable{
31   ...
32   public int getColumnIndex (String ColumnName){
33     ...
34   }
35 }
```

Objekte der Datenbankkommunikations-Ebene

Die im vorherigen Abschnitt genannten Objekte bilden den Bereich der Anwendungsebene, welcher den Zugriff auf Datenbankelemente durch die Abbildung von Tabellenstrukturen durch die Klassen `X_<Tabellenname>` bzw. der Nutzung fest codierter SQL-Statements ermöglicht. Die dabei genutzten SQL-Statements werden durch die Klassen `CStatement` und `CPreparedStatement` gekapselt und der zugrundeliegenden Datenbank angepasst. Die direkte Kommunikation mit dem DBMS und der damit einhergehenden Ausführung dieser SQL-Statements ist Aufgabe der nun folgenden Objekte.

Eines der zentralen Elemente ist das Objekt `CConnection`. Es ist eine nach dem *Singleton*-Pattern[12] implementierte Klasse und dient der Verwaltung der Verbindungsobjekte `CompiereDatabase` und `Server`, die im Folgenden erläutert werden.

Die Entwicklung von Compiere basierte über einen grossen Zeitraum hinweg auf der Datenbank *Oracle Database*. Dementsprechend sind viele hart-codierte SQL-Statements im Code von Compiere auf diese Datenbank ausgelegt. Die späte Implementierung der Unterstützung weiterer Datenbanken wurde dadurch erschwert. Um diese verteilten

[12] Das Singleton-Entwurfsmuster schreibt die Implementierung einer Klasse unter nur einmalig erlaubter Instanziierung vor. Zu diesem Zweck verfügt diese über einen nicht-öffentlichen Konstruktor. Die einzige Instanz wird in diesem Falle über eine statische Methode erzeugt, welche bei jedem Aufruf entweder diese einzige Instanz mittels des privaten Konstruktor erzeugt oder genau diese dem Aufrufer zurück gibt.

Statements beibehalten zu können, werden durch die Schnittstelle `CompiereDatabase` Methoden zur Translation von SQL-Statements definiert. Sie ermöglichen die Umwandlung von SQL-Statements der Datenbank *Oracle Database* in das Format der jeweiligen Datenbank. Zum Zeitpunkt der Ausarbeitung (Release 2.5.3a) wird dieses Verbindungsobjekt von den 3 Klassen `DB_Oracle`, `DB_PostgreSQL` und `DB_Sybase` implementiert. Nichtsdestotrotz findet davon nur `DB_Oracle` ausgeprägte Verwendung, da derzeit keine andere Datenbank durch Compiere in ausgetesteter Form vollständig unterstützt wird.

Neben den Aufgaben der Statementtranslation verwalten sie zusätzlich Objekte des Typs `Connection`. Diese repräsentieren Verbindungssitzungen zu dem DBMS-System und sind je nach verwendetem DBMS unterschiedlicher Ausprägung. Sie ermöglichen die Zusendung von SQL-Anfragen an das DBMS in Form von Statements. Die Verbindung wird dabei über den *Java Database Connectivity (JDBC)*-Treiber der jeweiligen Datenbank aufgebaut und kontrolliert. Im Zuge der Zusendung von SQL-Statements an den Server, werden durch das `Connection`-Objekt Statement-Objekte des Typs `PreparedStatement` erstellt. Sie nehmen mit Hilfe des `Connection`-Objektes die Zusendung des SQL-Statements an das DBMS vor und empfangen deren Ergebnis in Form einer entsprechenden Serverantwort, d.h. einem `java.sql.ResultSet` bei einer Query und ein Statusflag bei einem Update.

Die zweite Art der Verbindung bildet das Objekt `Server` ab. Es ist ein *Remote Interface*[13] des EJB-SessionBean `ServerBean` und erlaubt die Ausführung dessen Methoden innerhalb des JBoss-Applikations-Server. Über *RMI-IIOP (Remote Methode Invocations over the Internet Inter-ORB Protocol)* werden dazu Methodenaufrufe an den Applikations-Server übermittelt, die dann intern mittels *JNDI (Java Naming and Directory Interface)* an dieses SessionBean gerichtet werden. Innerhalb des Containers werden diese Methoden anschschließend ausgeführt und das Ergebnis zurück an das aufrufende Objekt ausserhalb des Containers zurückgesandt. [RAJ02]

In Kapitel 4.2.2 wurde bereits auf die rudimentäre Nutzung des JBoss-Applikations-Servers hingewiesen. So beschränkt sich die Nutzung des Enterprise-Java-Bean `ServerBean` auf die Weiterleitung empfangener SQL-Statements an ein `CompiereDatabase`-Objekt, respektive an `DB_Oracle`. Dies wird über die Generierung eines `CStatement` bzw. `CPreparedStatement` erreicht. Sie werden anschließend über das `CompiereDatabase`-implementieren Objekt ausgeführt, das dazu die Objekte `DB` und `CConnection` nutzt. Es gleicht dabei der Ausführung von SQL-Statements bei einem Fehlen eines Applikations-Servers und übernimmt damit eine als redundant einzustufende Rolle.

Neben dem SessionBean `ServerBean` findet ein weiteres EJB Verwendung, das SessionBean `StatusBean`. Dieses ist aber für die Betrachtung der Kommunikationsobjekte irrelevant, da es durch die Benutzung von `CConnection` lediglich der Vollständigkeit halber mit in die Grafik 4.5 aufgenommen wurde. Dieses *Bean* dient dem Einholen von Informationen bezüglich der Datenbank im Rahmen der Management-Servlets des Apa-

[13] Weitere Informationen zum J2EE-Applikations-Server können der J2EE-Spezifikation entnommen werden: http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

che Tomcat Servlet-Containers (Siehe 4.2.2) und bezieht zu diesem Zweck Informationen über CConnection. Des Weiteren dienen die in der Grafik abgebildeten Beziehungen dem Erlangen eines besseren Verständnisses der Kommunikationsstrukturen und erheben daher keinerlei Anspruch auf totale Vollständigkeit. Zur tiefergehenden Analyse der Klassenbeziehungen sei an dieser Stelle auf den Quellcode verwiesen.

4.2.5 Grafische Abbildung der Datenbank

Wie in den vorangegangenen Betrachtungen ersichtlich wurde, ist Compiere eine ERP-Lösung, welche sehr stark auf die zugrundeliegende Datenbank ausgerichtet ist. Ein besonderer Umstand der diese Ausrichtung noch verstärkt ist die Gestaltung der grafischen Benutzeroberfläche *(Graphical-User-Interface, GUI)* mittels Einträgen aus der Datenbank. Dies soll im Folgenden nun genauerer Betrachtung unterzogen werden.

Als Erstes soll zum diesem Zweck der Aufbau der GUI kurz erläutert werden. Der Start der Compiere-Java-Hauptanwendung (Siehe 4.2.2) öffnet die grafische Benutzeroberfläche des Compiere-Client. Sie beinhaltet eine Reihe von grafischen Elementen, die den Zugriff auf stets verfügbare Funktionalitäten ermöglicht. Diese sind innerhalb einer Toolbar bzw. innerhalb der Menüstruktur angeordnet und bieten Zugriff auf Funktionen wie Produktinformationen oder Workflow-Übersichten.

Neben diesen statischen Elementen verfügt die Oberfläche über einen Navigationsbaum, der alle dynamischen Elemente innerhalb der Client-Anwendung beinhaltet. Als dynamische Elemente werden in diesem Kontext folgende Elementtypen bezeichnet:

- `Window`
- `Form`
- `Workflow`
- `Process`
- `Task`
- `Report`

Jedes dieser dynamische Elemente besitzt eine Repräsentation vom jeweiligen Elementtyp innerhalb der Datenbank. Aus Platzgründen soll im Weiteren aber nur der dynamische Elementtyp des `Window` Betrachtung finden.

Der dynamische Elementtyp Window

Innerhalb des Navigationsbaumes können Elemente der genannten Elementtypen gruppiert und strukturiert werden. Der Baum selber kann hierbei auch an den jeweiligen Benutzer angepasst werden. Einer der quantitativ stärksten Typen innerhalb der Elementtypen ist der des `Windows`. Elemente dieses Types sind Repräsentanten dynamisch erstellter Fensterobjekte. Bei Ausführung eines dieser Elemente öffnet sich ein Fenster,

dessen Eigenschaften aus statischen und dynamischen Elementen besteht. Abbildung 4.6 verdeutlicht diese Unterteilung und zeigt die dynamischen Elemente innerhalb eines Window.

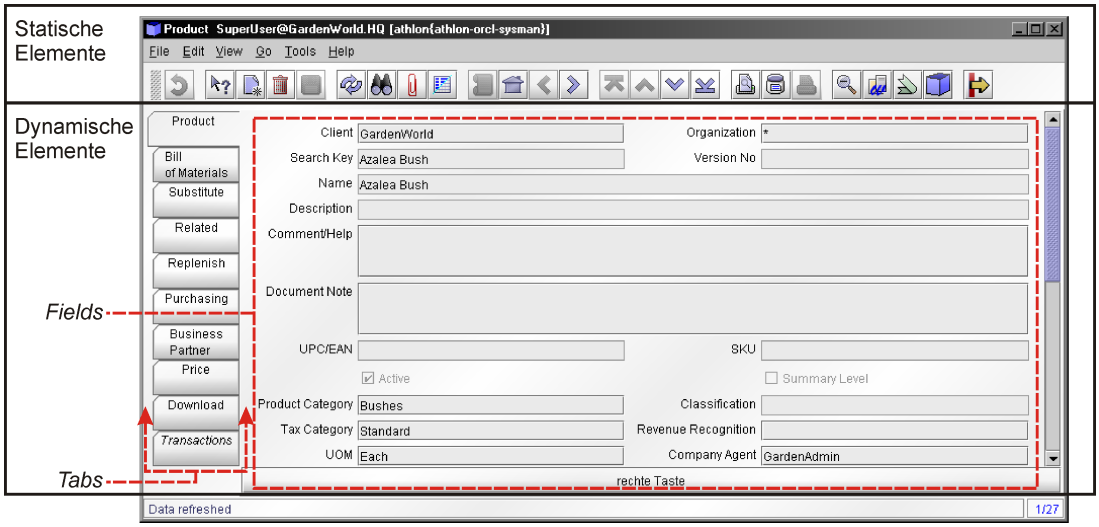

Abbildung 4.6: Screenshot der Compiere-Client-Anwendung mit Unterteilung der Elementtypen

Statische Elemente, wie die Druckfunktion, sind in jedem Element des Typs `Window` gleichermaßen vorhanden. Der dynamische Teil des `Window` umfasst dagegen variable Eigenschaften wie die Fenstergrösse oder dessen Symbol. Die wichtigsten Elemente des dynamischen Teils sind jedoch die Tabulator-Felder, die sogenannten `Tabs`. Diese bilden eine Tabelle innerhalb der Datenbank partiell oder total ab. Dabei wird jede Spalte der Tabelle mittels eines `Field` repräsentiert. Jedes dieser `Fields` entspricht einer Spalte, dem `Column`, dieser Tabelle. Je nachdem welches `Field` zu einem `Tab` zugeordnet ist wird die, dem `Field` zugehörige Spalte innerhalb des Tabulator-Feldes angezeigt. Das entsprechende grafische Element zur korrekten Anzeige des `Fields`, wie einem Text- oder Zahlenfeld, wird über den Elementtyp der vom `Field` repräsentierten Spalte ermittelt.

Jede Spalte einer Tabelle verfügt mittels der ihm zugehörigen Metadaten in der `AD_Column`-Tabelle auch über einen sogenannten `Callout`. Dieser Eintrag entspricht einem Java-Methodenpfad, wie z.B. „org.compiere.model.CalloutGLJournal.amt“. Ist solch eine Callout-Methode für eine Spalte gesetzt, so wird diese bei einer Änderungen eines Wertes dieser Spalte aufgerufen. Sie finden unter anderem bei Feldern Verwendung, die eine Änderung weiterer Felder implizieren. So kann zum Beispiel innerhalb des Bestellungs-`Window` bei Änderungen der bestellten Objekte eine Neuberechnung des Gesamtpreis-`Fields` vorgenommen werden. Jede dieser Methoden ist mit einer fest vorgegebenen Parameterliste implementiert und erlaubt damit das einfache Einbinden weiterer Callout-Methoden ohne Änderung der sie aufrufenden Objekte.

Da alle `Fields` eines `Tabs` Repräsentanten jeweils einer Spalte sind, hat jedes `Field` als

Bezug immer genau ein `Column` der Tabelle des umfassenden `Tab`. Ein Navigieren durch die einzelnen Zeilen dieser Tabelle wird durch Elemente des statischen `Window`-Bereichs ermöglicht. Der Zugriff auf jedes `Window` wird über die Tabelle `AD_WINDOW_ACCESS` geregelt. Diese enthält Zugriffsrechte für jedes Window und weist diese über Benutzerrollen zu.

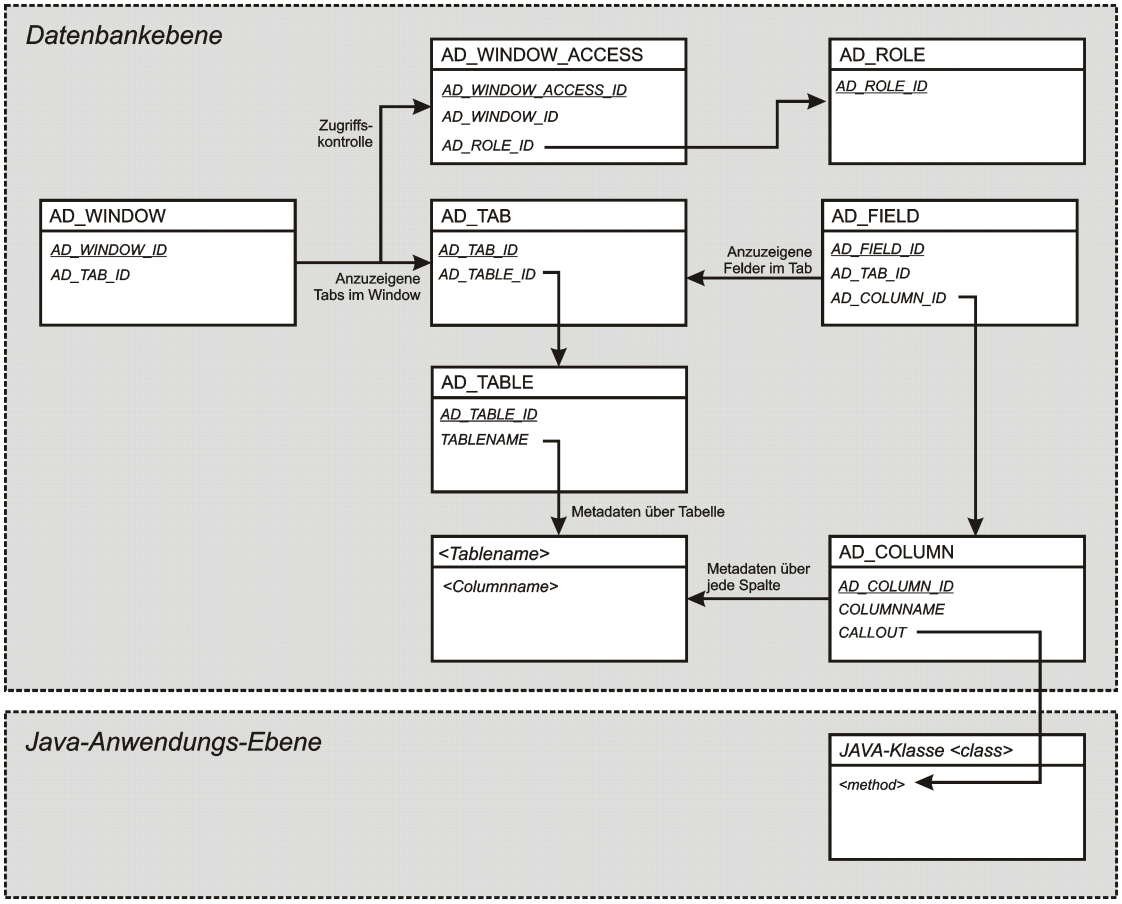

Abbildung 4.7: Datenbankstruktur zur Darstellung der dynamischen Window-Elemente

Abbildung 4.7 legt den genannten Sachverhalt grafisch dar. Zusätzlich soll an dieser Stelle noch ein kurzer Überblick über die genannten Datenbanktabellen zur Darstellung des Elementtypes Window gegegeben werden. Als Hinweis soll auch noch genannt sein, dass die abgebildeten Tabellen und Relationen zueinander aus Platzgründen nur partiell dargestellt sind. Die dargestellten Tabellen und Spalten sind jedoch für unsere Betrachtung ausreichend und sollten zum Verständniss der Funktionsweise der dynamischen Elemente beitragen. Für eine tiefergehende Studie sei hier auf weiterführende Lektüre und Dokumentationen verwiesen.

- `AD_WINDOW`

  Eine Tabelle zur Speicherung aller vorhandenen Window-Elemente. Ihre Zeilen

bilden damit die Objekte mit welchen alle Window-Elemente ihre Zugehörigkeit zu einem bestimmten Window definieren.

- `AD_TAB`

  Sie dient der Speicherung aller Tabs eines `Window`. Ihre Zeilen definieren die Reihenfolge der darzustellenden Tabs und dienen der späteren Zuweisung von Fields zu einem Tab.

- `AD_FIELD`

  Diese Tabelle speichert alle Fields, die innerhalb von Tabs dargestellt werden

- `AD_WINDOWS_ACCESS`

  Die Tabelle, welche jedem Windows ein Zugriffsrecht auf Grundlage der Benutzerrollen zuweist.

- `AD_ROLE`

  Alle Benutzerrollen werden innerhalb dieser Tabelle gespeichert.

- `AD_TABLE`

  Diese Tabelle dient der Speicherung aller Metadaten der Compiere-Tabellen.

- `AD_COLUMN`

  Sie enthält jeweils alle Metadaten der einzelnen Spalten einer Compiere-Tabelle. Anhand dieser Daten wird zum Beispiel ermittelt, von welchem Typ der Editor für das dazugehörige Feld in einem `Window` sein muss.

Die genannten Tabellen sind für den Aufbau der dynamischen *Window*-Komponenten wesentlich. Die Einbindung dieser Strukturen in die grafische Benutzeroberfläche der Hauptanwendung soll nun ansatzweise erläutert werden. (Abb. 4.8)

Bei dem Öffnen eines *Window*-Objektes wird eine `javax.swing.JFrame`-Subklasse namens `AWindow` instanziiert. Ihr wird dabei die `AD_Window_ID` aus dem darzustellenden Fenstereintrag der Tabelle `AD_WINDOW` übergeben. Im Anschluss wird ein `APanel`-Objekt generiert. Es ist eine Subklasse von `javax.swing.JPanel` und bildet den Container für alle statischen und dynamischen Elemente dieses *Window*. Im Zuge der Erstellung der dynamischen Fensterelemente wird ein `MWindow`-Objekt innerhalb dieses Panel erstellt, das die logische Kapselung aller, für die dynamischen Komponenten des *Window* relevanten, Elemente darstellt. Es beinhaltet `n` Objekte vom Typ `MTab`, welche wiederum jeweils `m` Objekte vom Typ `MField` besitzen. Diese drei Objekte dienen der grafischen Darstellung des dynamischen *Window*-Elementes und werden jeweils mit Hilfe sogenannter *Virtual Objects* erstellt.

Virtual Objects sind Objekte, die durch hartkodierte SQL-Abfragen und der Klasse `DB`, Informationen aus der Datenbank zu den von ihr abgebildeten Objekten, d.h. *Window*, *Tab* oder *Field*, beziehen. Aus diesem Grund werden diese in der Abbildung 4.8 als

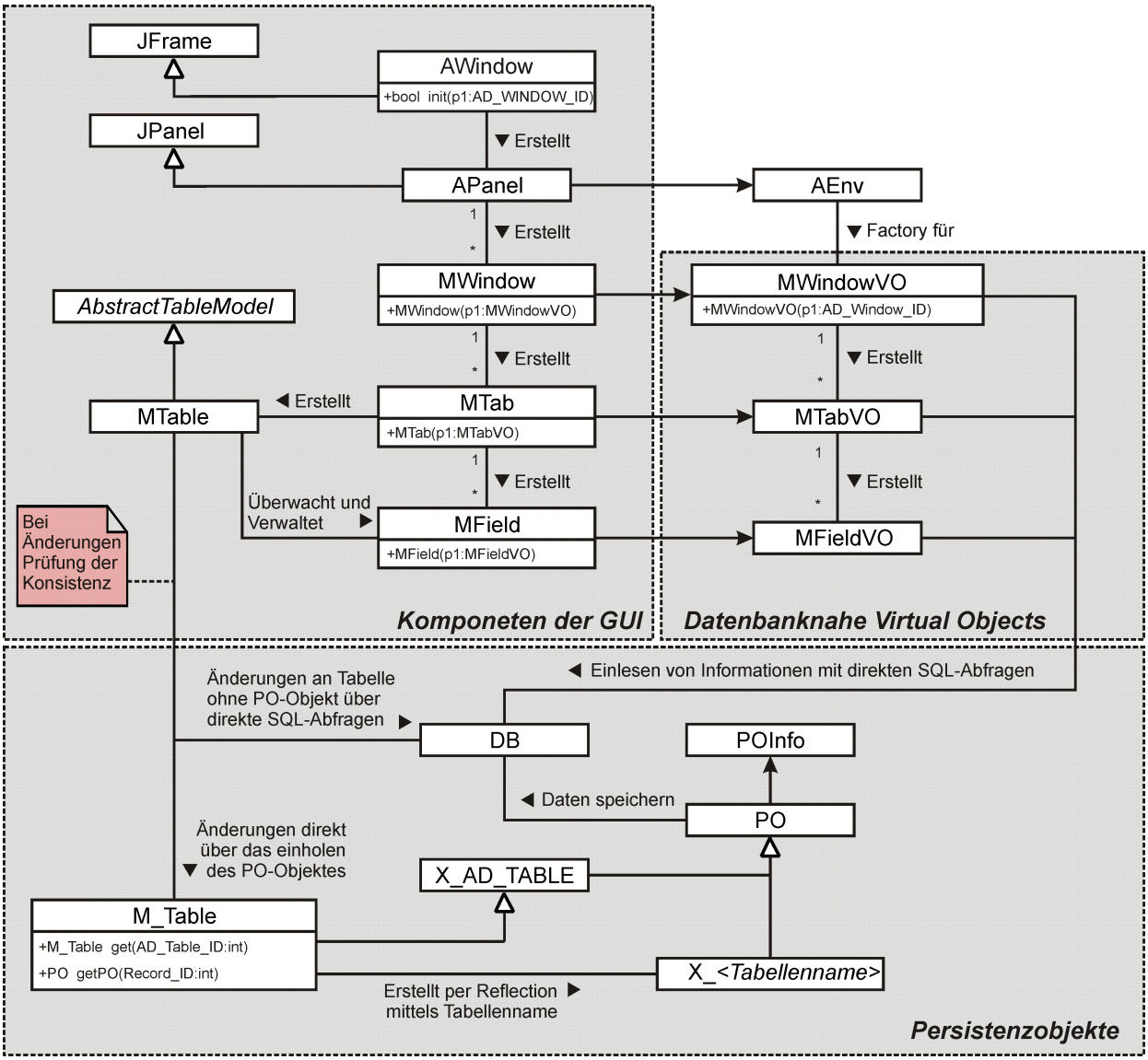

Abbildung 4.8: Integration der dynamischen Window-Elemente in die grafische Benutzeroberfläche von Compiere

Datenbanknahe Virtual Objects bezeichnet. Das Objekt `MWindowVO` wird dazu über die Klasse `AEnv` erzeugt. Dabei wird ihr wiederum die `AD_WINDOW_ID` übergeben, mit welcher sie die spezifischen Datensätze des entsprechenden `AD_Window` aus der Datenbank auslesen kann. Da bei der Erstellung dieser Klasse alle ihre beinhaltenden Objekte vom Typ `MTabVO` und `MFieldVO` erzeugt werden, dient sie als Quelle aller nötigen Objekte bei der Erstellung von `MTab` durch `MWindow` bzw. `MField` durch `MTab`.

Der schon benannte Umstand, dass *Tabs* in der GUI jeweils eine Datenbanktabelle abbilden, findet seine Entsprechung in `MTab`. Diese Klasse besitzt ein Objekt vom Typ `MTable`, einer Subklasse von `AbstractTabelModel`, welches alle von `MTab` erstellten `MFields` speichert und deren Änderungen und Überwachung ermöglicht. Des Weiteren erlaubt `MTable` ein Speichern der Datensätze unter Verwendung von Konsistenzprüfungen. Tabellen, welche Persistenzobjekte besitzen, werden dabei über die Klasse `M_Table` gespeichert. Diese generiert unter der Angabe einer `AD_TABLE_ID` bzw. `RECORD_ID` die passenden `X_<Tabellenname>`-Objekte, welche wiederum zum Speichern des Datensatzes genutzt werden. Tabellen ohne Persistenzobjekte hingegen werden durch `MTable` mittels hardcodierter SQL-Statements gespeichert. Weitere Informationen zu den Persistenzobjekten finden sich in Kapitel 4.2.4 und sollen an dieser Stelle nicht weiter erläutert werden.

In diesem Kapitel wurde mit Hilfe des Beispiels der Darstellung des `Window`-Elements gezeigt, wie stark Compiere an die Nutzung der Datenbank ausgerichtet ist. So werden eine Vielzahl dynamischer Elemente mit Hilfe der Datenbank generiert. Dies ermöglicht einem Unternehmen mit mehreren Compiere-Client-Systemen unter anderem das Ändern der dynamischen Elemente ohne das Anpassen jeder einzelnen Client-Anwendung. Um den Rahmen dieser Arbeit nicht zu sprengen, wird auf eine tiefer gehende Betrachtung der Erstellung der grafischen Oberfläche verzichtet.

Lediglich die Themen der *Datenbankanbindung* sowie des strukturellen *Datenbankaufbaus* werden in den Kapiteln 4.2.3 bzw. 4.2.4 weitere Erwähnung finden.

5 Aspektorientierte Softwareentwicklung

Ein wesentlicher Schritt zur Verbesserung der Softwareentwicklung war das Aufkommen des Paradigmas der *Objektorientierten Programmierung* (OOP) Mitte der 80'er Jahre des vergangenen Jahrhunderts. Trotz des verwendeten Objektmodells, das durch Vererbung und Komposition eine verbesserte Anpassung an reale Probleme erlaubte, wurden auch hier schnell Anforderungen erkannt, die durch OOP nicht vollständig implementiert werden konnten.

Zusammen mit den Anforderung nach steigender Komplexität wuchsen dadurch die akademischen- und industriellen Bestrebungen für den Entwurf eines neuen Programmierparadigmas. Aus diesen Forschungsaktivitäten entwickelte sich das Konzept der *Aspekte*, welche die Elemente der Anforderungen bezeichnen, die eine Überschneidung (*cross-cut*) mit den Basisfunktionalitäten darstellen. Sie bilden die Komponenten, bei denen eine saubere Integration in das entwickelte System mittels der bisherigen Programmiermodelle Probleme bereitet. Typischerweise sind diese überschneidenden Komponenten Elemente nicht-funktionaler Anforderungen wie z.B. die Synchronisation, Fehlerbehandlung oder das Logging. Aus dieser Problemstellung heraus entstand die *Aspektorientierte Programmierung* (AOP). Das Ziel war, mit ihr eine verbesserte Integration von Aspekten zu erreichen, die dazu auf den Konzepten der OOP aufbaut und diese erweitert.[KLM+97] Das Einbeziehen der AOP-Konzepte in die einzelnen Phasen der gesamten Softwareentwicklung, wie Analyse, Modellierung oder Implementierung, führt zu einer ganzheitlichen Ausrichtung an den Konzepten des AOP. Sie wird als *Aspektorientierte Softwareentwicklung* (Aspect-Oriented Software Development - AOSD) bezeichnet.

Die folgenden Abschnitte geben eine Übersicht über die verwendeten Begriffe im Kontext der Aspektorientierten Programmierung. Anschließend werden die für diese Arbeit relevanten aspektorientierten Programmierkonzepte vorgestellt.

5.1 Begriffsklärung und allgemeine Übersicht

Im Zuge der Einführung des neuen Paradigmas der Aspektorientierten Programmierung wurden neue Begriffe eingeführt, die in diesem Abschnitt kurz beschrieben werden sollen. Im Zuge einer problembezogenen Herangehensweise soll hier zuvor jedoch die Problematik hinter der Entwicklung der Aspektorientierung erläutert werden.

5.1.1 Problematik

Die objektorientierte Softwareentwicklung basiert auf dem grundlegenden Konzept der Modularisierung durch Objekte. Diese kapseln Elemente und Funktionalitäten des betrachteten Anforderungsraumes und schaffen so eine Abstraktion, die bei der strukturellen Modellierung hilft. Unterstützend wirken dabei die objektorientierten Konzepte der Vererbung und der Komposition ein. Jedoch erreicht man auch hier Grenzen, bei denen sich nicht alle Anforderungen sauber getrennt modellieren und implementieren lassen.

Crosscutting Concerns

Der allgemeine Designprozess in der objektorientierten Softwareentwicklung ist gekennzeichnet durch die Aufspaltung des Systems und seiner Anforderungen in die funktionalen Bestandteile. Die Anforderungen werden hierbei auch als *concerns* bezeichnet. Es gibt jedoch Anforderungen, welche innerhalb der, in diesem Prozess erstellten,Klassen-Hierarchie nicht modular implementiert werden können.

Concerns sind bei den nicht-funktionalen Anforderungen häufig innerhalb der Persistenz- und Transaktionseigenschaften, der Verteilung oder dem Logging zu lokalisieren. Beispiele für *Concerns* innerhalb funktionaler Anforderungen sind dagegen z.B. Funktionalitäten zum Drucken oder zur Visualisierung. In der Regel sind die *Concerns* in der Analysephase noch isoliert, verteilen sich dann aber in der Implementierungsphase über mehrere Klassen und schwächen somit die angestrebte Modularisierung. [Mos03]

Durch diese Verteilung der Concerns über mehrere Klassen wird die Modularitätsstruktur quasi durchschnitten, weshalb man diese auch *Crosscutting Concerns* nennt. Abbildung 5.1 verdeutlicht diese horizontale Lage der *Crosscutting Concerns* gegenüber der vertikalen Modularitätsstruktur.

Crosscutting Concerns äussern sich in zwei typischen Erscheinungsformen:

- *Scattering* (Verstreuung)

  Funktioneller Code verstreut sich über mehrere Klassen in Form von Redundanz und resultiert in folgenden Symptomen [DJS04]:

  - redundanter Code
  - unsauberere Struktur
  - erschwerte Pflege des Code
  - irrelevanter Code in Bezug zur umschließenden Klasse, wie z.B. Logging-Methoden

- *Tangling* (Verflechtung)

  Zwei oder mehr *Concerns* überlappen sich in einer Klasse und schwächen damit die angestrebte Modularisierung.

Beide Formen erschweren die Pflege des Code, da bei Änderungen der Code beim *Scattering* an mehreren Stellen angepasst werden muss bzw. beim *Tangling* eine Ent- und anschließende Verflechtung (un- und retangling) erforderlich ist.

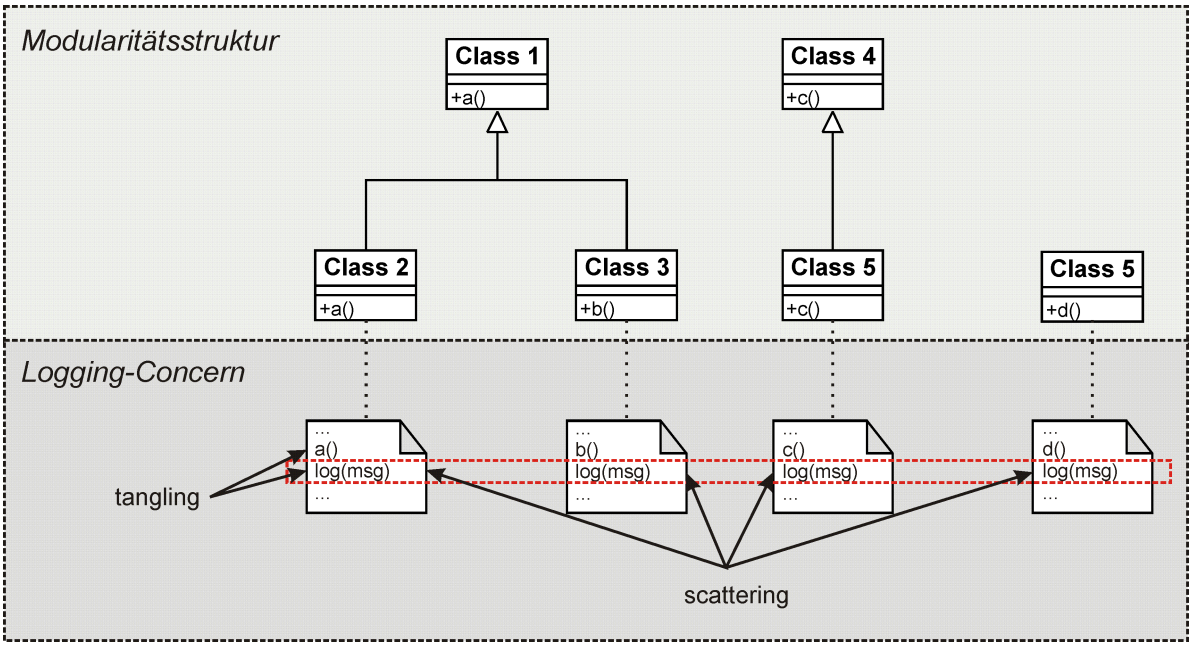

Abbildung 5.1: Scattering und Tangling innerhalb eines moduldurchschneidenden concern

Crosscutting Concerns werden häufig schon in einer frühen Phase innerhalb des Designprozesses identifiziert. Das Fehlen einer allumfassenden Lösung, des sprichwörtlichen *silver bullet*, erschwert jedoch das Vermeiden dieser Überschneidungen in der objektorientierten Programmierung.

Die Verwendung von Design Patterns, wie dem *Visitor* oder *Observer*, liegt oft begründet in dessen Fähigkeit, *Crosscutting Concerns* separieren zu können. Design Pattern helfen damit beim verringern von sich verflechtenden Code, dem *Tangling*. Die Grenzen der praktischen Anwendbarkeit von Design Patterns begrenzen jedoch den Grad an Nutzbarkeit im Sinne der Minimierung von *Crosscutting Concerns*.

Einen effizienteren Lösungsansatz zur Vermeidung von *Crosscutting Concerns* bietet die Aspektorientierung. Sie hilft bei der sauberen Trennung der einzelnen *Concerns* über alle Phasen der Softwareentwicklung hinweg. Ziel dieser sogenannten *Seperation of Concerns* ist die getrennte Definition, Modellierung und Implementierung der *Concerns*.

Eine passende Beurteilung der Aspektorientierten Programmierung und dem *Seperation of Concerns* gibt Laddad mit den Worten [Lad03]:

> „AOP marks the beginning of new ways of dealing with a software system

by viewing it as a composition of mutually independent concerns.“

5.1.2 Elemente der Aspektorientierung

Die aspektorientierte Programmierung eröffnet eine neue Ebene der Abstraktion, sowohl in der Designphase als auch in der Implementierung selbst. Die Abstraktion der modularisierten Concerns findet dabei über *Aspekte* statt. Sie können auf unterschiedliche Weise spezifiziert werden. Die verwendeten Techniken der vorliegenden Arbeit setzen so z.B. auf Klassen, welche durch spezielle Bezeichner als Aspekte markiert werden. Andere AOP-Implementierungen verwenden an dieser Stelle auch Metadateien, welche Aspekte in Form einer Aspektsprache definieren und so die Verbindung zwischen Aspekt- und Basisklasse herstellen. Als *Basisklasse* bezeichnet man in diesem Zusammenhang die Klasse, welche durch einen Aspekt erweitert bzw. dessen Code durch einen Aspekt ausgelagert wird. Sie beinhaltet die Elemente, auf die sich ein Aspekt bezieht.

In [KLM+97] werden die Basiselemente einer aspektorientierten Implementierung wie folgt zusammengefasst:

> „(a) a component language with which to program the components, (b) one or more aspect languages with which to program the aspects, (c) an aspect weaver for the combined languages, (d) a component program, that implements the components using the component language, and (e) one or more aspect programs that implement the aspects using the aspect languages.“

Den Vorgang des Einbindens bzw. Ausführens des Aspektcodes an den jeweiligen Stellen im Klassencode nennt man *Weben* oder *Aspect Weaving*. Das Programmmodul zur Durchführung dieses Webeprozesses nennt man *Weaver*. Dessen Implementierung hängt von der jeweiligen AOP-Implementierung ab und kann unterschiedliche Methoden der Verwebung bieten. Diese werden an dieser Stelle nur im Rahmen der, für die vorliegende Arbeit relevanten AOP-Implementierungen betrachtet, und sind in den folgenden Abschnitten dieses Kapitels zu finden. Ein allgemeiner Überblick über die Möglichkeiten des *Aspect Weaving* wird in Abschnitt 5.1.3 gegegeben.

Im Folgenden werden die Elemente betrachtet, die im Kontext der Aspektorientierung Anwendung finden:

Joinpoint

Als *Joinpoint* werden die Punkte im Code der Basisklasse bezeichnet, an denen der Aspekt Einfluss ausübt. Die Möglichkeiten zur Definition von *Joinpoints* hängt dabei von der AOP-Implementierung ab. Sie bieten aber für gewöhnlich mindestens Definitionen folgender Formen an:

- *Klassenkonstruktor- und Methodenausführung*

Dies erlaubt das Festlegen eines Joinpoints beim Ausführen einer bestimmten Methode oder Konstruktors innerhalb einer Klasse.

- *Zugriff auf Attribute*

  Der Zugriff auf Attribute einer Klasse kann als Joinpoint dienen, wobei zwischen lesendem und schreibendem Zugriff differenziert wird.

- *Kontrollfluss*

  Der Kontrollflussgraph der Anwendung kann hierbei zur Definition von Joinpoints genutzt werden. Sie werden dabei an bestimmte Ereignisse oder Zustände innerhalb des Kontrollflusses gebunden. Mehrere dieser Kontrollflusselemente können akkumuliert werden zu einem sogenannten *cflow*. Mit dessen Hilfe lassen sich Abfolgen von Ereignissen und Zuständen innerhalb des Kontrollflusses als ein Joinpoint zusammenfassen.

Die spezifischen Joinpoint-Möglichkeiten der für diese Arbeit relevanten AOP-Implementierungen, werden im Anschluss in den jeweiligen Kapiteln beschrieben.

Pointcut

Ein Pointcut ist die formale Beschreibung eines oder mehrerer Joinpoints innerhalb des Programmflusses. Die Art der Definition sowie dessen Syntax ist abhängig von der AOP-Implementierung und wird in den jeweiligen Kapiteln Erwähnung finden.

Advice

Als Advice wird der Code bezeichnet, der mit der Basisklasse verwoben werden soll.

Aspektbindung

Die zusammengeführten *Pointcuts* und *Advices* nennt man Aspektbindungen. Typischerweise sind diese Bestandteil einer gemeinsamen Aspektklasse. Alternativ dazu können sie jedoch auch, je nach AOP-Implementierung, voneinander getrennt definiert werden.

Introduction

Als Introduction bezeichnet man das Hinzufügen eines neuen Verhaltens zu den Basisklassen mit Hilfe von Aspektklassen. Es ist dadurch zum Beispiel möglich, alle Basisklassen um ein Attribut zu erweitern, ohne die Basisklassen selbst ändern zu müssen. Dabei wird das Attribut innerhalb einer Aspektklasse eingeführt, welche auf alle diese Basisklassen Einfluss ausübt.

Die Implementierung jedes dieser Elemente hängt von der verwendeten Aspektsprache ab und kann dazu noch zusätzliche Elemente beinhalten. In den Kapiteln 5.2 und 5.3 werden diese für die in dieser Arbeit relevanten AOP-Implementierungen näher erläutert.

5.1.3 Verwebung

Das in diesem Kapitel vorgestellte Konzept des *Separation of Concerns* erfordert Techniken zur Integration der ausgelagerten *Concerns* in den ursprünglichen Basiscode. Dieser Prozess, den man *Verwebung* (Aspect Weaving) nennt, kann zu verschiedenen Zeitpunkten und in verschiedenen Entwicklungskontexten ausgeführt werden. Diese sollen hier nun kurz beschrieben werden.

- *Compile-Time*

  Das Verweben findet hierbei zum Zeitpunkt des Kompilierens statt und wird aufgrund der dadurch erzwungenen Unveränderbarkeit des Verhaltens auch *Static Weaving* genannt. (Abb. 5.2) Ein spezieller Compiler generiert hierbei Klassen, welche bereits den Aspektcode beinhalten und somit keine weitere Änderung zur Laufzeit erfahren müssen. Dies erspart den Aufwand weiterer Verwebungen zur Lade- bzw. Laufzeit und ist damit effizienter in der Ausführung. Einmal kompiliert, bleiben die verwobenen Aspekte dadurch unveränderbarer Bestandteil des kompilierten Code. Eine benötigte Änderung der Aspekte erfordert somit eine Neukompilierung der gesamten Basis- und Aspektklassen.

  Aufgrund dieser statischen Beschränkungen, obliegt es dem Programmierer selbst alle nötigen Basisklassen vor der Verwebung zu identifizieren. Basisklassen, die erst zur Laufzeit dynamisch hinzukommen, können so bei einer statischen Verwebung nicht mit dem Aspektcode verwebt werden. Sie müssen dem Programmierer schon vor der Kompilierung bekannt sein.

  Compile-Time-Weaving wird daneben noch unterschieden in *Sourcecode-Weaving* und *Binärcode-Weaving*, bzw. *Bytecode-Weaving* innerhalb der Programmiersprache Java. *Sourcecode-Weaving* bezeichnet dabei das präprozessuale Verweben des Basiscodes mit dem Aspektcode, noch bevor ein Compiler den erzeugten Code kompiliert. Nachteilig ist hierbei die Gebundenheit an die Verfügbarkeit des Sourcecode, die eine Erweiterung geschlossener Klassen unmöglich macht. *Bytecode-Weaving* ist das postprozessuale Verweben des Aspektcodes in den schon kompilierten Bytecode der Basisklasse. Im Gegensatz zum *Sourcecode-Weaving* erlaubt dieser damit das Verweben mit vorkompilierten Komponenten. Beide Varianten sind unter anderem in AspectJ, das im Abschnitt 5.1.5 noch näher betrachtet wird, verfügbar. Neben dem Java-spezifischen *Bytecode-Weaving* gibt es diese Technik auch für andere Programmiersprachen. So wird in [SGSP02] eine AOP-Implementierung mit *Compile-Time-Weaving* für die objektorientierte Programmiersprache C++ vorgestellt.

- *Load-Time*

  Der Aspektcode wird hierbei erst zur tatsächlichen Ladezeit des Basiscode hinzugewebt. Dadurch wird ein einfacher Austausch von Verhalten über die jeweiligen Aspektobjekte ermöglicht. Sie erfordert jedoch auch Aufwand, der sich negativ auf die Geschwindigkeit der Applikation auswirkt. Ein spezieller Classloader transformiert hierbei den Basiscode beim Laden der Basisklasse. Der Basiscode selbst liegt

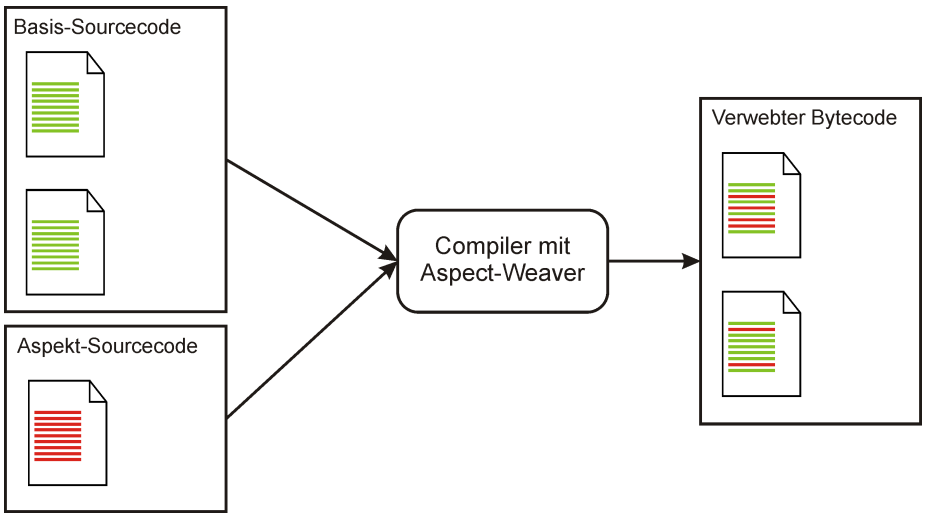

Abbildung 5.2: Aspect-Weaving zur Kompilierzeit

dazu in vorkompilierter Form vor, z.B. als Java-Bytecode. Er kann somit von einem beliebigen Compiler generiert worden sein, solange dieser nur Standard-Bytecode erzeugt.

JavaAssist, *AspectWerkz* und *JMangler* sind Beispiele für AOP-Implementierungen auf Basis von Java und des *Load-Time-Weaving*. Eine grafische Darstellung dieser Technik gibt Abbildung 5.3.

- *Run-Time*

  Eine Verwebung zur Laufzeit der Applikation ist von den genannten Verfahren die flexibelste. Sie verwebt den Aspektcode direkt mit dem Basiscode während dieser ausgeführt wird. Dies gestattet eine dynamische Änderung der Aspekte zur Laufzeit der Applikation. Der Nachteil dabei ist der zusätzlich anfallende Aufwand zur Überwachung des Kontroll- und Datenflusses, der in Performanzeinbußen resultiert.[1] Das *Run-Time-Weaving* wird in Form zweier Typen unterschieden.

  Das *Pseudo-Runtime-Weaving* ist ein Konzept, bei welchem die Applikation zur Laufzeit überwacht wird. Dies kann mittels einer, bei der Kompilierung oder während des Ladevorgangs durchgeführten Instrumentalisierung zur Überwachung oder durch die Verwendung von Debugging-Methoden implementiert werden. Beide Techniken sind aufwendig, da sie bei jedem instrumentiertem Punkt prüfen, ob an dieser Stelle ein Aspekt greift oder nicht.

  Der zweite Typus ist das *Generic-Runtime-Weaving*. Hier wird die gesamte Laufzeitumgebung, wie die *Java Virtual Machine*, ausgetauscht oder durch ein Plugin erweitert. Die verbesserte Geschwindigkeit gegenüber dem *Pseudo-Runtime-Weaving* wird hier durch die Bindung an eine angepasste Laufzeitumgebung erreicht. Nachteilig wirkt sich dies jedoch auf die praktische Anwendbarkeit dieser

[1] Bei *PROSE* liegt der Verlust gegenüber der Nichtverwendung bei ungefähr 10% [NA05]

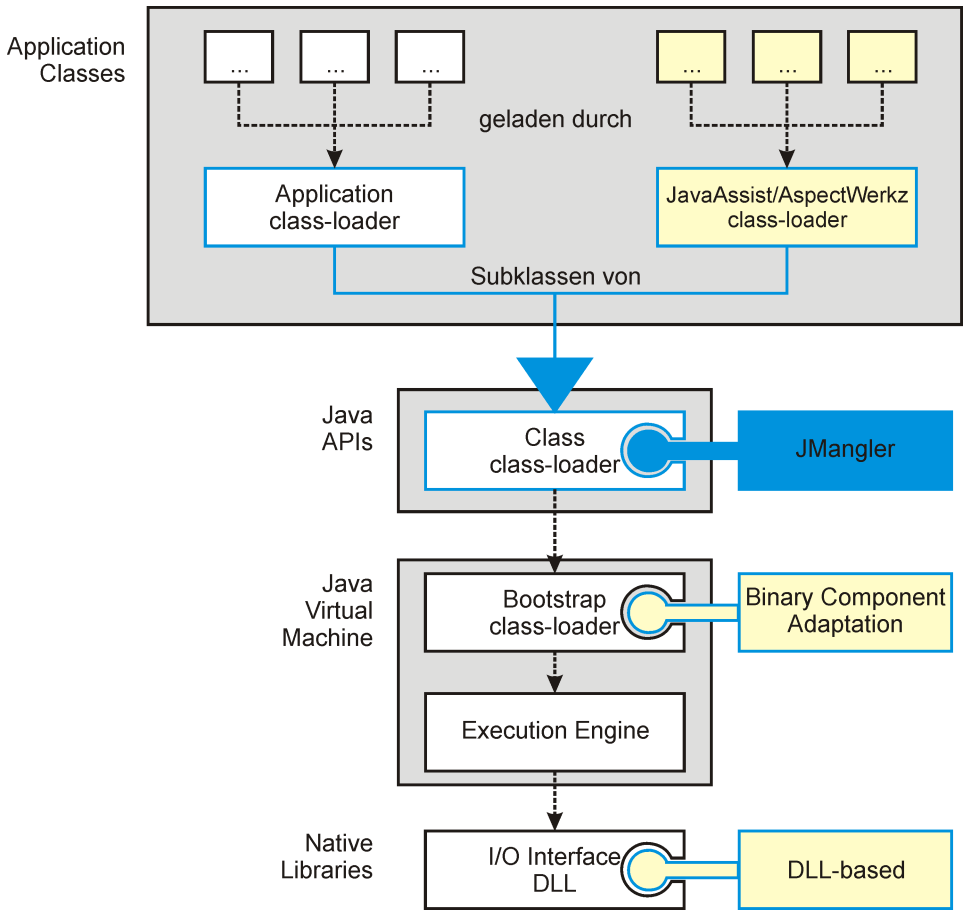

Abbildung 5.3: Aspect-Weaving zur Ladezeit am Beispiel von JMangler (nach [KCA04])

Technik aus. So muss auf jedem Zielsystem die Laufzeitumgebung entsprechend angepasst werden. Neuere Laufzeitumgebungen bieten aus diesem Grund hier teilweise schon Unterstützung. So ist ab Java 1.4 eine, *HotSwap* genannte Technik zum Austausch von Code während der Laufzeit der Applikation implementiert. Mit dieser Technik entfällt das, mit Nachteilen behaftete Austauschen der gesamten Laufzeitumgebung. Der sich negativ auswirkende Performanzunterschied zum `Load-` und *Compile-Time-Weaving* bleibt nichtsdestotrotz bestehen. [CST03]

Beispiele einer AOP-Implementierung unter Verwendung von *Run-Time-Weaving* sind *Steamloom*[2], *PROSE*[3] und *AspectC++*, welches in [SGSP02] vorgestellt wird.

5.1.4 Die Vor- und Nachteile der Aspektorientierung

Ein Hemmfaktor für das Etablieren eines neuen Paradigmas in der Programmierung ist die Hürde des Erlernens. Abgesehen von der Zeit, welche das Verinnerlichen und

[2] http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp
[3] http://prose.ethz.ch

Meistern der Idee der Aspekte erfordert, gibt es weitere Faktoren, die sich negativ auf den Entwicklungsprozess auswirken können. Die durch AOP gewonnen Vorteile sollen in diesem Abschnitt mit den Nachteilen verglichen werden und einen Überblick über die Möglichkeiten und Grenzen der Aspektorientierung aufzeigen. [Lad03]

Einige der Vorteile bei der Nutzung von AOP sind:

- *Stärkung von „Design by Contract“*

  Das modularisierende Auslagern von *Crosscutting Concerns* verringert den Anteil des nicht zum Modulkern gehörenden Code und beschränkt dadurch die Verantwortlichkeiten der Module auf deren Kernfunktionalitäten. Dies erhöht die Nachverfolgbarkeit von Fehlerzuweisungen und stärkt damit das Konzept von *Design by Contract*. Problematisch kann jedoch die Definition von Contracts zwischen Aspekt- und Basisklassen selbst werden, da diese nicht immer leicht voneinander abzugrenzen sind.

- *Verbesserung der Modularität*

  Ein hoher Grad an Modularität verringert redundanten Code und ermöglicht so das Ändern einzelner Module, ohne den Code an weiteren Modulen anpassen zu müssen.

- *Erleichterung von Evolutionsschritten*

  Die Trennung der Kernmodule von den Aspekten erhöht deren Unabhängigkeit. Zukünftige Erweiterungen können mittels Hinzunahme oder Änderung der Aspektklassen vorgenommen werden, ohne die Kernmodule ändern zu müssen. Damit wird eine flexiblere Anpassung an neue Anforderungen ermöglicht.

- *Spätes Design*

  Das erleichterte Hinzufügen von Funktionalitäten erlaubt die Konzentration auf Kernfunktionalitäten in der Entwurfsphase. Alle weiteren *Crosscutting Concerns* können anschließend über Aspekte hinzugefügt werden.

- *Erleichterte Wiederverwendung*

  Durch die schwache Kopplung von Aspekten und Basisklassen wird die Wiederverwendung des Aspektcodes erleichtert. Das Austauschen von Aspekten wird dabei durch eine Änderung der *Advice*-Spezifikationen ohne weitere Codeänderungen ermöglicht.

Die genannten Vorteile führen zu kürzeren Zyklen der Entwicklung und vermindern die Entwicklungskosten. Ihnen gegenüber stehen Eigenschaften, die sich auf die Softwareentwicklung negativ auswirken können:

- *Schwer nachzuvollziehender Kontrollfluss*

Die, in Abschnitt 5.1.1 beschriebene, horizontale Dimension der Aspekte in Verbindung mit der vertikalen Dimension der Modellhierarchie der Basisklassen erschwert das Analysieren des Kontrollflussgraphen und erhöht den Komplexitätsgrad des Gesamtmodells. Einige Entwicklungsaktivitäten, wie das Debuggen oder generelle Verstehen der Programmausführung werden damit komplexer und verringern die zeitliche Effizienz der Gesamtentwicklung.

- *Schwächung der Kapselung*

  In der Objektorientierten Programmierung kapselt eine Klasse die gesamte Struktur und Funktionalität eines Objektes. AOP schwächt diese Kapselung durch das Auslagern von Verhalten in Aspekte, wenn auch in einer nachvollziehbaren und kontrollierten Art und Weise.

- *Erschwertes Testen*

  Unit-Tests bei OOP-Anwendungen betrachten die Basisklasse als zu testendes Objekt. Das Aufspalten von *Concerns* in Basis- und Aspektklassen führt zu getrenntem Verhalten, welches erst zum Zeitpunkt des Verwebens das zu testende Zielverhalten erzeugt. Aus diesem Grund erfordert das Testen von AOP-Anwendungen besondere Methodiken. Diese Problematik ist Thema aktueller Forschungen. Ein Ansatz verwendet dabei z.B. Kontroll- und Datenflussgraphen zur Verfolgung des Programmverhaltens, mit deren Hilfe dann verschiedene Tests durchgeführt werden können [RB04] [ZRZ04]. Hierbei gibt es verschiedene Stufen der Einbeziehung der Aspekte, angefangen vom einfachen Testen der Basisklasse bis hin zum Testen der Basisklasse inklusive aller im System vorhandenen Aspekte. Der Aufwand innerhalb der Testaktivitäten wird so durch Aspekte noch erhöht und bildet eine erschwerende Komponente bei der Verwendung von Aspektorientierung in der Softwareentwicklung.

Das Konzept der *Seperation of Concerns* ist die Kernidee hinter dem Paradigma der Aspektorientierung. Es bietet vielversprechende Vorteile, wie erleichterte Wartung und Verständlichkeit, wodurch die genannten Nachteile in vielen Anwendungsfällen zu rechtfertigen sind. Der Umfang der *Crosscutting Concerns* entscheidet jedoch hierbei über die positive Anwendbarkeit von AOP. Letztendliche ist aber eine Entscheidung zugunsten der Benutzung von AOP nicht absolut zu treffen, sondern vielmehr individuell für jedes Projekt vorzunehmen.

5.1.5 AOP-Implementierungen für Java

Im Folgenden sollen einige der vorhandenen aspektorientierten Ansätze kurz vorgestellt werden.

- *AspectJ*[4]

  AspectJ ist eine aspektorientierte Erweiterung für Java, das als erste AOP-Implementierung überhaupt im Rahmen eines Forschungsprojektes durch Gregor Kiczales entstand. Hierbei werden Aspekte innerhalb eigener Java-Klassen gekapselt und mit AspectJ-Sprachdefinitionen angereichert. Ein spezieller AspectJ-Compiler ermöglicht das Verweben mit den Basisklassen und erzeugt reguläre Java-Byte-Code-Klassen, die von jeder Java-VM ausgeführt werden können. Die Aspektklassen beinhalten dazu die Elemente der dynamischen Crosscuttings, d.h. *Pointcuts* und *Advices*, sowie der statischen Crosscuttings, den *Introductions*. Sie werden zum Zeitpunkt des Kompilierens mit den Basisklassen verwoben. Ab dem Release 1.1 unterstützt AspectJ des Weiteren auch das Verweben zur Laufzeit. [Aspa][Lad03]

- *AspectWerkz*[5]

  Das Open-Source-Projekt AspectWerkz [6] benutzt ByteCode-Änderungenm um eine Verwebung zur Kompilier-, Lade- und Laufzeit zu ermöglichen. Aspekte, Advices und Introductions werden hierbei in Standard-Java-Klassen geschrieben, wobei die Aspektbindungen über Annotationen[7], Doclets oder XML-Dateien definiert werden. Seit dem Januar 2005 arbeitet das Projekt im Verbund mit AspectJ zusätzlich an einer gemeinsamen AOP-Plattform. [Aspb]

- *Composition Filters*

  Das Konzept der *Composition Filter* erweitert das herkömmliche Objektmodell um eine Reihe von sogenannten *Message Filter*. Diese umschließen Objekte und filtern deren ein- und ausgehende Nachrichten. Sie können auf den Nachrichten verschiedene Aktionen ausführen, wie Löschen, Verändern oder Umleiten, und auch selber Nachrichten senden. Sie ermöglichen damit eine Verhaltensänderung bei Nachrichtentransfers, wie Methodenaufrufe auf Objekten und können so *Concerns* ausserhalb der Objekte kapseln. *ComposeJ* und *ConcernJ* sind Beispiele einer Java-basierenden Implementierung des Konzeptes der *Composition Filters*. [BA01]

- *JAC (Java Aspect Components)*[8]

  Dieses Projekt des *ObjectWeb-Consortium* ist ein Open-Source-AOP-Framework in Form eines nicht J2EE-zertifizierten Applikation-Server. Dieser ersetzt EJBs durch *Plain Old Java Object* (POJOs) und bietet eine Vielzahl von Möglichkeiten zur Nutzung von Aspekten innerhalb des Containers. JAC unterstützt ein dynamisches Verweben bzw. Entweben und erlaubt damit das Hinzufügen bzw. Entfernen von Aspekten zur Laufzeit der Applikation.

- *JMangler*[9]

[4] Entwickelt im Xerox Palo Alto Research Center - http://www.eclipse.org/aspectj
[5] http://aspectwerkz.codehaus.org
[6] Lizenziert unter der *Lesser General Public License* (LGPL)
[7] Annotationen wurden erst ab Java 5 eingeführt
[8] http://jac.objectweb.org
[9] http://roots.iai.uni-bonn.de/research/jmangler

JMangler ist ein Framework, das Elemente einer Java-Applikation während ihrer Ladezeit adaptieren kann. Durch die Benutzung des *Generic Interception* ist die Adaption unabhängig von dem verwendeten Classloader und kann dadurch auch in komplexeren Umgebungen, wie einem Applikation-Server, angewandt werden. Die offene Architektur ermöglicht eine Integration von individuellen Webeverfahren und begründet die Existenz einer Vielzahl von AOP-Implementierungen auf Basis von JMangler. Zur Transformation werden hierbei nur die kompilierten Basisklassen benötigt. Diese werden beim Laden durch JMangler mittels *Transformer-Komponenten* und *Composition specifications* transformiert und mit dem Aspektcode verwebt. [KCA04]

- *Hyper/J*[10]

  Hyper/J ist ein Java-basierter Ansatz für ein, *Hyperspace* genanntes, *Multi-dimensional Separation of Concerns*. Es erlaubt die Identifikation, Kapselung und Integration mittels einer multi-dimensionalen Komposition von Concerns. Die Concerns sind innerhalb einer *Concern Matrix* angeordnet, wobei jede Achse der Matrix einer Dimension der Concerns entspricht. Ein *Concern Mapping* spezifiziert, wie die einzelnen Concerns innerhalb der *Concern Matrix* angeordnet sind.

  Sogenannte *Hyperslices* kapseln Concerns und werden in *Hypermodules* zu Integrationsblöcken zusammengefasst, die der späteren Komposition dienen. Diese Verwebung findet bei Hyper/J zum Zeitpunkt der Kompilierung statt und ist damit statisch. Als Spezifikationsmittel der genannten Hyper/J-Komponenten dient eine spezielle Hyper/J-Sprache, wobei die Concerns mittels Standard-Java implementiert werden. [OT00]

Des Weiteren sind an dieser Stelle noch die Entwicklungen *JBoss AOP* und *Object Teams* zu nennen. Da diese im Rahmen dieser Arbeit von besonderem Interesse sind, werden sie in gesonderten Abschnitten noch tiefergehend Beachtung finden.

5.2 JBoss AOP

Der JBoss-Applikation-Server, im Weiteren vereinfachend auch nur JBoss genannt, ist ein Open-Source-Projekt[11] aus der Produktreihe des Softwareherstellers JBoss Inc.[12]. Im Jahre 2004 wurde JBoss ab der Version 4.0 durch Sun Microsystems als J2EE 1.4 Applikation-Server zertifiziert. Neben dieser Konformität zur J2EE-Spezifikation erfuhr der JBoss ab der Version 4.0 weitere funktionelle Erweiterungen. Die Einbindung des Konzeptes der Aspektorientierung war eines davon. Diese neue Komponente ist in Form eines zusätzlichen Frameworks in den Applikations-Server integriert und wird bezeichnet

[10] Entwickelt am *IBM's Thomas J. Watson Research Center* - http://www.alphaworks.ibm.com/tech/hyperj
[11] Lizenziert unter der *Lesser General Public License* (GPL)
[12] http://www.jboss.org

als *JBoss AOP*. Abschnitt 5.1.2 gab einen Überblick über die typischen Elemente der Aspektorientierung. Darüber hinaus besitzt JBoss-AOP einige spezielle Elemente und Charakteristika, welche Inhalt der folgenden Abschnitte sein werden.

5.2.1 Das Pointcut-Modell

Wie im vorherigen Abschnitt 5.1.2 schon erläutert, sind *Pointcuts* formale Beschreibungen von Zuständen im Programmfluss, an denen Änderungen im Verhalten durch Wirken von Aspektcode stattfinden. Der Aspektcode wird zu diesem Zweck in Aspektklassen definiert. Diese verwenden reines Java und benötigen daher keinen besonderen Compiler. Die gesamte logische Verknüpfung von Pointcuts und Aspektklassen wird über XML-Deskriptoren innerhalb einer Datei namens `jboss-aop.xml` definiert.

Listing 5.1: Beispiel eines JBoss-AOP XML-Deskriptor zur Definition von Pointcuts und Interceptors

```
1 <aop>
2   <aspect class="beansAOP.FacadeAdvice"/>
3
4   <bind pointcut="execution(public void beans.FacadeBean->methodeA())
        ">
5     <advice aspect="beansAOP.FacadeAdvice" name="methodeAAdv"/>
6   </bind>
7 </aop>
```

Abbildung 5.1 zeigt an einem Beispiel den Inhalt des XML-Deskriptor innerhalb der Datei `jboss-aop.xml`. Zeile 2 verdeutlicht, wie Aspektklassen geladen werden. In diesem Fall wird die Aspektklasse `FacadeAdvice` des Package `beansAOP` geladen. Sie beinhaltet den zu verwebenden Code, den Advices, in Form von Methoden. Jedes dieser Methoden wird dabei jeweils genau ein Objekt übergeben, welches von der Klasse *Invocation* erbt und dem aufrufenden Pointcut-Typus entspricht. Diese Objekte ermöglichen dem Advice Zugriff auf das ursprüngliche Objekt, wie eine aufgerufene Methode oder ein Attribut, und gestattet auch das Fortschreiten im Programmfluss an der ursprünglich unterbrochenen Stelle.

Listing 5.2 gibt ein Beispiel einer Aspektklasse mit zwei Advices. Mittels overloading werden hier die unterschiedlichen Joinpoints durch ihre jeweiligen, dem Advice übergebenen Invocationobjekte unterschieden. In Zeile 4 des XML-Deskriptors wird der Pointcut mit dem Schlüsselwort `Pointcut=` definiert und beschreibt in diesem Beispiel eine Verwebung zum Zeitpunkt der Ausführung der Methode `beans.FacadeBean.methodeA()` mit Hilfe das Schlüsselwortes `execution`. Die auszuführende Advice-Methode wird innerhalb der Pointcut-Definition in einem `advice`-Block mit dem Schlüsselwort `aspect` definiert. Im Beispiel wird hier die Advice-Methode `methodeAAdv()` der Klasse `beansAOP.FacadeAdvice` ausgeführt. Ihm wird, dem Pointcut entsprechend, ein Invocation-Objekt vom Typ `MethodInvocation` übergeben. Darüber hinaus veranschaulicht das Beispiel die Verwendung dieser `Invocation`-Objekte. Eine nähere Beschreibung soll für sie aber ausbleiben.

Listing 5.2: Beispiel einer JBoss-AOP Interceptor-Klasse mit Advices

```
1  import java.lang.reflect.Method;
2  import org.jboss.aop.joinpoint.FieldInvocation;
3  import org.jboss.aop.joinpoint.MethodInvocation;
4
5  public class FacadeAdvice{
6    public Object methodeAAdv(FieldInvocation inv) throws Throwable{
7      //Nächsten Programmschritt speichern
8      Object obj = inv.invokeNext();
9
10     doSomething();
11
12     //Nächsten Programmschritt zur Ausführung zurückgeben
13     return obj;
14   }
15
16   public Object methodeAAdv(MethodInvocation inv) throws Throwable{
17     //Aufgerufene Methode speichern
18     Method meth = inv.getActualMethod();
19
20     doSomething(meth);
21
22     //Nächsten Programmschritt zur Ausführung zurückgeben
23     return inv.invokeNext();
24   }
25
26 }
```

Am gezeigten Beispiel ist zu erkennen wie der strukturelle Aufbau eines JBoss-AOP-XML-Deskriptor aussieht. Die Bindungen an Aspektklassen werden innerhalb der dieser Desktiptoren über `bind` Kontrukte definiert. Sie sind formal definiert als:

```
<bind [name] pointcut_def [cflow] >
      { <interceptor_ref name> |
        <stack_ref name> |
        <advice aspect name>  }
</bind>
```

Die Bezeichner `interceptor_ref` und `stack_ref` beziehen sich auf Definitionen ausserhalb des `bind`-Konstruktes. `Aspect` und `name` innerhalb des `advice`-Konstruktes definieren die Aspektklasse sowie die auszuführende Methode bei Erreichen des Pointcut bzw. CFlow-Stack-Zustandes. Für eine tiefergehende Behandlung der XML-Deskriptor-Elemente sei auf entsprechende Lektüre verwiesen. Im Folgenden soll nur die Definition der Pointcuts näher erläutert werden, da diese für das Verstehen der möglichen Joinpoints im JBoss-AOP eine wichtige Rolle spielen.

Formal werden Pointcuts wie folgt definiert [Rup05]:

```
pointcut_def :== action ( kausdruck [logic kausdruck])}
```

```
kausdruck :== [attribut] [return] klassen [-> methode|feld ]
logic :== AND|OR|!
```

methode|feld

Dieser Parameter gibt die Signatur einer Methode bzw. eines Konstruktors oder den Namen eines Feldes an. Bei der Angabe von Methoden und Kontruktoren können zusätzlich Wildcards verwendet werden. Im Kontekt der Pointcuts werden zwei Arten von Wildcards unterschieden:

- * : 0 oder mehr Zeichen. Überall zu verwenden ausser in Annotationen
- .. : Eine beliebige Anzahl von Parametern in Methoden- und Konstruktorsignaturen

klassen

An dieser Stelle werden die Klassen aufgelistet, auf welche der Advice angewandt werden soll.

attribut

Das optionale Attribut bezeichnet die Sichtbarkeit, wie public oder private, einer Methode.

return

Die optionale Angabe des Rückgabetypes kann nur für Methoden mit angegeben werden. Auch sie erlaubt die Benutzung von Wildcards.

action

Wann ein Advice ausgeführt werden soll, wird durch die Definition einer Action angegeben. Bei Eintreten dieser Action werden die angegebenen Advices ausgeführt. Im Folgenden werden die möglichen Action-Typen genannt. Zusätzlich dazu werden für die entsprechenden Invocation-Objekte, welche an die Advice-Methoden übergeben werden, genannt:

- *all*

  Aktionen dieses Typs greifen auf allen Operationen einer Klasse. Sie ist Obermenge aller möglichen Pointcut-Ereignisse. Hierbei können alle möglichen Invocation-Typen an die Advice-Methode übergeben werden. Mittels *overloading* oder entsprechendem Typcasting können Unterscheidungen im Aspektcode vorgenommen werden.

- *call*

  Dieses Ereignis tritt beim Zugriff auf Methoden oder Konstruktoren einer Klasse ein. Anders als bei *execution* wird hierbei der Code des Aufrufers modifiziert und nicht der des Aufgerufenen. Übergeben werden hier Objekte vom Typ

`CallerInvocation`. Sie erlauben einen Zugriff auf das aufrufende sowie das aufgerufene Objekt.

- *execution*

  Dem Typ *call* identisch, tritt dieses Ereigniss ein, wenn eine Methode oder ein Konstruktor ausgeführt wird. Hierbei wird, anders als bei *call*, der aufgerufene Code modifiziert und nicht der aufrufende. An das Advice wird dabei jeweils ein Objekt vom Typ `MethodeInvocation` bzw `ConstructorInvocation` übergeben.

- *field*

  Dieses Ereigniss akkumuliert die Ereignisse *get* und *set* und greift damit bei jedem Zugriff, ob lesend oder schreibend, auf ein Attribut einer Klasse. Das entsprechende Invocationobjekt ist vom Typ `FieldInvocation`.

- *get*

  Jeder lesende Zugriff auf ein Attribut wird durch dieses Ereignis repräsentiert. Entsprechendes Invocationobjekt ist hier vom Typ `FieldReadInvocation`.

- *set*

  Alle Schreibzugriffe auf ein Attribut werden mit dieser Ereignissdefinition beschrieben. Als Invocationobjekt dient dabei ein Objekt vom Typ `FieldWriteInvocation`.

- *has, hasfield, within, withincode*

  Diese Ereignisstypen dienen zur Definition zusätzlicher Bedingungen, die erfüllt sein müssen, damit das Gesamtereignis als erreicht eingestuft wird. Weitere Informationen dazu sind der angegebenen Lektüre zu entnehmen.

Wie beschrieben wurde definieren Pointcuts eine Menge von Joinpoints. Allerdings entspricht jeder dieser Joinpoints nur genau einem Ereignis. Ein Zusammenführen von mehreren Ereignissen zu einem Gesamtereignis ist damit nicht möglich. Zu diesem Zweck gibt es bei vielen aspektorientierten Implementierungen, unter anderem auch in JBoss-AOP, die Möglichkeit der Definition von sogenannten *CFlows*. Sie bilden ein Ereignis welches aus einer definierten Abfolge von Unterereignissen besteht. So lassen sich zum Beispiel feste Abfolgen von verschiedenen Methodenaufrufen zu einem Ereignis verbinden. Bei Ausführung aller definierten Methoden in der definierten Reihenfolge greift das Ereignis und eine Verwebung mit einem Advice kann stattfinden. Definiert werden diese CFlows innerhalb der `bind`-Konstrukte. Auf eine tiefergehende Betrachtung der *cflow*-Technik wird an dieser Stelle aber verzichtet.

Mit Hilfe des XML-Deskriptors und der Pointcut-Definitionen kann der AspectDeployer die Verwebung von Basis- und Aspektcode beim Erreichen des definierten Joinpoint vornehmen. Die dazu nötigen Schritte werden nun in Kapitel 5.2.2 näher erläutert.

5.2.2 Verwebung und Deployment von Aspektklassen

Wie bei allen J2EE-Applikations-Servern werden auch im JBoss-Applikations-Server Komponenten über Pakete, typischerweise als `.jar` oder `.ear` Archive, dem Container zur Verfügung gestellt und durch sogenannte Deployer-Komponenten in die Umgebung integriert. Diesen Prozess nennt man Deployment.

JBoss-AOP wird im Rahmen dieses Deployment-Prozesses als zusätzliche Komponente der Laufzeitumgebung des JBoss-Applikations-Servers hinzugefügt. Dieser beinhaltet einen gesonderten Deployment Manager, den *AspectDeployer*, zum Einlesen der Aspektpakete und dem *AspectManager* zur Verwaltung der eingelesenen Aspektklassen.

Alle Aspektklassen werden zum Zweck des Deployment im Applikation-Server zusammen mit dem AOP-Deployment-Deskriptor `jboss-aop.xml` in Deploymentpakete verpackt. Das Dateiformat entspricht gewöhnlichen J2EE-Deploymentpaketen. Lediglich die Dateiendung wird zum Zwecke der Unterscheidung geändert in `.aop`. Alle Deploymentpakete mit dieser Endung oder einer enthaltenden XML-Datei mit der Endung `-aop.xml` werden vom Standard-Deployer, dem `MainDeployer`, an `AspectDeployer` weitergeleitet und durch diesen weiterverarbeitet.

Der `AspectDeployer` beginnt mit dem Auslesen und Parsen des AOP-Deployment-Descriptor `jboss-aop.xml`. Aus den geparsten Elementen werden in einem Prozess, den man als *Instrumenting* bezeichnet, Klassenrepräsentationen generiert. Im Anschluss daran werden aus diesen instrumentierten Klassen Instanzen gebildet. Innerhalb dieses *Instrumenting* findet im Falle des *Load-Time-Weaving* die Verwebung des Aspekt- und Basiscode statt.

Die Schritte des Deployment und der Instrumentalisierung sind zusammenfassend in Abbildung 5.4 dargestellt.

Durch JBoss-AOP werden drei Webetechniken implementiert, die im Folgenden kurz genannt werden [JBo]:

- *Compile-Time-Weaving*

  JBoss-AOP erlaubt das Verweben der Basisapplikation mit dem Aspektcode vor einem Deployment der Applikation. Dazu wird der *JBoss-AOP Precompiler*, auch *AopC* genannt, verwendet, um den Aspektcode auf Bytecode-Ebene mit dem vorkompilierten Basiscode zu verweben.

- *Load-Time-Weaving*

  Eine Verwebung zur Ladezeit wird, wie auch schon bei anderen AOP-Implementierungen (Siehe 5.1.5), durch eine Änderung der *Classloader* ermöglicht. Dieser ist Teil des Applikations-Servers und verwebt die Basisklassen beim Laden mit dem entsprechenden Aspektcode.

- *HotSwap*

  Als *HotSwap* wird das von JBoss-AOP implementierte *Run-Time-Weaving* bezeichnet. Hierbei werden die Basisklassen ohne eine vorherige Verwebung dem Deploy-

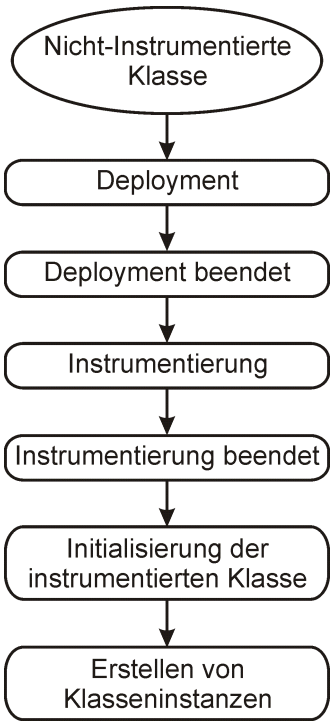

Abbildung 5.4: Deployment und Instrumentalisierung von Klassen innerhalb des JBoss-Applikation-Server (nach [DJS04])

ment zugeführt und instrumentiert. Nachdem ein Joinpoint der geladenen JBoss-AOP-Deskriptoren zur Laufzeit der Basisklassen greift, wird der Kontrollfluss unterbrochen und eine Verwebung vorgenommen. Anschließend wird im Kontrollfluss der Aspektdefinition entsprechend fortgefahren.

JBoss-AOP gestattet dabei die gleichzeitige Verwendung des *Compile-Time-* und *Load-Time-Weaving*. Die zum Zeitpunkt der Kompilierung verwebten Klassen werden dabei von der Transformation zur Ladezeit ausgeschlossen und damit nicht zweifach um den Aspektcode erweitert. In anderen Java-basierenden AOP-Implementierungen kann dies zu zweifacher Ausführung des Aspektcodes führen, da eine Verwebung während der Kompilierung für gewöhnlich selbst wieder Standard-Bytecode erzeugt. Dieser ist anschließend durch den *Classloader* des *Load-Time-Weaving* von anderen, bisher nicht verwebten Bytecode nicht zu unterscheiden und wird so erneut einer Transformation unterzogen. JBoss AOP bietet somit einen Vorteil gegenüber anderen Implementierungen und steigert dadurch den Umfang seiner praktischen Anwendbarkeit.

5.3 Object Teams

Das aspektorientierte Programmiermodell *ObjectTeams* erwuchs aus vorangegangenen Entwicklungen im Bereich der Aspektorientierung und verfolgt das Konzept der Rollen-

basierten Programmierung. Neben der Java-Implementierung *ObjectTeams/Java* existieren des Weiteren Implementierungen für C++, bezeichnet als *ObjectTeams/C++* [Pfe03], und für die objektorientierte Skriptsprache *Ruby* [Vei02].

Von diesen ist *ObjectTeams/Java* die bisher ausgereifteste Lösung und dient als Grundlage der vorliegenden Arbeit. Sie ist eine, auf dem JMangler-Framework (Siehe 5.1.5) aufbauende Lösung, welche die Adaption der Basisklassen zu deren Ladezeit erlaubt. Dies gestattet eine vom Basiscode getrennte Entwicklung des Aspektcode. Die Verwebung auf Bytecode-Ebene ermöglicht zusätzlich auch die Adaptierung von vorkompiliertem Basiscode. Näheres zu der zugrundeliegenden Technik von ObjectTeams/Java kann [Hun03] und [Bin02] entnommen werden.

Dieses Kapitel wird im Folgenden die konzeptionelle Seite des Programmiermodells *ObjectTeams* anhand der Implementierung *ObjectTeams/Java* erläutern. [Obj]

5.3.1 Rollenbasierte Programmierung

Das Paradigma der Objektorientierung beschreibt Objekte als isolierte Entitäten mit einer festen und einheitlichen Menge von Fähigkeiten. Die *rollenbasierte Programmierung* bietet Konzepte zur Lockerung dieser festen Objekteigenschaften zum Zwecke der besseren Anpassung an Objektinteraktionen. So werden Rollen durch Kristensen beschrieben als [Ks96]:

> „A role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects."

Eine Rolle wird einem Objekt zugewiesen und kann deren Verhalten übernehmen. Dadurch lassen sich Rollen als Untermengen des Objektverhaltens verstehen die an die Anforderungen der Rolle zugeschnitten sein können. Rollen bieten damit die Möglichkeit der eingeschränkten Sicht auf ihre Basisobjekte und können darüber hinaus zusätzliche Funktionalitäten einbringen.

Die Charakteristika von Rollen werden in [Ks96] zusammenfassend beschrieben als:

- SICHTBARKEIT: Die Sichtbarkeit eines Objekts wird auf die Methoden und Attribute der dazugehörigen Rolle beschränkt. Der Zugriff auf Methoden und Attribute des Objekts wird demnach nur durch die Ausprägung der Rolle selbst bestimmt.

- ABHÄNGIGKEIT: Die Existenz einer Rolle hängt direkt von dem Objekt ab, zu welcher sie gehört. Ihre Lebensdauer wird demnach durch die Lebensdauer des Objektes bestimmt.

- IDENTITÄT: Rollen bilden mit ihren Objekten eine Einheit und haben eine von ihnen abhängige Identität. Durch diese vereinte Entität können sie nicht gleichzeitig zu verschiedenen Objekten gehören. Es wird zwischen einer Rollen- und Entitätenidentität unterschieden, wobei zwei Rolleninstanzen Entitätenäquivalent sind, wenn sie zum selben Objekt gehören.

- DYNAMIK: Ein Objekt kann während seiner gesamten Lebensdauer Rollen annehmen und ablegen.

- MULTIPLIZITÄT: Einem Objekt können mehrere Instanzen eines Rollentyps gleichzeitig zugewiesen sein.

- ABSTRAKTHEIT: Rollen können zueinander in Generalisierungs- und Aggregationsbeziehungen stehen.

Im Folgenden wird die Bedeutung der rollenbasierten Programmierung für die Implementierung *ObjectTeams/Java* weiter konkretisiert werden.

5.3.2 Kollaborationen

Bei der Entwicklung von Anwendungen mittels objektorientierter Sprachen lassen sich für gewöhnlich zwei Dimensionen der Abstraktion identifizieren. Neben der strukturellen Abstraktionssebene mittels Klassenhierarchien, finden sich zusätzlich Kollaborationen zwischen den Objekten dieser Klassen (Abb. 5.5). Jedes Objekt kann an solch einer Kollaboration beteiligt sein, wobei jede Kollaboration immer mehrere Objekte umfasst, die ihrerseits Teil weiterer Kollaborationen sein können. Ein beteiligtes Objekt spielt innerhalb der Kollaboration eine bestimmte Rolle, welche in der Objektklasse implementiert ist. Die klassenübergreifenden Kollaborationen bilden selbst *Crosscutting Concerns*. Objektorientierte Sprachen ohne eine entsprechende Erweiterung bieten hier keine Möglichkeit, diese Kollaborationen zu modularisieren. [Her02b]

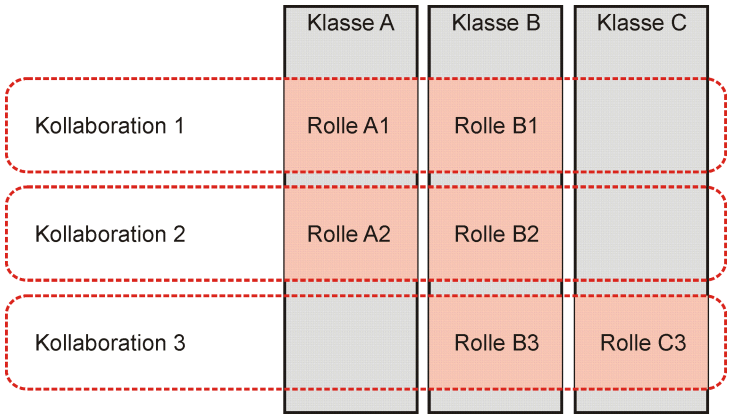

Abbildung 5.5: Dimension der Kollaborationen innerhalb von Klassenhierarchien

Die Kapselung ganzer Kollaborationen und ihrer Rollen in Module ist das Ziel des *ObjectTeams*-Modells. Die Module treten dabei in Form der sogenannten *Teams* und der in ihnen gekapselten *Rollen* in Erscheinung. Sie werden Thema der folgenden Betrachtungen sein.

5.3.3 Teams

Das *ObjectTeams*-Modell dient der Modularisierung von Kollaborationen und aller darin enthaltenen Rollen. Um Kollaborationen selber als Modul zu gestalten, wurde das Konzept der *Teams* eingeführt. Ein Team beinhaltet alle Rollen der durch ihn modularisierten Kollaboration. Er dient damit als eine Art Container für die Rollen einer Kollaboration. Kollaborationen und Rollen können durch Teams getrennt von Basisklassen definiert und non-invasiv, d.h. ohne Änderung der Basisklassen an diese gebunden bzw. später mit ihnen verwebt werden. Neben den Rollen kann jedes Team eigene Attribute und Methoden definieren, welche innerhalb jeder Rolle des Teams zur Verfügung stehen.

Ein Team kann des Weiteren durch Vererbung spezialisiert werden. Diese ist jedoch eingeschränkt. Teamklassen ist es nur erlaubt von anderen Teamklassen zu erben. Wenn eine Teamklasse explizit von keiner anderen Teamklasse erbt, so ist diese immer eine Unterklasse der Teamklasse `org.objectteams.Team`. Demnach sind alle Teamklassen Spezialisierungen dieses Superteams. Vererbungen zwischen einer Teamklasse und einer nicht als Team definierten Klasse sind in beiden Richtungen nicht zulässig. Der Zugriff auf ein solches Superteam wird von einem Team aus mit dem Standard-Java-Schlüsselwort `super` ermöglicht.

Teams werden in Form von Klassen definiert, in welche Rollen als *inner-classes* hinzugefügt werden können. Teamklassen ohne innere Rollenklassen können als Superteams zur Vererbung von Methoden und Atrributen genutzt werden. Sie dienen für gewöhnlich der Definition von Kollaborationen. Die sie spezialisierenden Teamklassen bilden dann den sogenannten *Konnektor*, d.h. die Teamklasse, welche die Bindungen der Klassen und Methoden zwischen den Rollen- und Basisklassen beinhaltet. Als Schlüsselwörter zur Definition von Teams und Rollen dienen `team`, welches eine Teamklasse kennzeichnet, und `playedBy` das zur Bindung einer Rollenklasse an die entsprechende Basisklasse genutzt wird. Listing 5.3 veranschaulicht die Nutzung dieser beiden Elemente.

Listing 5.3: Teamklasse mit Rollendefinitionen

```
1  public abstract team class TeamS {
2    public abstract class RoleA{}
3  }
4
5  public team class TeamA extends TeamS{
6    //Attribut auf Team-Ebene
7    private int attributeA;
8
9    //Methode auf Team-Ebene
10   private void doSomething(){}
11
12   public class RoleA playedBy BasisA{}
13 }
```

Im Beispiel aus Listing 5.3 werden zwei Teams erstellt. Das abstrakte Team `TeamS` dient hierbei als Superklasse der Teamklasse `TeamA`. Dieses erweitert auf Teamebene das Superteam um ein Attribut und eine Methode. Neben der Teamvererbung existiert in

diesem Fall auch noch eine Rollenvererbung in der `TeamA.RoleA` von `TeamS.RoleA` erbt. Dies ist Thema des folgenden Abschnittes, in dem das Beispiel noch weitere Verwendung finden wird.

Teamaktivierung

Alle *callin*-Bindungen der Rollenklassen (Siehe 5.3.4) werden erst wirksam, wenn die umschließende Teamklasse aktiv geschaltet ist. Jede Teamklasse erbt von `org.objectteams.Team` zu diesem Zweck unter anderem die Methoden `activate()` und `deactivate()`. Sie dienen der Aktivierung bzw. Deaktivierung der erbenden Teamklasse. Zusätzlich gibt es die Möglichkeit einer temporären Teamaktivierung. Dazu führt ObjectTeams/Java das Schlüsselwort `within` ein. Syntaktisch wird diese Art der Aktivierung wie folgt notiert:

`within` (*Teamklassen-Instanz*) *Statementblock*

Das folgende Listing 5.4 zeigt die verschiedenen Möglichkeiten der Teamaktivierung auf.

Listing 5.4: Möglichkeiten der Teamaktivierung einer Teamklassen-Instanz

```
1 ...
2 TeamA myTeam = new TeamA();
3 //Aktivieren des Team
4 myTeam.activate();
5 //Deaktivierung des Team
6 myTeam.deactivate();
7 //Aktivierung des Team für die Ausführungsdauer des do-Blocks
8 within (myTeam) do{...}
9 ...
```

Die genannten Möglichkeiten der Teamaktivierung erlauben ein Aktivieren und Deaktivieren von Teamklassen zur Laufzeit und geben damit die Kontrolle über die Rollenbeziehungen der Basisklassen. Mit ihnen lässt sich zur Laufzeit der Basisapplikation bestimmen, wann eine Rolle Verhaltensänderungen an der Basisklasse vornehmen soll und wann nicht.

Eine weitere Möglichkeit der Teamaktivierung wird durch die Verwendung sogenannter *Guards* gegeben. Diese können an callin-Bindungen, Rollenmethoden, Rollenklassen und Teamklassen gebunden werden. Mittels boolsche Ausdrücke entscheiden sie, ob das jeweilige Element, an welches Sie gebunden sind, zum Zeitpunkt des Eintreten des Elementereignisses aktiviert bzw. deaktiviert ist. Durch die Möglichkeit der freien Definition dieser boolschen Ausdrücke, unterstützen *Guards* die dynamische Gestaltung der Aktivierung der jeweiligen Elemente.

An dieser Stelle soll aber eine weiterführende Betrachtung der *Guards* ausbleiben.

5.3.4 Rollen

Wie in Kapitel 5.1 beschrieben, werden Aspektadaptionen durch das Definieren von Aspektcode und Pointcuts gebildet. Im ObjectTeams-Modell werden diese gemeinsam

innerhalb von Rollenklassen definiert. Eine Trennung von Pointcut-Definitionen und Aspektcode, wie zum Beispiel bei JBoss-AOP (Siehe 5.2), findet nicht statt. Rollen können dabei zum einen direkt innerhalb der Teamklassen definiert oder separiert in eigenen Dateien gespeichert werden. In Fall der seperaten Speicherung werden diese dann über den, von ObjectTeams/Java eingeführten, JavaDoc-Tag `@role «Rollenklasse»` in ein Team eingebunden. Semantisch unterscheiden sich diese beiden Arten der Rolleneinbettung jedoch nicht.

Die Bindung von Rollenklassen an die entsprechenden Basisklassen wird über das Schlüsselwort `playedBy` vorgenommen. Aus Sicht der Basisklasse definiert es, welche Rolle diese Basisklasse spielt. Die Rollenklasse selber beinhaltet neben dem Aspektcode, in Form von Methoden und Attributen, auch die Definitionen der Pointcuts durch sogenannte *callin*- und *callout*-Bindungen. Sie werden im Folgenden erläutert und im Listing 5.5 beispielhaft demonstriert.

- *callin-Bindungen*

  Aus Sicht der Rolle bezeichnet ein *callin* eine eingehende Kommunikationsrichtung. Der Kontrollfluss des Basisobjekts wird hierbei unterbrochen, um Rollenverhalten entsprechend den Pointcuts einzufügen. Es ist das Äquivalent zur Verwebung von Aspekt- und Basiscode in anderen aspektorientierten Sprachen. Eine callin-Bindung ermöglicht eine Pointcut-Definition mit folgenden Ausführungstypen:

  - `before` : Ausführung des Rollenverhaltens *vor* Aufruf der Basismethode
  - `after` : Ausführung des Rollenverhaltens *nach* Aufruf der Basismethode
  - `replace` : Ausführung des Rollenbverhaltens *anstelle* Basismethode

  Die callin-Bindungen *before* und *after* werden im Sinne einer Erweiterung der Basismethode verwendet. Rollenverhalten wird bei diesen vor bzw. nach dem Aufruf der Basismethode hinzugefügt. Die *replace*-Bindung ersetzt hingegen die gesamte Basismethode durch die entsprechende Rollenmethode. Zur Kennzeichnung ihrer expliziten Nutzung werden Rollenmethoden einer `replace-callin`-Bindung zusätzlich mit dem Schlüsselwort `callin` ausgestattet. Sie gestatten nur eine ausschliessliche Nutzung durch `replace`-Bindungen und unterbinden darüber hinaus die Definition der Methodensichtbarkeit.

  Jede `Callin`-Bindung ist syntaktisch wie folgt aufgebaut:

  Rollenmethode **<- {before, after, replace}** *Basismethode*

  Innerhalb aller Rollenmethoden steht des Weiteren das Schlüsselwort `base` zur Verfügung. Dieses erlaubt den Zugriff auf das Basisobjekt der Rolle. `Callin`-Rollenmethoden können jedoch jeweils von mehreren `callin`-Bindungen genutzt werden. Daraus resultiert, dass die Basismethode, dessen Aufruf zum Ausführen der `Callin`-Rollenmethode führte, variieren kann. Aus diesem Grund gibt es zusätzlich einen sogenannten `base-call`. Dieser wird innerhalb einer `callin`-Rollenmethode zur Ausführung der Basismethode benutzt, welche zur Aktivierung

des `callin`-Pointcuts geführt hat. Innerhalb einer `callin`-Rollenmethode mit der Signatur, d.h. Methodenname inklusive angestellter Parameterliste, <*Signatur*>, wird dieser getätigt mit dem Aufruf `base.`*`<signatur>`*. Ein Beispiel für solch einen `base-call` ist in Zeile 15 des Listing 5.5 zu finden.

- *callout-Bindungen*

  Ein *callout* führt entlang der entgegensetzten *callin*-Kommunikationsrichtung. Er definiert eine Weiterleitung von Methodenaufrufe innerhalb der Rollenmethode an Basismethoden der assoziierten Basisklasse. Zu diesem Zweck wird für jede `callout`-Bindung eine entsprechende abstrakte Methode definiert, die Repräsentant der Zielmethode der Basisklasse ist. Diese abstrakten Methoden werden innerhalb der Definition der `callout`-Bindung verwendet, dessen Syntax wie folgt:

  `Rollenmethode` `->` `Basismethode` [**with** { *`withExpression`*}]

  `withExpression` := { *expression* -> Basismethodenparameter, result <- *expression* } [, withExpression] *`withExpression`*

  Das optionale `with`-Konstrukt dient dem Parametermapping von Rollen- und Basismethoden. Innerhalb dieses Blocks können über die Pfeilrichtung „`->`“ Parameter der Rollenmethode an Parameter der Basismethode gebunden werden. Die umgekehrte Richtung „`result <-`“ gestattet die Definition eines Rückgabewertes der Rollenmethode anhand eines Expressionausdrucks. Dieser Ausdruck kann selbst das `result`-Schlüsselwort beinhalten, wobei dieser sich dann jedoch auf das Rückabeergebnis der Basismethode bezieht. Als *expressions* werden in diesem Kontext Java-spezifische Ausdrücke bezeichnet.

  Im Listing 5.5 ist in den Zeilen 25-27 ein Beispiele für beide Arten des Parameter-Mapping innerhalb einer callout-Bindung zu finden. Zeile 26 sorgt in diesem Falle für ein Mapping des Parameters `integer` der Rollenmethode `doCalc` auf den Parameter `i` der Basismethode `doBaseCalc`. In Zeile 27 wird ein `result`-Mapping definiert, welches aus dem Rückgabewert der Basismethode ein `java.lang.Integer`-Objekt erzeugt und dieses als Rückgabewert der Rollenmethode nutzt.

  Zusätzlich erlaubt ObjectTeams/Java den Zugriff auf Attribute der Basisklasse mittels einer `callout`-Bindung. Diese Technik gestattet es Attribute jedweder Sichtbarkeit zu lesen, als auch zu ändern. Sie macht den Entwickler damit unabhängig von bereits vorhandenen Getter- bzw. Settermethoden und gibt ihm somit mehr Freiheiten. Beispielhaft zeigt Zeile 30 und 31 die Verwendung eines Getter- und Setter-callout. Der jeweilige Zugriff auf Seiten der Rollenmethode geschieht dabei in Form zweier Methoden, welche selbst die Struktur einer Standard Getter- bzw. Setter-Methode besitzen.

In den beiden genannten Bindungsdefinitionen hat die linke Seite einer „<-“ bzw. „->“ Pfeilrelation stets einen rollenbezogenen und die rechte Seite stets einen basisbezogenen Bezug. Bindungsdefinitionen werden damit lokationssensitiv und sind semantisch empfindlich gegen Spiegelung der Definitionsdeskriptoren.

Listing 5.5: Beispiel einer Teamklasse mit Vererbung und `callin`- bzw. `callout`-Bindungen

```
1  public abstract team class TeamS {
2    public abstract class RoleA{
3      //optionale, abstrakte Repräsentation der Callout-Rollenmethode
4      abstract Integer doCalc(Integer i);
5    }
6  }
7  public team class TeamA extends TeamS{
8    public class RoleA playedBy BasisA{
9
10     private void doAfter(){...}
11
12     callin void doReplace(){
13       ...
14       //Aufruf der ursprünglichen Basismethode mittels eines base-
             call
15       base.doReplace();
16     }
17
18     //After callin-Bindung
19     void doAfter() <- after void baseMethodA();
20
21     //Replace callin-Bindung
22     void doReplace() <- replace void baseMethodB();
23
24     //callout-Bindung aus eine Basismethode
25     Integer doCalc(Integer integer) -> int doBaseCalc(int i) with {
26         integer.intValue() -> i,
27         result <- new Integer(result) }
28
29     //callout-Bindung auf ein Basisattribut
30     int getAttr() -> get int attr;
31     void setAttr(int newVal) -> set int attr;
32   }
33 }
```

Neben den Teamklassen können auch Rollenklassen Teil von Vererbungshierarchien sein. Dabei wird zwischen einer expliziten und impliziten Vererbung unterschieden. Die *explizite Vererbung* wird deklarativ über das Schlüsselwort `extends` vorgenommen und ist auf keinen Klassentyp begrenzt. Sie gleicht der Vererbungstechnik in Standard-Java-Applikationen. Dabei dient das Java-Schlüsselwort `super` dem Zugriff auf die Superklasse.

Eine *implizite Vererbung* erfolgt dagegen ohne Angabe von weiteren Schlüsselwörtern. Sie ergibt sich aus der Vererbungshierarchie der sie umgebenen Teamklassen. Rollenklassen innerhalb eines Teams erben dabei automatisch von namensgleichen Rollenklassen des Superteams. Sie überschreiben dann die Rollen des Superteams und übernehmen bzw. erweitern selbst deren Eigenschaften.

Beide Arten der Vererbung erlauben das Überschreiben von Rollenmethoden, nicht jedoch das Ersetzen von callin- bzw. callout-Bindungen. Um auf die Superklasse der impliziten Vererbung zuzugreifen, wird durch ObjectTeams/Java das neue Schlüsselwort `tsuper` eingeführt. Es referenziert auf die namensgleiche Rollenklasse des Superteams, von welcher die Rollenklasse erbt.

Im Listing 5.5 erbt in Zeile 7 die Teamklasse `TeamA` von der abstrakten Teamklasse `TeamS`. Damit erbt diese auch alle ihre Rollenklassen, in diesem Falle die abstrakte Rolle `RoleA` aus Zeile 2. Sie besitzt lediglich eine abstrakte Methode `doCalc(Integer)`, welche an die erbende `RolleA` aus Zeile 8 vererbt wird. Diese Methode ist äquivalent zur resultierenden Rollenmethode der Callout-Bindung aus Zeile 25. Da Callout-Bindungen immer, wenn noch keine solche vorhanden ist, eine abstrakte Version ihrer gebundenen Rollenmethode erstellen, ist die abstrakte Methode `doCalc` aus Zeile 4 optional, d.h. ohne Auswirkungen auf Klassen oder Bindungen. Sie ersetzt lediglich die automatisch generierte Methode der Callout-Bindung und dient so lediglich dem besseren Verständniss.

5.3.5 Modellierung aspektorientierter Anwendungen

Die Modellierung von aspektorientierten Anwendungen erfordert aufgrund der neuen Elemente eine Anpassung der vorhandenen Modellierungssprachen. Mit diesem Ziel wurde für ObjectTeams eine Erweiterung der *Unified Modeling Language* (UML) mit dem Namen *UFA* (UML for Aspects) definiert [Her02a]. Des Weiteren wurde, im Rahmen einer Diplomarbeit, eine Implementierung dieser UFA-Spezifikationen über ein grafisches Modellierungstool für die Entwicklungsumgebung *PIROL* entwickelt. [Hac02]

Einen Einblick in die Modellierung von ObjectTeams-Anwendungen mit Hilfe von *UFA* gibt Abbildung 5.6. Es ist unter anderem zu erkennen, dass Teams in UFA durch Pakete dargestellt werden. Diese können unterschieden werden in abstrakte Teams, welche für gewöhnlich Kollaborationen definieren, und Konnektoren. Konnektoren beinhalten typischerweise die Rollenklassen, die an Basisobjekte gebunden werden mitsamt der Bindungsdefinitionen. Semantisch werden diese auf Modellebene durch die Typbezeichner `{abstract}` und `«connector»` an Stelle des Paketstereotyps in der UML dargestellt. Um die Vererbung zwischen Teams zu modellieren, wurde in UFA die Vererbung zwischen Paketen ermöglicht. Des Weiteren wird die Bindung eines Teamkonnektors an eine Basisklasse mittels des UML-Aggregationssymbols symbolisiert. Zur besseren Unterscheidung wurde dieser in UFA mit dem Stereotyp `adapt` versehen.

Rollen werden in UFA in Form von Klassen definiert, welche um ein Eigenschaftsfeld erweitert wurden. Dieses neue *compartment* dient der Darstellung von callin- und callout-Bindungsdefinitionen. So besitzen Rollenklassen insgesamt drei compartments, die zur Darstellung der Methoden, Attribute und Bindungen dienen. Die `playedBy`-Beziehung einer Rollenklasse wird über eine Aggregation des Klassennamen der Rollen- und der dazugehörigen Basisklasse modelliert. Zu diesem Zweck erhalten diese Rollenklassen als Klassennamen *`Rollenklasse = Basisklasse`*, wobei dann = eine `playedBy`-Beziehung kennzeichnet.

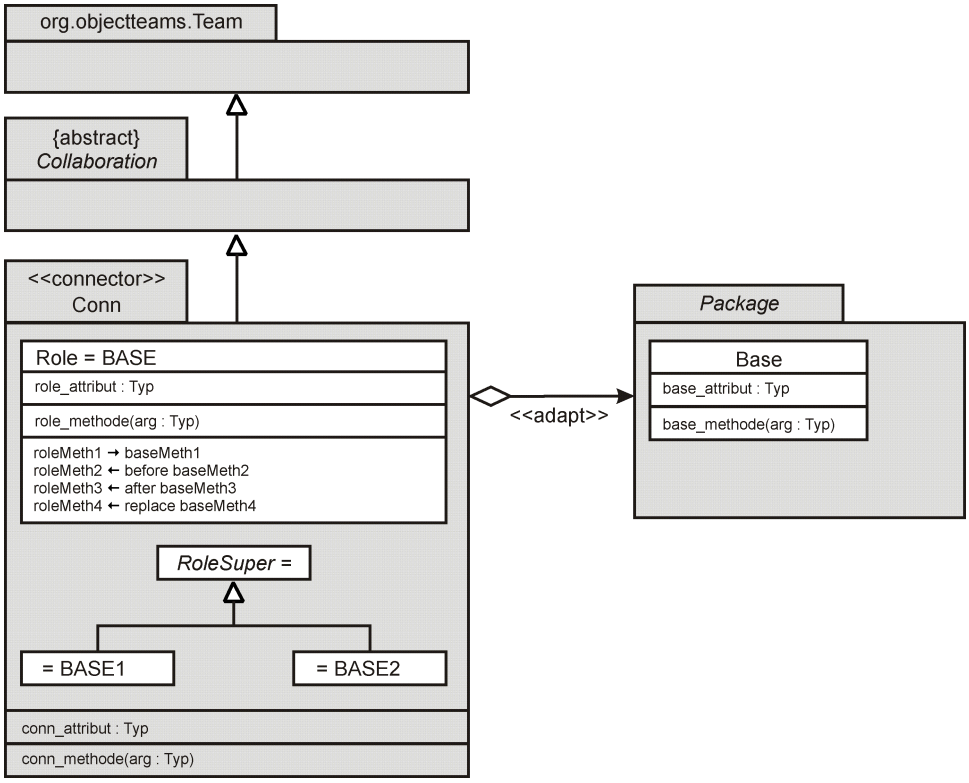

Abbildung 5.6: Modellierung von ObjectTeams-Anwendungen mit UFA (nach [Her02a])

Ein Beispiel einer aspektorientierten Modellierung mit UFA gibt Abbildung 5.7. Es ist ein UFA-Modell einer aspektorientierten Anwendung auf Grundlage des Listing 5.5.

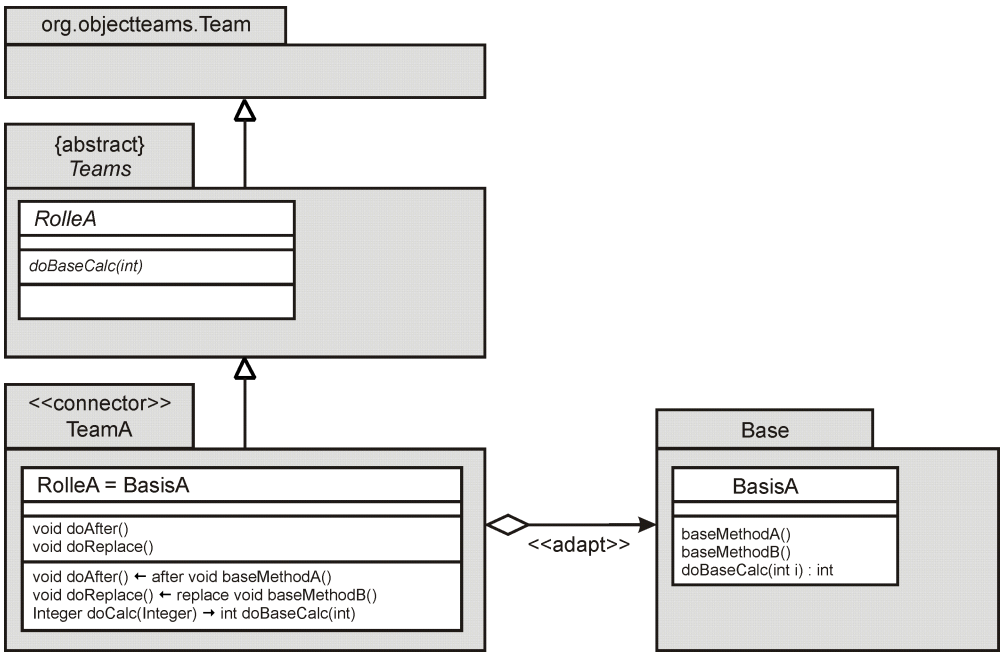

Abbildung 5.7: Beispiel einer Adaption von Basisklassen mit ObjectTeams

5.3.6 Translations-Polymorphismus

Basisklassen können mit einer Vielzahl von Team- und Rollenklassen in Relation treten. Im umgekehrten Falle besitzt jedoch jede Rollenklasse eine `playedBy`-Relation zu genau einer Basisklasse. Diese Relationen benötigt man für den Zugriff auf das Rollen- bzw. Basisobjekt innerhalb eines `callin` bzw. `callout`. Da die dafür benötigte Substitution nicht auf Vererbungstechniken basiert, wird eine Technik zur impliziten Überführung von Rollen- und Basisobjekt benötigt. Sie stellt eine Typkonformität zwischen Rollen- und Basisklasse her und erlaubt damit die Verwendung von Polymorphie. Diese sogenannte *Translations-Polymorphie* verwendet, je nach Überführungsrichtung, Techniken des `lifting` bzw. `lowering`. Sie sind die einzigen Mechanismen, die Zugriff auf die Rollen-Basis-Relationen haben. [Her02b]

lowering

Jede Instanz einer Rollenklasse hält intern eine Referenz auf ihr Basisobjekt vor. Diese ist während der gesamten Laufzeit der Rolle vorhanden und kann nicht verändert werden. Der Zugriff auf dieses Basisobjekt wird dabei *lowering* genannt. Die lowering-Translation wird automatisch vom Compiler vorgenommen, wenn ein Rollentyp angegeben wurde, aber ein entsprechender Basistyp erwartet wurde.

Listing 5.6: Compiler-Typisierung durch lowering-Translation in einer Rollenklasse

```
1 public class RoleA playedBy BasisA { ... }
2 public void doSomething() {
3   RoleA r = new RoleA();
4   BasisA b = r;
5 }
```

Listing 5.6 zeigt ein Beispiel einer automatischen Typisierung mittels einer *lowering*-Translation durch den Compiler. In Zeile 3 wird hier eine Rolleninstanz gebildet, die im Anschluss an ein Basisobjekt gebunden wird. Da die Rollenklasse nicht typkonform zur Basisklasse ist, führt der Compiler automatisch eine lowering-Translation auf das Basisobjekt der Rolle aus und stellt damit die Typkorrektheit des Ausdrucks in Zeile 4 wieder her.

Daneben erlaubt eine explizite lowering-Translation den manuellen Zugriff auf das Basisobjekt einer Rolle. Sie wird über die Rollemmethode `lower()` durchgeführt und würde im vorangegangenem Listing in Zeile 4 der Form `BasisA b = r.lower();` sein.

lifting

Die umgekehrte Translation zum *lowering* ist das *lifting*. Es wird für den Zugriff auf die gebundenen Rollen eines Basisobjekts benötigt. Da jedoch ein Basisobjekt beliebig viele Rollen *spielen* kann, bedarf diese Art der Translation mehr an Kontextinformationen als das *lowering*. Der Compiler verwendet sie unter anderem beim `result`-Mapping einer `callout`-Bindung und innerhalb von Methodenaufrufen einer `callin`-Bindung.

6 Compiere Monitor

Die vorangegangenen Kapitel führten thematisch in die Bereiche der aspektorientierten Programmierung und der ERP-Systeme ein. Im Kontext dieser Arbeit lag der Schwerpunkt hierbei besonders auf ObjectTeams (Siehe 5.3) und dem ERP-System *Compiere* aus Kapitel 4. Sie dienen als Grundlage zur Erfüllung der Zielsetzung dieser Arbeit - der Evaluierung der Techniken der aspektorientierten Programmierung zur Stärkung der Wandlungsfähigkeit eines ERP-System. Zu diesem Zweck wurde ein Programm zur Integration von ObjectTeams-Elementen in Compiere entwickelt, der *Compiere Monitor*. Im Folgenden soll dieser einer ausführlichen Betrachtung unterzogen werden.

6.1 Aspektorientierung in Compiere

In Kapitel 5 wurde deutlich, dass die Adaption bzw. Erweiterung eines Systems mittels des Paradigmas der Aspektorientierung einigen Einschränkungen und Bedingungen unterliegt. Speziell die Möglichkeiten der Verwebung (Siehe 5.1.3), wirken in diesem Zusammenhang als wichtige Kriterien bei der Auswahl möglicher aspektorientierter Sprachen. Ziel dieses Abschnittes ist es, im Kontext von Compiere einen kurzen Überlick der verschiedenen Ansätze mit ihren Vor- und Nachteilen zu geben.

Als kommerzielles Open-Source-Produkt ist der Quellcode, unter Berücksichtigung der jeweiligen Lizenzbestimmungen, potenziell offen für jedwede Veränderung. Dies bedeutet, dass das Produkt veränderbar und anschließend selbst kompilierbar ist. Dieses Faktum ist grundlegend, da es ein wichtiges Kriterium bei der Eingrenzung möglicher aspektorientierter Techniken und damit auch zu verwendender Aspektsprachen ist. Es erlaubt dadurch die Nutzung von Sprachen, welche *source-code-weaving zur compile-Zeit* erlauben. Da *load-time-* und *run-time-weaving* Binär- bzw. Bytecode verwenden, spielt für sie die Offenheit des Compiere-Quellcodes hingegen keine Rolle. Lediglich bei der Entwicklung des Aspektcodes wirkt diese Eigenschaft wieder positiv ein. Offener Quellcode hilft dem Entwickler das Systemverhalten besser zu verstehen, um so den Basiscode effizienter adaptieren zu können. Compiere bietet daher sehr gute Voraussetzungen für eine Erweiterung mittels aspektorientierter Sprachen.

Wie in 4.2.2 beschrieben wurde, besitzt Compiere zwei Applikationsebenen. Zum einen gibt es die Compiere-Hauptanwendung. Sie ist als Standard-Java-Applikation implementiert und erlaubt, in Verbindung mit der genannten Quelloffenheit, potenziell die Nutzung aller Java-basierenden Aspektsprachen. Die zweite Ebene bildet der JBoss-Applikationsserver. Die Nutzung eines eigenen Classloaders unterbindet hier die Verwendung der meisten Aspektsprachen. Lediglich Aspektimplementierungen mit einer

Verwebung zur *compile-Zeit* sind hier eine Ausnahme, da diese nach der Kompilierung wieder Standard-Java-Bytecode darstellen und so vom Classloader des Applikationsserver problemlos genutzt werden können. Aspektsprachen, die *load-* und *run-time-weaving* oder gar ein dynamisches Weben verwenden, können hingegen in einem J2EE-Container nicht ohne Weiteres verwendet werden. Um diesem Abhilfe zu schaffen, wurde durch die JBoss Inc. die Aspektsprache JBoss AOP (Siehe 5.2) entwickelt. Sie erlaubt die Verwendung dynamischer Webetechniken innerhalb des JBoss-Applikationsservers und erweitert damit die Möglichkeiten zur Nutzung aspektorientierter Programmierung in Compiere. Aufgrund der eingeschränkten Verwendung des JBoss-Applikationsservers durch Compiere (Siehe 4.2.2), liegt das Augenmerk dieser Arbeit jedoch auf der Erweiterung der Compiere-Hauptanwendung.

Damit der Aspektcode der Compiere-Hauptanwendung flexibel ausgetauscht werden kann ohne den Quellcode von Compiere neu kompilieren zu müssen, fiel die Auswahl auf ObjectTeams/Java (Siehe 5.3). Es unterstützt das *load-time-weaving* für Java-Applikationen und bietet damit genügend Potential, um die angestrebten Funktionalitäten und nötigen Aspekte zum Erreichen der Ziele erfolgreich zu implementieren.

Darüber hinaus wirkt sich die einfache Integrationsfähigkeit von *ObjectTeams/Java* in Compiere zusätzlich positiv aus. Es bedarf dazu lediglich einer Anpassung der ausführenden Kommandozeile im Startscript von Compiere. Dabei werden, über die Verwendung des *Bootstrap-Loaders* von *JMangler* (Siehe 5.1.5) alle nötigen Komponenten von *ObjectTeams/Java* geladen, noch bevor Compiere selbst seine Ausführung beginnt. Die dazugehörige *Bootstrap*-Klasse ersetzt hierbei die ursprüngliche Startklasse von Compiere und fügt vor Ausführung dessen Startmethode den nötigen Code zur Einrichtung der *JMangler*-Komponenten aus. Die genauen Details sind für den weiteren Verlauf dieser Arbeit jedoch irrelevant und bleiben daher unerwähnt.

6.2 Zielsetzung

Kapitel 2.5 behandelte thematisch die Vor- und Nachteile von Open-Source-Lösungen für Unternehmen. Der dort genannte Punkt der *Flexbibility* spielt besonders bei kommerziellen Open-Source-Systemen eine bedeutende Rolle. Insbesondere die Änderung des Originalcodes führt hier zu Nachteilen. Es ist ungewiss ob, und wann eine individuelle Änderung am Code Teil eines offiziellen Releases wird. Individualänderungen am System nehmen einem darüber hinaus die Fähigkeit, Produktevolutionen zu folgen. Man grenzt sich damit von den Vorteilen der weltweiten Entwicklergemeinschaft aus, da man Releases verwendet, die zum Zeitpunkt der ersten Individualisierung aktuell waren, es später aber nicht mehr sein müssen, und zudem nicht den offiziellen Releases entsprechen. Durch diese fehlende Einheitlichkeit des verwendeten Release wird die Möglichkeiten des Support über öffentliche und kostenfreie Wege damit stark eingeschränkt.

Ziel dieser Arbeit ist es, die positiven Einflüsse der Aspektorientierung auf die Wandlungsfähigkeit aufzuzeigen. Um die genannten Nachteile von Individualisierungen dabei

zu verhindern, besteht ein Sekundärziel darin, den Basiscode des verwendeten Compiere-Release unverändert zu lassen. Zu diesem Zweck wurde der *Compiere Monitor* entwickelt. Er dient der Erstellung und Verwaltung von Aspekten für Compiere. Dessen Ziele und Problemstellungen werden nun kurz genannt:

- ***Zentralisierung des Aspektcodes***

  Compiere ist ein verteiltes System, d.h. viele Klient-Anwendungen greifen auf einen Server, in diesem Fall den JBoss-Applikations-Server bzw. das Datenbankmanagementsystem, zu. Dies erschwert die Verteilung von Aspektcode auf die unterschiedlichen Klienten. Aus diesem Grund kommt der zentralisierten Aspektverwaltung eine wichtige Bedeutung zu. Sie erlaubt das Ändern von Aspektcode ohne jeden der Klient-Systeme anpassen zu müssen.

- ***Unterstützung der Compiere-Aspektgenerierung***

  Die Wandlungsfähigkeit von ERP-Systemen wird bestimmt durch eine Vielzahl von Kriterien und Indikatoren (Siehe 3.2). Hier passende Elemente auszuwählen und diese mit geeigneten Aspekten zu stärken, war das Ziel dieser Arbeit. Diese Aspekte bilden so die Grundlage zum Erreichen dieses Zieles und müssen durch den *Compiere Monitor* automatisch generiert werden können. Um dabei die Anpassbarkeit der einzelnen Aspekte zu erhöhen, wurde als Sekundärziel definiert, diese Compiere-bezogenen Aspektklassen mittels grafischer Masken anpassen und neu erstellen lassen zu können. Dies bedeutet, dass man z.B. steuern kann, welche Compiere-Nutzer-Einstellungen überwacht werden sollen ohne den entsprechenden Aspektcode manuell ändern zu müssen.

- ***Unabhängigkeit gegenüber der ObjectTeams/Java-Entwicklungsumgebung***

  Die Entwicklung von Aspektcode für ObjectTeams/Java wird unterstützt durch eine Erweiterung der Eclipse[1]-Entwicklungsumgebung, dem *Object Teams Development Tooling (OTDT)*[2]. Sie bietet auf ObjectTeams-bezogene Eclipse-Features, wie das Refactoring oder Debuggen. Dies erleichtert die Entwicklung von Aspekten, erfordert jedoch auch die Verwendung des genannten OTDT. Um die Aspekte von diesem unabhängig verändern zu können, bietet auch der Compiere Monitor Komponenten zur einfachen Umgestaltung der Aspekte. Der Compiere Monitor arbeitet dabei mit Jar-Archiven und bietet somit die Möglichkeit der Anwendung für potenziell jede Applikation. Lediglich der Compiere-spezifische Teil ist an die entsprechenden Jar-Archive von Compiere gebunden. Er beinhaltet die *Aspektgeneratoren* sowie deren *Konfiguratoren*. Sie werden im Folgenden noch weitere Erwähnung finden.

Die genannten Elemente wurden als Grundzielsetzung bei der Implementierung des *Compiere Monitor* definiert. Nähere Erläuterung zu diesen und anderen Funktionalitäten finden sich im nun folgenden Abschnitt.

[1] http://www.eclipse.org
[2] http://www.objectteams.org/distrib/otdt.htm

6.3 Funktionalitäten

Währen der Entwicklung des *Compiere Monitor* kristallisierten sich einige Anforderungen heraus, die über die anfängliche Grundzielsetzung aus Abschnitt 6.2 hinausreichen. Sie erleichtern die Benutzung des Programms sowie der Aspekte und sollen, zusammen mit den Funktionalitäten erster Priorität, hier nun vorgestellt werden.

- ***Zentrale Verwaltung der Aspektklassen***

  Wie im vorherigen Abschnitt schon genannt, wurde diese Funktionalität als eine der Grundzielsetzungen definiert. Sie bedeutete für die Entwicklung des *Compiere Monitor*, dass die Persistenz der verwendeten Daten über einen zentralen Server sichergestellt wird. Da dies typischerweise in Form einer Datenbank erreicht wird, fiel die Entscheidung auf die Verwendung der Datenbank, welche schon durch Compiere genutzt wird. Es wird dabei davon ausgegangen, dass Compiere auf einer Oracle-Datenbank aufgesetzt wurde. Andere Datenbanken werden zwar theoretisch von Compiere unterstützt (Siehe 4.2.1), der *Compiere Monitor* ist derzeit aber nur für Oracle-Datenbanken ausgelegt. Die Möglichkeit der Erweiterung besteht jedoch auch bei ihm durch die Implementierung weiterer Dialektklassen. Diese enthalten alle nötigen Befehle und Statements unter Berücksichtigung des datenbankspezifischen SQL-Dialekts. Aus Zeitgründen wurde hier nur eine Implementierung für Oracle-Datenbanken vorgenommen. Details zur Implementierung dieser Technik finden sich im Abschnitt 6.7.

  Die Verwendung der Compiere-Datenbank gestattet die Benutzung der Compiere-Schnittstellen und führt damit zu einer nahtlosen Integration in dessen Umgebung. So müssen z.B. für den *Compiere Monitor* keinerlei Daten bezüglich der Datenbank eingegeben werden, wenn Compiere bereits auf dem Zielsystem eingerichtet wurde. Der *Compiere Monitor* verbindet sich bei dessen Start automatisch mit der definierten Compiere-Datenbank und erstellt bei Bedarf die nötigen Tabellen, ohne vorher weitere Schritte unternommen zu haben. Eine weitere Einrichtung ist somit nur noch bei den Compiere-Klienten nötig, welche eine Adaption durch die Aspekte erfahren sollen. Die Anpassung bezieht sich bei ihnen auf die Startscripte und verwendeten Java-Bibliotheken. Die Klienten beziehen die Daten ebenfalls automatisch von ihrer lokal eingestellten Compiere-Datenbank und bleiben so immer synchron mit den zentral gespeicherten Aspektelementen. Die hierbei verwendeten Techniken finden sich in Abschnitt 6.5 wieder.

- ***Aspektgeneratoren***

  Die Implementierung zur Unterstützung der Compiere-Aspektgenerierung wurde mittels sogenannter *Aspektgeneratoren* vorgenommen. Die einzelnen Aspektgeneratoren bilden Klassen, welche in hardcodierter Form Informationen über die zu erstellenden Teamklassen besitzen. Mit ihnen werden bei Generierung die entsprechenden Datenbankobjekte erstellt. Die Generatoren dienen so der automatischen Erzeugung der Aspektklassen, welche zur Steigerung der Wandlungsfähigkeit aus-

gewählt wurden. Der Benutzer kann mit ihnen, ohne selber Aspektcode erstellen zu müssen, diese grundlegenden Aspekte erzeugen und weiter verwenden.

Einige der Aspektgeneratoren werden darüber hinaus durch sogenannte *Aspektkonfiguratoren* (Siehe 6.4.2) beeinflusst. Dies sind grafische Masken zur Eingabe verschiedenster Aspekteinstellungen. So können dort z.B. Compiere-Aktivitäten zur Überwachung markiert werden, die dann bei der nächsten Aspektgenerierung Einfluss auf die generierten Team- und Rollenklassen finden. Die automatisch generierten Aspektklassen lassen sich des Weiteren über einen Generatorschlüssel identifizieren und damit gezielt löschen bzw. regenerieren. Er erlaubt eine getrennte Behandlung der individuell erstellten und generierten Aspekte. Einen näheren Überlick über die Aspektgeneratoren gibt Abschnitt 6.4.

- ***Aspekterzeugung und -kompilierung***

  Die durch Aspektgeneratoren erzeugten bzw. individuell erstellten Aspektklassen werden über eine Vielzahl von Tabellen innerhalb der Datenbank abgebildet. Diese Aufteilung dient der Modularisierung der Aspektelemente, wie Team- oder Rollenmethoden, und der damit verbundenen Vereinfachung von Änderungen dieser Aspektelemente. Sie bedingt aber auch eine Komponente, welche diese einzelnen Aspektmodule wieder in eine vollständigen Aspektklasse rekombiniert - den *Aspect-Builder*. Er erzeugt aus den einzelnen Modulen wieder vollständige Teamklassen und ermöglicht damit erst deren Kompilierung auf den Compiere-Klienten.

  Die Kompilierung ist Teil der veränderten Startprozedur der Compiere-Hauptanwendung und wird über den *Team-Creator* vorgenommen. Dieser verbindet sich vor dem Start der Hauptanwendung mit den relevanten Tabellen des Compiere Monitor und erzeugt mit Hilfe des *Aspect-Builder* kompilierbaren Aspektcode in Form von `<teamname>.java`-Dateien. Sie werden anschließend kompiliert und alle gemeinsam in einem `Jar`-Archiv zusammengefasst. Die `Jar`-Datei dient anschließend als Quelle der zu benutzenden Aspektklassen beim Start der, mit ObjectTeams/Java angereicherten Compiere-Hauptanwendung.

- ***Aspektvalidierung***

  Der Compiere Monitor dient als Verwaltungswerkzeug für die Aspektmodule der Datenbank. Ihre eigentliche Benutzung findet durch den *Aspect-Builder* statt. Er generiert aus ihnen Aspektklassen, welche beim Start der Compiere-Klientanwendungen durch den *Team-Creator* kompiliert werden. Erst während dieser Kompilierungsphase der generierten Aspektklassen lassen sich Fehler entdecken. Der *Team-Creator* gibt in diesem Fall Fehlerausgaben über die Systemconsole, sowie über ein grafisches Fenster aus.

  Die Benutzung fehlerhafter Aspektklassen kann zu einem unerwarteten Verhalten der Compiere-Klientanwendung führen. Um dies zu vermeiden ist eine Fehlerprüfung nötig, noch bevor die einzelnen Aspektmodule in der Datenbank gespeichert werden. Zu diesem Zweck wurde die Techniken des *Team-Creator* in den *Compiere Monitor* integriert. Sie erlaubt eine Überprüfung der resultierenden Aspektklassen

von Aspektmodulen, welche nur lokal im Speicher vorhanden sind und ermöglicht so eine Erkennung von Fehlern noch bevor diese Aspektmodule für die Compiere-Klientanwendungen verfügbar sind.

- ***Logische Constraints***

  Das Konzept der *Constraints* findet sich unter anderem in der logischen Constraint-Programmierung[Frü97] sowie bei Datenbankmanagementsystemen. Constraints sind dabei Einschränkungen, welche definiert werden, um z.B. den Erhalt von Invarianten zu fördern oder in der logischen Programmierung eine Funktion direkter modellieren zu können.

  Dieses Konzept wurde im Compiere Monitor aufgegriffen und in Form von sogenannten *Rules* implementiert. Sie überprüfen ganze Tabellen, einzelne Zeilen oder auch nur bestimmte Zellen auf deren Zustand bezüglich eines Constraints hin und erlauben damit die Definition beliebiger Regeln. So wurden z.B. Regeln für die Tabelle der Teamklassen, `TableAspectTeamClass`, definiert, welche fordern, dass der Teamklassenname eindeutig und ein gültiger Java-Identifier sein muss. Die Einschränkungen unterstützen so den Anwender bei der korrekten Eingabe der Daten und helfen bei der Modellierung kompilierbarer Aspektklassen. Funktionell ähneln Rules daher der Aspektvalidierung - auch sie können die Generierung kompilierbaren Aspektcodes durch das Erkennen von fehlerhaften Eingaben unterstützen. Rules arbeiten jedoch auf tieferer Ebene und gestatten so eine Prüfung auf spätere Validität direkt zum Zeitpunkt der Eingabe. Daneben sind Rules auch nicht auf logische Constraints bezüglich der Aspektvalidierung begrenzt. So dienen sie unter anderem auch der Prüfung auf bereits vorhandene Tabellen bei Erstellung dynamischer Logging-Tabellen, die im Anschluss noch erläutert werden. Sie besitzen damit, im Vergleich zur Aspektvalidierung, einen erweiterten Funktionsrahmen.

  Daten, die eine oder mehrere der zugewiesenen Rules verletzen, werden dem Benutzer grafisch erkennbar gemacht und erlauben keine Datenbankspeicherung ihrer selbst. Die Evaluierung eines Rules kann unter Umständen mehrere Tabellen einbeziehen und dementsprechend viel Verarbeitungszeit benötigen. Aus diesem Grund ist diese Technik als optionale Funktion implementiert, d.h. sie kann im *Compiere Monitor* zur Laufzeit deaktiviert werden. Weiterführend wird Abschnitt 6.7 die Implementierung der Constrainttechnik im *Compiere Monitor* durch die Verwendung der *Rules* noch eingehender erläutern.

- ***Import- und Export***

  Anforderungen, wie die Verwendung mehrerer Compiere-Datenbankserver oder die Sicherheit vor einem Datenverlust, erhöhen den Bedarf des Im- und Exports von Daten. Zusätzlich neben den schon obligatorisch gewordenen Im- und Exportfunktionen der Datenbankmanagementsysteme, bietet der *Compiere Monitor* zu diesem Zweck ebenfalls eine solche Funktion. Sie basiert auf der *eXtensible Markup Language (XML)*, die technisch weiterführend in Abschnitt 6.5 beschrieben wird. In

Verbindung mit dem zugrunde liegenden XML-Schema ermöglicht die XML eine zeiteffiziente Integration der Daten in ein Programm und macht sie potenziell wiederverwendbar für jede andere Applikation.

Innerhalb des *Compiere Monitor* erlaubt diese Funktion den Im- sowie Export beliebiger Tabellen. Dabei werden die Tabellen als Ganzes im- bzw. exportiert. Zusammen mit der Benutzung der Primärschlüssel als Referenzzeiger resultiert daraus jedoch das Problem der mehrfachen Vergabe eines Primärschlüssels. Wird eine Tabelle exportiert, werden die Primärschlüssel aller Zeilen mit exportiert. Dies dient dem Erhalt der einzelnen Referenzen. Werden diese Zeilen nun importiert ohne ihren ursprünglichen Primärschlüssel vollständig wiederherzustellen, kann es zu Inkonsistenzen kommen. So können z.B. die Primärschlüssel von Referenzen einer importierten Zeile auf bereits existierende Zeilen zeigen. Werden diese bereits existierenden Zeilen nicht durch die ursprünglichen Zeilen ersetzt, so zeigt die Referenz zwar auf eine Zeile, jedoch auf eine andere als zum Zeitpunkt des Exports.

Das Fehlen eines Sekundärschlüssels macht es daher nötig, dass alle bestehenden Datensätze einer Tabelle während eines Imports gelöscht werden. Die Verwendung eines solchen Sekundärschlüssels würde eine Identifikation jeder einzelnen Zeile auf höherer Ebene gestatten und das genannte Problem lösen. Sie impliziert jedoch das Einführen einer weiteren Spalte und macht das Gesamtmodell damit komplexer. Eine zweite Möglichkeit zur Verhinderung des genannten Problems ist die Nutzung eines XML-Schemas, welches die einzelnen Charakteristika jedes Elements genau definiert und damit die generische Behandlung auf Ebene von Tabellen, Zeilen und Spalten verhindert. Sie würde einen Import der Daten ohne vorheriges Löschen ermöglichen, ist jedoch aufwendiger zu modellieren.

Da dieses Problem nur innerhalb der Importfunktion besteht, wurde auf eine Implementierung eines der beiden genannten Lösungsalternativen aus Zeitgründen verzichtet.

- ***Datenbankprüfung***

  Mit den ungefähr 50 Tabellen des *Compiere Monitor* erwuchs der Bedarf einer Möglichkeit zur Prüfung der Datenbanken auf strukturelle Konsistenz. Zu diesem Zweck wurde eine Datenbankprüfung innerhalb des *Compiere Monitor* implementiert. Sie überprüft alle Tabellen des *Compiere Monitor* auf Korrektheit. Dabei wird untersucht, ob die Tabelle existiert, sie die richtigen Spaltennamen und Datentypen besitzt und im Falle der Verwendung einer *Oracle Database*, die Sequenz zum Ermitteln des nächsten freien Primärschlüssels existiert. Die Ergebnisse dieser Prüfungen werden ermittelt und angezeigt. Ist eines der Elemente fehlerhaft, kann diese Tabelle vom Nutzer neu erstellt werden. Neben dieser Prüfung durch den Anwender, wird während der Initialisierung der Tabellen beim Start des *Compiere Monitor* eine automatische Prüfung der Existenz von Tabellen, Spalten und Sequenzen vorgenommen. Beim Fehlen eines der Elemente wird dieses automatisch neu erstellt und im Startprozess fortgefahren. Damit wird die Benutzung des *Compiere Monitor* ohne eine manuelle Einrichtung von Tabellen durch den Benutzer

ermöglicht und die Tabellenverwaltung erleichtert.

- ***Dynamische Logging-Tabellen***

  Als Erweiterung der statischen Tabellen des *Compiere Monitor* besteht zusätzlich die Möglichkeit der Einrichtung von weiteren, frei definierbaren Tabellen. Aufgrund ihres dynamischen Charakters finden sie jedoch keinerlei Anwendung innerhalb der bereits erwähnten *Aspektgeneratoren*. Sie dienen vielmehr der manuellen Einbeziehung in Aspektklassen. So können sie z.B. verwendet werden, um Daten aus Aspektklassen heraus in der Datenbank zu speichern bzw. von ihr zu lesen. Da die Zielsetzungen dieser Arbeit jedoch besonders auf den Aspekten der Aspektgeneratoren liegt, bietet der *Compiere Monitor* aus Zeitgründen keine weiterführende Unterstützung für die Verwendung der *dynamischen Logging-Tabellen*. Lediglich die Funktion ihrer Erstellung bzw. Änderung wird mit Hilfe der grafischen Benutzeroberfläche des *Compiere Monitor* unterstützt. Sie wurden an dieser Stelle nur der Vollständigkeit halber erwähnt und werden in dieser Arbeit keine weitere Beachtung erfahren.

6.4 Die Compiere-Aspektgeneratoren

Im vorangegangenen Abschnitt wurde ein Überblick über die Funktionalitäten des Compiere Monitor gegeben. Eine wichtige Rolle spielt dabei der Compiere-Aspektgenerator. Er ist ein Verbund mehrerer Aspektgeneratoren, von denen jeder jeweils eine Teamklasse erstellt. Zusammen generieren sie die Menge der Aspekte, die im Rahmen dieser Arbeit definiert werden, um die Wandlungsfähigkeit von Compiere zu erhöhen. Diese Aspekte, ihre Generatoren und Konfiguratoren sowie weitere zugehörige Elemente sollen in diesem Abschnitt näher betrachtet werden. Abbildung 6.1 dient hierbei als Grundlage. Sie gibt eine Übersicht über die Hierarchien der erstellten Team- und Rollenklassen, wobei zielgerichtet abstrahiert wurde - Methoden, Attribute und Bindungen wurden in diesem Modell ausgespart.

6.4.1 Aspektklassen

Als Erstes werden nun die Teamklassen beschrieben, welche durch die Generatoren erzeugt werden. Sie sind gruppiert über ihre funktionalen Zielsetzungen und bilden so insgesamt 6 Teamklassen. Im Zuge einer Beschreibung dieser Teamklassen soll jedoch an dieser Stelle noch eine Modifikation des ObjectTeams-Modells innerhalb dieser Arbeit erwähnt werden. Das Modell bezeichnet ursprünglich alle Klassen eines Teams als Rollen, auch wenn diese keine `playedBy`-Beziehung besitzen. Zur besseren Unterscheidung werden Klassen ohne diese Relation in der vorliegenden Arbeit jedoch einfach als `Klasse` bezeichnet. Klassen mit einer `playedBy`-Relation zu einer Basisklasse werden hingegen, konform zum Modell, weiterhin als `Rolle` oder `Rollenklasse` benannt.

Im Folgenden werden nun die einzelnen Teamklassen einer genaueren Untersuchung unterzogen.

- `CompiereTeam`

  Die abstrakte Teamklasse `CompiereTeam` ist Superklasse aller generierten Aspektklassen und erbt selbst nur noch von `org.objectteams.Team`, d.h. von der obersten Teamklasse einer jeden Teamklassenhierarchie. Sie enthält keine Rollen, sondern dient vielmehr als eine Ansammlung von Methoden und Klassen, die durch alle ihre Sub-Teamklassen Verwendung finden können. So beinhaltet sie unter anderem Methoden für den Zugriff auf oft benutzte Compiere-Daten, wie z.B. der Benutzer- oder Rollen-ID des aktuell angemeldeten Benutzers. Da diese einfachen *Getter*-Methoden ähneln werden sie nicht weiter betrachtet werden. Daneben besitzt die Teamklasse jedoch noch eine Reihe von Klassen, von welchen eine die abstrakte Klasse `WFNodeItem` ist. Diese bietet Methoden, welche im Kontext von Compiere-Workflowelementen Anwendung finden. Wird ein Workflowknoten eines beliebigen Typs (Siehe 4.1.8) aktiviert, so sorgen diese Methoden unter anderem für eine Speicherung des Quellknoten. Mit diesem kann bei Beendigung der Knotenaktivität eine Evaluierung der Workflowerfolgskriterien (Siehe 6.4.2) durchgeführt werden. Für jeden Typ eines Workflowknoten gibt es eine dazu passende Subklasse, welche die unterschiedlichen Ausprägungen eines jeden Typs beachtet.

  Diese Subklassen sind `WFNodeItemWindow`, `WFNodeItemForm`, `WFNodeItemWorkflow`, `WFNodeItemTask` und `WFNodeItemProcess`. Entsprechend ihrer Bezeichnung unterstützen sie dabei die Workflowknoten des Typs *Window*, *Form*, *Workflow*, *Task* und *Process*. Sie werden in den folgenden Betrachtungen der einzelnen Teamklassen noch weiter erwähnt werden.

  Listing 6.1 zeigt beispielhaft die Nutzung der Subklasse `WFNodeItemWindow` durch die Rolle `OT_APanel`. Dabei öffnet die Aktivierung eines Workflowknotens des Typs *Window* das entsprechende *Window*-Objekt. Dies wird durch die `initPanel`-Bindung der Rolle in Zeile 44 erkannt. Die dazugehörige Rollenmethode prüft in den Zeilen 33-35, ob die Öffnung des `Window` durch einen Workflowknoten veranlasst wurde. Ist dies der Fall, so führt sie die Methode `startWFNode(...)` der Superklasse `WFNodeItemWindow` aus. Diese dient unter anderem der Speicherung genau dieses Workflowknoten, welcher durch seine Bedeutung im Folgenden auch nur Quellknoten genannt wird. Die Prüfung selbst wird dabei über die Methode `getStartingWFNode()` aus Zeile 17 vorgenommen. Sie durchsucht die Menge aller aktuell aktivierten Workflowknoten auf einen Knoten hin, welcher zum Start des *Window*-Objektes führt. Ist ein solcher Knoten vorhanden so bedeutet dies, dass das `Window`-Objekt von einem Workflowknoten gestartet wurde und die Methode gibt das dazugehörige `WFNode`-Objekt als Rückgabewert zurück. Um hierbei den unterschiedlichen Workflowknotentypen gerecht zu werden, wurden die bereits genannten `WFNodeItem`-Subklassen eingeführt. Jede dieser Klassen implementiert ihrer Ausprägung entsprechend die abstrakte Methode `getStartingWFNode()` der Klasse `WFNodeItem` (Zeile 6). Sie erlauben damit einen generischen Zugriff auf den Workflowknoten, welcher zum Start des Basisobjekts der von ihnen erbenden Rollenklasse führte.

Das Schließen des `Window`-Objekts wird in Zeile 47 durch die Überwachung der `dispose()`-Methode erkannt. In Zeile 41 führt diese dann über die Methode `evaluateWFNode()` der Rollensuperklasse eine abschließende Evaluierung der Erfolgskriterien des Quellknotens aus, welcher für den Start des *Window*-Objektes verantwortlich war. Diese Evaluierung ist unter anderem der Grund für die Speicherung des Quellknoten durch die Methode `startWFNode(...)` aus Zeile 8.

Neben den bereits erwähnten Elementen zeigt das Listing weitere Attribute und Klassen, die im Kontext anderer Teamklassen von Interesse sind. Sie werden Bestandteil der anschließenden Teamklassenbeschreibungen sein.

Listing 6.1: Ausschnitt aus der Teamklasse `CompiereTeam`

```
1  public abstract team class CompiereTeam{
2    public abstract class WFNodeItem {
3      private WFNode sourceNode = null;
4      private static ArrayList successWFNodes = new ArrayList();
5      protected static ArrayList startingWFNodes= new ArrayList();
6      protected abstract WFNode getStartingWFNode(int itemID);
7
8      protected void startWFNode(WFNode arg0, int winNo){
9        sourceNode = arg0;
10       ...
11     }
12     protected void evaluateWFNode(){...}
13     ...
14   }
15
16   public class WFNodeItemWindow extends WFNodeItem {
17     protected WFNode getStartingWFNode(int adWindowID){...}
18   }
19
20   public class WFNodeItemWorkflow extends WFNodeItem {
21     protected WFNode getStartingWFNode(int adWorkflowID){...}
22     protected void startWorkflow(){
23       successWFNodes.clear();
24     }
25   }
26   ...
27 }
28
29 public team class CompiereWindowMonitor extends CompiereTeam
       when(MAspectTeamClass.isTeamActive(x){
30
31   public class OT_APanel extends WFNodeItemWindow playedBy org.
         compiere.apps.APanel{
32     public void initPanel(int adWindowID){
33       WFNode startingWFNode = getStartingWFNode(adWindowID);
34       if(startingWFNode != null){
35         startWFNode( startingWFNode, getCurWinNo() );
36       ...
```

```
37      }
38
39      public void dispose(){
40        ...
41        evaluateWFNode();
42      }
43
44      void initPanel(int adWindowID) <- after boolean initPanel(
            int arg0, int arg1, MQuery arg2)
45        with {adWindowID  <-  arg1} when (
              MCompiereAspectLogActivity.isActiveWindowAction(...));
46
47      void dispose() <- before void dispose();
48      int getCurWinNo() -> get int m_curWindowNo;
49    }
50  }
```

- `CompiereFormMonitor`

  Eine weitere, zu überwachende Compiere-Komponente ist die der *Forms*. Zu diesem Zweck wurde die Teamklasse `CompiereFormMonitor` eingeführt. Sie unterstützt zum einen das Monitoring von Workflowknoten des Typs *Form* und beinhaltet daneben auch Bindungen und Methoden zum Loggen des Benutzerverhaltens innerhalb eines jeden *Form*.

  Beide Funktionalitäten werden dabei über die Rollenklasse `OT_FormFrame` mit `playedBy`-Beziehung an der Klasse `org.compiere.apps.form.FormFrame` implementiert. Zum Zwecke des Erkennens sich öffnender und schließender *Form*-Objekte, besitzt sie zwei Bindungen an den Basismethoden `openForm(..)` und `dispose()`. Funktionell sind diese äquivalent zu den Bindungen `initPanel(..)` und `dispose()` der Rolle `OT_APanel` des Teams `CompiereWindowMonitor` aus Listing 6.1. Eine weitere Beschreibung sowie Erwähnung im Listing 6.2 soll daher für die Rolle `OT_FormFrame` ausbleiben.

  Die Klasse `FormFrame` bildet als Unterklasse von `javax.swing.JFrame` den Rahmen, in welchem alle individuellen *Form*-Klassen eingebettet werden. Eine solche *Form*-Klasse stellt zu diesem Zweck jeweils selbst ein `javax.swing.JPanel` dar und implementiert für eine generische Behandlung durch das umgebene `FormFrame` lediglich das Interface `FormPanel`. Dieses beinhaltet die beiden Methoden `init(...)` und `dispose()`, dessen Verwendung im weiteren Verlauf noch deutlicher werden wird.

  Die Methoden und Komponenten der Klasse `FormFrame` bilden den statischen Teil innerhalb der Anzeige eines *Forms*, ähnlich wie dies bei *Window*-Objekten der Fall ist (Siehe 4.2.5). Die einzelnen *Form*-Klassen hingegen können variieren und bilden somit den dynamischen Teil eines jeden `FormFrame`-Objekts. Für ein umfassendes Monitoring müssen so auch, neben der Beobachtung der statischen Elemente eines `FormFrame` mittels der Rollenklasse `OT_FormFrame`, alle relevanten Ereignisse

innerhalb der dynamischen *Form*-Panel überwacht werden. Da die *Form*-Panel jedoch Eigenimplementierungen darstellen und sie nicht wie *Window*-Objekte mittels Datenbankeinträge generiert werden, besitzt man keine Informationen über ihre Komponenten. Eine zu den *Window*-Objekten vergleichbar generische Überwachung ist somit nicht möglich.

Stattdessen wird eine Überwachung hier unter Verwendung der *Form*-Methoden erreicht. Die zu überwachenden Methoden werden zu diesem Zweck über den Aspektkonfigurator *Formularelemente* (Siehe 6.4.2) definiert. Sie werden bei der Generierung der Teamklasse durch den Compiere-Aspektgenerator herangezogen, wobei er für jede definierte Methode jeweils eine Rollenmethode mit passender Bindung generiert. Listing 6.2 stellt ein Beispiel für solch eine generierte `CompiereFormMonitor`-Teamklasse dar und dient als Grundlage der folgenden Betrachtungen. Die Rollenklasse `OT_VInvoiceGen` adaptiert hierbei das Compiere-Form `VInvoiceGen`. Dieses spiegelt als *Form*-Panel den dynamischen Teil des `FormFrame` wieder und erfordert daher eine Defintion der zu überwachenden Formmethoden innerhalb des Aspektkonfigurators. Konkret sind dies in diesem Beispiel die Methoden `cmdEval()` und `executeQuery()`. Da im Beispiel keine weiteren Methoden anderer *Form*-Klassen zur Überwachung definiert wurden, generiert der Compiere-Aspektgenerator, neben der statisch erzeugten Rollenklasse `OT_FormFrame`, darüber hinaus keine weiteren Klassen.

Zu jedem *Form*, für welches eine überwachte Methode definiert wurde, wird durch den Aspektgenerator eine eigene Rollenklasse definiert. Sie werden in Abbildung 6.1 durch die abstrakte Basis- und Rollenklasse *Formklassen* bzw. *dynamische Rollen* dargestellt. Konkret bedeutet dies am Listing 6.2, dass hier in Zeile 16 eine Rollenklasse mit entsprechender Bindung an das zu überwachende Form `FormInvoiceGen` erzeugt wurde. Die Rollenklassen erhalten dabei den Namen des *Forms* mit dem Präfix `OT_`. Sie beinhalten zu jeder der definierten *Form*-Methoden eine eigene Rollenmethode mit denselben Namen und Parametern. Diese werden bei der Definition einer *callin-after*-Bindung an der entsprechenden *Form*-Methode auf Rollenseite genutzt. Jede zur Überwachung definierte *Form*-Methode besitzt demnach in der Rollenklasse jeweils eine *after*-Bindung sowie eine Rollenmethode mit gleicher Signatur. Diese sind im Beispiel in den Zeilen 17-26 zu erkennen. Sie zeigt auch die Verwendung der Loggingmethode `logIndividual(...)` aus Zeile 2. Durch sie wird eine Speicherung der übergebenen Daten in der Datenbank ausgeführt.

Darüber hinaus erbt jede Rollenklasse eines `Form` von der Klasse `OT_FormRole`. Sie ist Teil desselben Teams und beinhaltet eine `init`-Methode, deren Zweck im Folgenden kurz beschrieben wird. Ausgangspunkt ist dabei der Bedarf nach einem Zugriff auf die *Form*-interne Fensternummer. Die Fensternummer selbst wird innerhalb der erwähnten Logging-Methode aus Zeile 2 benötigt und im Anschluss an den einzelnen Teamklassenbeschreibungen noch näher erläutert. Eine individuell erzeugte `Form`-Klasse besitzt über die Schnittstelle `FormPanel` hinaus keine generischen Zugriffsmöglichkeiten auf *Form*-spezifische Methoden. Da diese Schnittstelle keine Getter-Methode für die *Form*-interne Fensternummer anbietet, fehlt somit

in allen `Form`-Klassen die Möglichkeit eines generischen Zugriffs auf diese Fensternummer. Anstelle einer *callout*-Bindung an dieser Getter-Methode wird nun die Methode `init(int winNo, FormFrame fFrame)` des Interface `FormPanel` genutzt. Da jede `Form`-Klasse diese Schnittstelle implementieren muss, wird so eine generische Zugriffsmöglichkeit auf die Fensternummer geschaffen. Dieser Methode wird in Zeile 11 die Fensternummer als Parameter übergeben, die anschließend in Form eines Rollenattributs gespeichert wird. Die Methode selbst erhält dabei ihre Bindung durch die erbende Rollenklasse (Zeile 24).

Zusammen mit der Rolle `OT_FormFrame` erlauben die generierten Rollenklassen der einzelnen *Forms* so eine Überwachung der nötigen Benutzeraktivitäten.

Listing 6.2: Ausschnitt einer generierten `CompiereFormMonitor`-Teamklasse

```
1  public team class CompiereFormMonitor extends CompiereTeam when(
       MAspectTeamClass.isTeamActive(x){
2    private void logIndividual(String desc, int winNo, String
       method){
3      //do logging
4    }
5    public class OT_FormFrame extends WFNodeItemForm playedBy
       FormFrame{
6      ...
7    }
8
9    public class OT_FormRole {
10     protected int winNo = -1;
11     protected void init(int winNo, FormFrame fFrame){
12       this.winNo = winNo;
13     }
14   }
15
16   public class OT_VInvoiceGen extends OT_FormRole playedBy
       FormInvoiceGen{
17     public void cmdEval(){
18       logIndividual("Evaluation", winNo, "public void cmdEval()"
           );
19     }
20     private void executeQuery(){
21       logIndividual("Query-Ausführung", winNo, "private void
           executeQuery()");
22     }
23
24     void init(int winNo, FormFrame fFrame) <- after void init(
         int winNo, FormFrame fFrame) when (
         MCompiereAspectLogActivity.isActiveFormAction(...));
25     void cmdEval() <- after void cmdEval() when (
         MCompiereAspectLogActivity.isActiveFormAction(...));
26     void executeQuery() <- after void executeQuery() when (
         MCompiereAspectLogActivity.isActiveFormAction(...));
27   }
28 }
```

- `CompiereWorkflow`

  Als abstrakte Teamklasse ähnelt *CompiereWorkflow* funktionell der Teamklasse *CompiereTeam*. Sie bietet oft verwendete Methoden für Klassen, die sich mit den Compiere-Workflowelementen beschäftigen. Das sind z.B. Methoden zum Einholen aller Workflowknotens eines Workflows oder zur Ermittlung der Position eines Workflowknotens innerhalb des zugehörigen Workflows. Im Falle der, durch den Aspektgenerator erstellten Teamklassen, findet sie durch die Klasse *CompiereWorkflowMonitor* Verwendung. Wie schon bei der Teamklasse `CompiereTeam` ist auch hier eine erweiternde Nutzung durch die Generierung weiterer Subklassen problemlos möglich.

- `CompiereWorkflowMonitor`

  Diese Teamklasse bildet den Ausgangspunkt bei der Überwachung der Compiere-Workflowaktivitäten. Sie dient dem Erkennen von Aktivierungen einzelner Workflowknoten und beinhaltet darüber hinaus Elemente, welche zum weiteren Monitoring der aktivierten Objekte betragen. Zu diesem Zweck besitzt sie zwei Rollen für die Compiere-Klassen `org.compiere.apps.wf.WFPanel` und `org.compiere.apps.AMenu`. Listing 6.3 dient der Veranschaulichung der nun folgenden Erläuterung und zeigt die wichtigsten Codeelemente dieser Teamklasse.

  Die Compiere-Klasse `WFPanel` ist die grafische Komponente zur Darstellung eines Workflows. Sie stellt alle Workflowknoten durch entsprechende Symbole dar und erlaubt deren Aktivierung mittels entsprechender Benutzereingaben. Die dazugehörige Rolle `OT_WFPanel` dient hierbei der Überwachung der einzelnen Knotenaktivierungen. Zur Erfüllung dieser Aufgabe besitzt sie in Zeile 8 eine Bindung an der Basismethode `start(WFNode)`. Diese wird bei einer Knotenaktivierung aus dem `WFPanel` heraus aufgerufen, wobei die Bindung eine Ausführung der Rollenmethode aus Zeile 5 impliziert. Diese fügt der Liste `startingWFNodes` aus der Teamklasse `CompiereTeam` (Listing 6.1, Zeile 5) den startenden Workflowknoten hinzu. Die Liste dient der Sammlung aller derzeit startenden Workflowknoten, wobei die einzelnen Knoten wieder aus dieser Liste gelöscht werden, sobald dessen zugehöriges Objekt aktiviert wurde, wie z.B. nach dem Öffnen eines `Window`-Objektes. Wie bereits während der Beschreibung der Teamklasse `CompiereTeam` angedeutet wurde, findet diese Liste innerhalb der Implementierungen der Methode `getStartingWFNode()` einer jeden `WFNodeItem`-Subklasse Verwendung. Weitere Erläuterungen zu dieser Knotenliste finden sich im Anschluss in Abschnitt 6.4.1.

  Daneben existiert eine zweite Rolle `OT_AMenu`, welche die Klasse `AMenu` adaptiert. Neben dem Erkennen einfach geöffneter Workflows dient sie ebenso dem Monitoring von sich öffnenden Sub-Workflows, d.h. den Workflows, welche über Workflowknoten bereits geöffneter Workflows aktiviert werden. Dazu wird mit der Bindung aus Zeile 19 das Öffnen eines Workflow bzw. Sub-Workflow überwacht. Die dazugehörige Rollenmethode aus Zeile 12 geht so bei Öffnung eines Workflows eine Reihe von Schritten durch. Zum einen führt sie die Methode `startWorkflow()` ihrer Superklasse aus. Diese findet sich in Zeile 22 des Listing 6.1 wieder und

dient dem Zurücksetzen des Workflowzustandes. Konkret löscht sie dabei die Elemente der Liste `successWFNodes` des `CompiereTeam`. Diese Liste beinhaltet alle Workflowknoten, die im Sinne ihrer definierten Erfolgskriterien (Siehe 6.4.2) erfolgreich ausgeführt wurden. Weitere Details dazu finden sich im Abschnitt 6.4.1. In den Zeilen 14-16 wird im Anschluss geprüft, ob der sich öffnende Workflow ein Sub-Workflow ist. Die dabei verwendete Methodik entspricht der bereits beschriebenen aus den Zeilen 33-35 des Listing 6.1. Ist der sich öffnende Workflow ein Sub-Workflow, d.h. er wurde von einem Workflowknoten aus aktiviert, wird die bereits beschriebene Methode `startWFNode(...)` der Superklasse `WFNodeItem` ausgeführt (Listing 6.1, Zeile 8). Innerhalb dieses Methodenaufrufes wird ein `get-callout` auf dem Basisattribut `m_WindowNo` genutzt, der die aktuelle Fensternummer des sich öffnenden `WFPanel`-Objekts darstellt. Die weitere Bedeutung dieser Fensternummer wird im Anschluss noch erläutert werden.

Listing 6.3: Ausschnitt aus der Teamklasse `CompiereWorkflowMonitor`

```
1
2  public team class CompiereWorkflowMonitor extends
       CompiereWorkflow when(MAspectTeamClass.isTeamActive(x)) {
3
4    public class OT_WFPanel playedBy WFPanel {
5      private void start(WFNode node){
6        startingWFNodes.add(arg0);
7      }
8      void start(WFNode node) <- before void start(WFNode node);
9    }
10
11   public class OT_AMenu extends WFNodeItemWorkflow playedBy
         AMenu {
12     protected void startWorkFlow(int wfID){
13       startWorkflow();
14       WFNode startingWFNode = getStartingWFNode(wfID);
15       if(startingWFNode != null)
16         startWFNode( startingWFNode, getWinNo() );
17       ...
18     }
19     void startWorkFlow(int wfID) <- before void startWorkFlow(
           int wfID);
20
21     int getWinNo() -> get int m_WindowNo;
22   }
23 }
```

- `CompiereMonitor`

  Die anfangs erwähnte Aufteilung der einzelnen Teamklassen nach ihren Funktionen macht nur bei Gruppierungen Sinn, deren funktioneller Umfang eine eigene Teamklasse rechtfertigen. Funktionen die schwer zu einer anderen Teamklasse zuzuordnen sind und selbst eine eigenständige Kapselung nur bedingt rechtfertigen, werden

innerhalb der Teamklasse *CompiereMonitor* implementiert. Sie ist daher die quantitativ am stärksten ausgeprägte Teamklasse der Aspektgeneratoren. So enthält sie zum Beispiel eine Rolle `OT_AEnv` für die Klasse `org.compiere.apps.AEnv`. Sie dient der Überwachung von Funktionen, die nicht an ein bestimmtes Anzeigeobjekt, wie einem *Window* oder *Form*, gebunden sind, sondern vielmehr in jedem dieser Anzeigeobjekte verfügbar ist, wie z.B. der Compiere-Taschenrechner oder -Kalender.

Daneben enthält diese Teamklasse auch eine Rolle zur Initialisierung der Loggingsitzung beim Start von Compiere und weitere Rollen zur Überwachung von Aktivitäten des Typs *Task* und *Process*. Einige von ihnen finden in Abschnitt 6.4.1 noch weitere Erwähnung. Ihre Implementierung ähnelt den Rollen des `CompiereWindowMonitor` und soll daher an dieser Stelle keine nähere Betrachtung finden.

- `CompiereWindowMonitor`

  Ein weiterer, funktioneller Bedarf liegt in der Überwachung der Benutzeraktionen innerhalb eines darstellenden Elements vom Typ *Window*. Die Gruppierung dieser Funktionen findet dabei ihren Ausdruck in der Teamklasse `CompiereWindowMonitor`. So werden durch sie Änderungen an Datenelementen sowie Aktionen am *Window* selbst erkannt und ausgewertet. Implementiert sind diese Funktionen über zwei Rollenklassen welche nun weiter beschrieben werden.

  Zum einen dient die Rollenklasse `OT_GridController` als Adaption der Compiere-Klasse `org.compiere.grid.GridController` dem Monitoring von Änderungen an den grafischen Elementen, die die Tabellendaten repräsentieren. Diese Elemente stellen den dynamischen Teil eines `Window`-Objektes dar (Siehe 4.2.5) und können mittels der Basismethode `vetoableChange(...)` überwacht werden. Zeile 7 des Listing 6.4 zeigt die dazugehörige Bindung. Innerhalb der Rollenmethode aus Zeile 6 wird eine Überprüfung des geänderten Feldes vorgenommen. Wurde das Feld bisher noch nicht geändert, so wird der Zeitpunkt der Änderung an diesem Feld mit Hilfe des `java.util.Hashtable`-Attributs `editedField` gespeichert. Erneute Änderungen führen hingegen nur zu einem Aktualisieren des Änderungszeitpunktes des Feldes in diesem Hashtable. Da das das Attribut durch beide Rollen dieser Teamklasse genutzt wird, ist es in Form eines Teamattributs implementiert.

  Die zweite Rollenklasse adaptiert die Compiere-Klasse `org.compiere.apps.APanel` und heisst `OT_APanel`. Sie wurde bereits im Listing 6.1 auszugsweise dargestellt und findet daher keine erneute Erwähnung im Listing 6.4. Einige Feinheiten sollen jedoch kurz erwähnt sein. So wurde bereits erwähnt, dass Änderungen an Datenfeldern eines *Window* innerhalb des Attributs `editedField` gespeichert werden. Diese werden erst in Form von entsprechenden Logging-Einträgen in der Datenbank gespeichert, wenn der Benutzer die Änderungen speichert oder das `Window`-Objekt als solches schliesst. Zu diesem Zweck werden diese Einträge unter anderem innerhalb der `dispose()`-Methode aus Zeile 39 des Listing 6.1 gespeichert und das Hashtable `editedField` zurückgesetzt. Weitere Details dazu sind in Abschnitt 6.4.1 zu finden.

Da die Rollenklassen dieser Teamklasse durchweg statisch erzeugt und dabei durch keinerlei Aspektkonfiguratoren beeinflusst werden, soll an dieser Stelle auf eine weitere Betrachtung verzichtet werden.

Listing 6.4: Ausschnitt aus der Teamklasse `CompiereWorkflowMonitor`

```
1  public team class CompiereWindowMonitor extends CompiereTeam
       when(MAspectTeamClass.isTeamActive(x)) {
2
3    private Hashtable editedFields = new Hashtable();
4
5    public class OT_GridController playedBy GridController {
6      public void vetoableChange(PropertyChangeEvent e){...}
7      void vetoableChange(PropertyChangeEvent e) <- after void
           vetoableChange(PropertyChangeEvent e) when (
           MCompiereAspectLogActivity.isActiveWinDatachange(...));
8      ...
9    }
10   ...
11 }
```

Neben den beschriebenen Details jeder einzelnen Teamklasse besitzen diese darüber hinaus eine Gemeinsamkeit, die durch den *Compiere Monitor* als solches vorgegeben wird. Jede nicht abstrakte Teamklasse erhält durch den *Aspect-Builder* bei Erstellung des Aspektklassen-Code automatisch einen *Guard*. Dieser ist Bestandteil der Implementierung des *Teamclass-Schedule*-Konfigurators (Siehe 6.4.2) und wird in den gezeigten Listings dieses Abschnitts allgemein ausgedrückt durch *`when(MAspectTeamClass.isTeamActive(x)`*. Die aufgerufene Methode dient dabei der Aktivitätsprüfung eines Teams. Dabei wird ihr der Primärschlüssel des zu prüfenden Teams übergeben, der durch die Variable *x* dargestellt ist. Sie wird im Prozess der Erzeugung des Aspectcodes durch den *Aspect-Builder* mit dem tatsächlichen Primärschlüssel des jeweiligen Teams ersetzt. Hat die Teamklasse zusätzlich noch einen benutzerdefinierten Guard zugewiesen bekommen, so wird der automatisch generierte *Guard* mittels einer boolschen *UND*-Verknüpfung mit dem benutzerdefinierten *Guard* kombiniert.

Darüber hinaus besitzt eine Vielzahl aller erzeugten Bindungen ebenfalls einen Guard. Sie wurden aus strukturellen Gründen nicht im einzelnen genannt, sind jedoch in den Listings zu finden. Die Guards sind dabei alle der Form eines statischen Methodenaufrufes auf der Klasse `MCompiereAspectLogActivity`. Sie prüft, ob der entsprechende Aktivitätstyp durch den Aspektkonfigurator *Logging-Profile* für das aktuell angemeldete Compiere-Profil aktiviert ist. Je nach Verwendungszweck der Bindungen variiert die dabei verwendete Methode. So verwenden generelle Aktivitäten auf einem *Form* die Methode `isActiveFormAction(...)` (Listing 6.2, Zeile 24-26), wohingegen *Window*-Aktionen `isActiveWindowAction(...)` (Listing 6.1, Zeile 45) und Datenänderungen `isActiveWinDatachange(...)` (Listing 6.4, Zeile 7) nutzen. Jeder dieser Methoden werden dabei die Compiere-Profildaten in Form von Primärschlüsseln der entsprechenden Compiere-Tabellen übergeben. Konkret sind dies die Schlüssel des Compiere-

Benutzers, der Rolle sowie der Organisation und des Klienten. Sie bilden häufig verwendete Compiere-Daten und sind daher über Methoden der eingangs beschriebenen Superklasse `CompiereTeam` zugänglich.

Weitere Details zur Verwendung der automatisch erzeugten Team- und Bindungsguards sind Abschnitt 6.4.2 zu entnehmen.

Zusammenwirken aller Aspektklassen

Neben der Auflistung aller Teamklassen, die durch den Compiere-Aspektgenerator erstellt werden, soll weiterführend ein Einblick auf das Zusammenwirken dieser Teamklassen gegeben werden. Dies wird beispielhaft anhand einer Reihe von Benutzerinteraktionen erläutert. Zusammen ergeben sie ein typisches Anwenderszenario von Compiere, bei welchem der Benutzer Elemente eines Workflows ausführt. Zum Zwecke der besseren Übersicht werden sie in Form einer Aufzählung notiert und beschrieben.

1. ***Start von Compiere***

   Der Benutzer startet die Compiere-Hauptanwendung. Dabei wird, nachdem sich der Benutzer am Compiere-System angemeldet hat, eine Logging-Sitzung des *Compiere Monitor* erstellt. Diese bildet eine Art Container für alle darauf folgenden Logging-Einträge. Ihre Initialisierung ist innerhalb der Teamklasse `CompiereMonitor` über eine Bindung der Rollenklasse `OT_AMenu` implementiert. Die Bindung dient dabei dem Erkennen einer Ausführung der Methode `jbInit()` der Compiere-Klasse `AMenu`. Diese Methode ist Teil der Erstellung des Hauptfensters der Compiere-Hauptanwendung verwendet und wird einmalig bei dessen Start ausgeführt. Neben den Compiere-Benutzerdaten werden im Zuge der Initialisierung der Logging-Sitzung auch Zusatzinformationen über die Compiere-Umgebung geloggt, wie z.B. das verwendete Betriebssystem oder die Version der Java-Virtual-Machine. Sie dienen der späteren Zuordnung aller gesammelten Daten zu einem konkreten Benutzer und System.

2. ***Starten eines Workflows***

   Grafisches Hauptelement der Compiere-Clientanwendung ist eine Baumstruktur innerhalb des Hauptfensters, die der Darstellung aller verfügbaren Benutzerelemente dient. Darunter befinden sich unter anderem Elemente zum Starten von Workflows. Der Start eines solchen Workflows wird über die Rolle `OT_AMenu` der Teamklasse `CompiereWorkflowMonitor` überwacht und geloggt.

3. ***Starten eines Workflowknoten vom Typ Window***

   Workflows bestehen aus einer Aneinanderreihung von Knoten verschiedenen Typs. Im Folgenden wird von einem Knoten des Typs *Window* ausgegangen. Knoten dieses Typs öffnen ein Fenster, dessen Inhalt mit Hilfe der Datenbank generiert wird (Siehe 4.2.5).

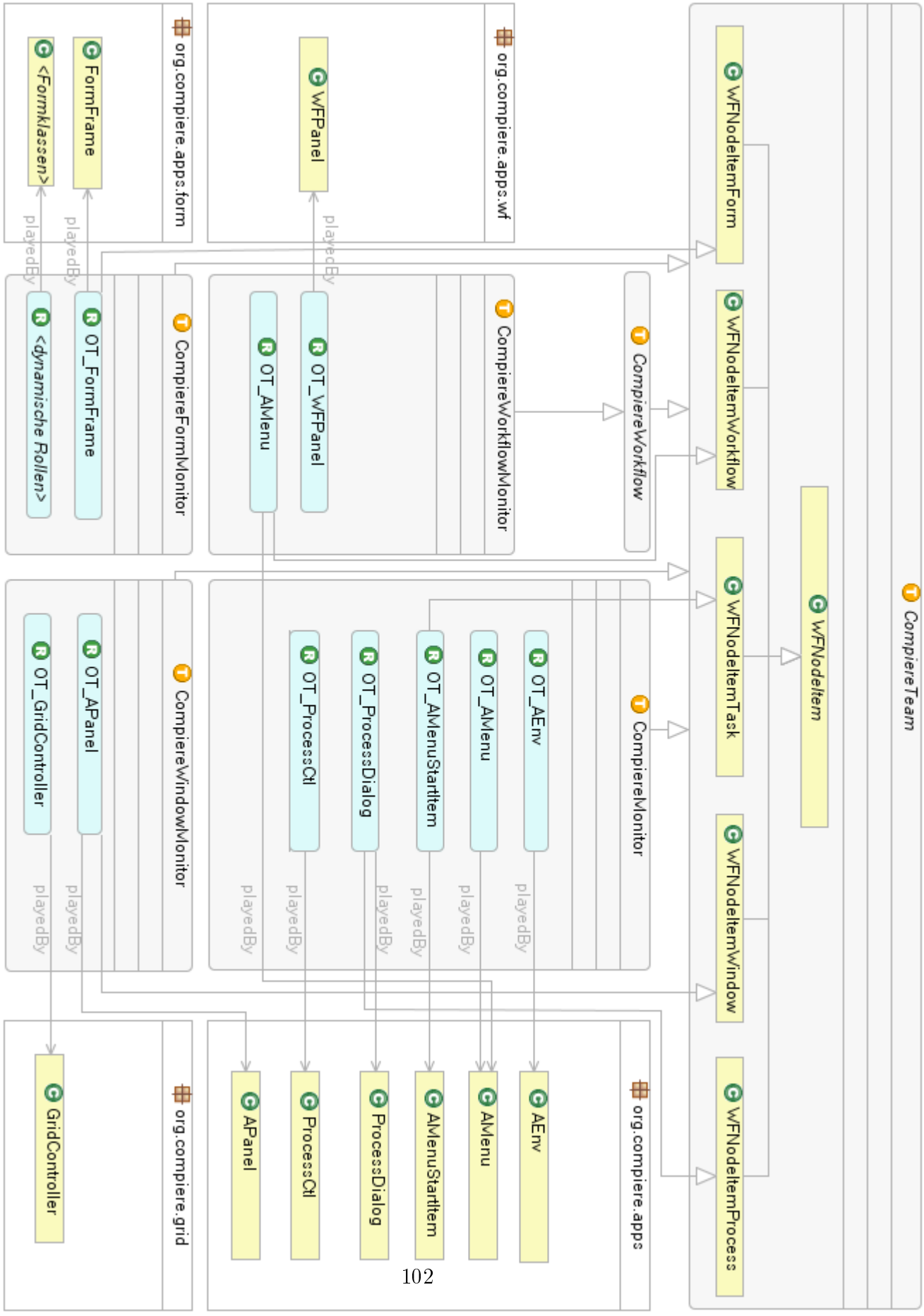

Abbildung 6.1: Modell der Aspekte des Compiere-Aspektgenerators. (Modelliert nach UML 2.1 mit Anlehnungen an UFA aus Abschnitt 5.3.5)

Der Start eines jeden Workflowknotens wird über die Rolle `OT_WFPanel` der Teamklasse `CompiereWorkflowMonitor` überwacht. Bei Erkennung eines Startvorgangs wird der startende Knoten in einer statischen `java.util.ArrayList` mit Namen `startingWFNodes` innerhalb des Superteams `CompiereTeam` als Referenz gespeichert (Listing 6.1, Zeile 5). Dies ist von Nöten, da der Prozess der Ausführung eines Workflowknotens dem normalen Ausführen außerhalb eines Workflows gleicht. Dies bedeutet, dass ein sich öffnendes Element vom Typ *Window* keine Information darüber besitzt, ob es von einem Workflow aus aufgerufen wurde oder nicht. Die Liste der startenden Workflowknoten wird verwendet, um genau dies für ein öffnendes Element zu ermitteln. Das Starten eines Workflowknotens geschieht nebenläufig über die Verwendung von Threads. Es ist damit möglich mehrere Workflowknoten zu aktivieren noch bevor die entsprechenden Objekte überhaupt geöffnet wurden. Darin liegt die Verwendung dieser Liste anstelle einer einfachen Referenz auf Workflowknoten begründet. Letztere würde hier eine nicht-parallele Abarbeitung voraussetzen.

Das sich öffnende Fenster wird im Anschluss über die `initPanel(...)`-Bindung der Rolle `OT_APanel` in der Teamklasse `CompiereWindowMonitor` erkannt. Im Zuge der Identifizierung der Quelle dieser Aktion wird geprüft, ob das *Window* durch einen der Workflowknoten aus der Liste `startingWFNodes` gestartet wurde. Dazu benutzt es die Methode `getStartingWFNode()` der Superklasse `WFNodeItemWindow`. Das Ergebnis dieser Prüfung entscheidet über den Typ des Loggingeintrages. Wurde es durch einen Workflowknoten gestartet, so wird darüber hinaus die Methode `startWFNode(...)` der Superklasse `CompiereTeam` aufgerufen. Sie entfernt den Quellknoten des gestarteten *Window*-Objekts wieder aus der Liste `startingWFNodes` und speichert ihn für eine spätere Verwendung in Form des Attributs `sourceNode` der Superklasse `WFNodeItem` ab (Listing 6.1, Zeile 3).

4. ***Ändern und Speichern von Daten innerhalb des Fensters***

   Wie in Abschnitt 4.2.5 beschrieben, stellen die einzelnen Datenfelder innerhalb eines *Window* Tabelleninhalte dar. Änderungen an einem dieser Datenfelder bedeutet eine Änderung der Tabellendaten. Diese werden über die Rolle `OT_GridController` innerhalb der Teamklasse `CompiereWindowMonitor` erkannt und direkt in einem `java.util.Hashtable` der Teamklasse gespeichert (Listing 6.4, Zeile 3). Das geänderte Datenfeld dient dabei als Schlüssel. Der Wert wird bei jeder Änderung am Datenfeld aktualisiert. Er enthält ein Element des Typs `java.util.Date` und hat semantisch die Bedeutung des Zeitpunktes der zuletzt getätigten Änderung. Änderungen an Datenfeldern werden also nicht direkt bei jeder Änderung geloggt, sondern registriert und zeitlich der letzten Änderung gleichgesetzt.

   Das Loggen selbst findet zum Zeitpunkt einer Speicherung der Daten statt oder bei Verlassen des *Window* selbst, d.h. beim Schließen des Fensters. Im Falle der Speicherung durch den Benutzer wird die Liste der geänderten Datenfelder im Anschluss zusätzlich noch zurückgesetzt. So werden anschließend nur die Änderungen geloggt, welche seit dem letzten Speichervorgang stattgefunden haben. Das Spei-

chern selbst wird über die Rolle `OT_APanel` der Teamklasse `CompiereWindowMonitor` erkannt und führt, neben einem entsprechenden Loggingeintrag für die Aktion selber, zu einem generieren von Loggingeinträgen für alle geänderten Datenfelder. Selbige Schritte werden beim Schließen des Fensters vorgenommen.

5. ***Schließen des Fensters***

   Das Schließen eines *Window* wird über eine Bindung an der `dispose()`-Methode der Compiere-Klasse `APanel` erkannt (Listing 6.1, Zeile 47). Sie veranlasst, wie im obigen schon erwähnt, eine Generierung von Logging-Einträgen für alle Datenfeldänderungen und logt den Schließvorgang an sich. Da das *Window* potentiell durch einen Workflowknoten gestartet worden sein kann, wird eine Methode zur Evaluierung der zugewiesenen Erfolgskriterien des Quellknotens aufgerufen. Sie ist Teil der Rollen-Superklasse `WFNodeItem` und verwendet als Quellknoten die, beim Öffnen des Fensters gesetzte Referenz des Rollenattributs `sourceWFNode`. Zur Evaluierung selbst werden die Definitionen der Workflowerfolgskriterien herangezogen, die im späteren Kapitel 6.4.2 erläutert werden. Das Ergebnis dient der abschließenden Bewertung des Workflowknotens als erfolgreich bzw. nicht erfolgreich.

   Im Zuge dieser Prüfung wird ebenfalls der gesamte Workflow auf Erfolg hin getestet. Dazu besitzt die Teamklasse `CompiereTeam` ein statisches `java.util.ArrayList` mit Namen `successWFNodes` (Siehe Listing 6.1, Zeile 4). Dieses wird beim Start eines Workflows bereinigt und beinhaltet alle Workflownoten, die bei der genannten Evaluierung als erfolgreich gewertet wurden. Enthält diese Liste alle Workflowknoten des Workflows, so wird dieser selbst als erfolgreich gewertet und dementsprechend in seinem Logging-Eintrag vermerkt.

   Jeder unterschiedliche Typ von Workflowknoten, wie z.B. `Form` oder `Process`, besitzt ebenfalls einen solchen Aufruf dieser Evaluierungsmethode an den Stellen ihrer zugehörigen Rollenklassen, die ein Verlassen des ausgeführten Elements signalisieren. So wird eine Evaluierung neben der beschriebenen `APanel`-Klasse z.B. auch bei Aufruf der `dispose()`-Methode der Compiere-Klassen `FormFrame` und `ProcessDialog` durchgeführt. Sie sind alle Subtypen grafischer Java-Klassen, wie z.B. `javax.swing.JPanel`, bei welchen diese Methode das Schließen ihrer selbst bedeutet.

Die beschriebenen Aktivitäten sind Teil der herkömmlichen Arbeitsschritte in Compiere und erläutern den detaillierten Zusammenhang zwischen den einzelnen Rollen- und Teamklassen sowie das Teilen benötigter Daten über verschiedene Attribute ihrer Superklassen. Ein Detail wurde dabei jedoch außen vorgelassen, das der späteren Zuordnung der Logging-Ereignisse zueinander. Compiere erlaubt die gleichzeitige Verwendung unterschiedlichster Elemente. So kann der Benutzer zum Beispiel an zwei `Window`-Elementen parallel arbeiten. Eine Auswertung der aufgezeichneten Ereignisse auf Basis des Ereigniszeitpunktes ist somit ungenügend, da die Ereignisse aus unterschiedlichen Fenstern heraus aufgetreten sein können.

Aus diesem Grund arbeitet das Logging des *Compiere Monitor* mit den internen Fensternummern von Compiere. Diese werden für jedes, sich öffnende Fenster vergeben, sei

dies nun ein Fenster zur Darstellung eines `Form`, `Window`, `Process`, `Report` oder `Task` (Siehe 4.2.5). So hat jedes Fenster seine eindeutige Zuweisung zu einer Fensternummer und kann damit genau voneinander unterschieden werden. Sie wird bei jedem Logging-Eintrag verwendet, um die Zuweisung des Ereignisses zu einem bestimmten Fensterelement zu gewährleisten. Die Fensternummer selbst wird dabei entweder über entsprechende Bindungen ermittelt und in Form eines Rollenattributes gespeichert, oder sie wird direkt mittels `get-callout`-Bindungen von der Basisklasse bezogen. Ersteres findet dabei lediglich innerhalb des dynamischen Teils der Teamklassen `CompiereFormMonitor` Anwendung und wurde bereits in Abschnitt 6.4.1 eingehend erläutert.

Wie gezeigt wurde, ist das Logging des *Compiere Monitor* über eine Verkettung verschiedenster Rollen- und Teamklassen implementiert worden. Sie gibt einen detaillierten Überblick über das Benutzerverhalten und hilft damit beim Erkennen von Änderungsbedarf im Kontext der Wandlungsfähigkeit. Konkrete Auswirkungen sollen an dieser Stelle jedoch nicht genannt werden. Sie sind Thema des späteren Kapitels 7.1.

6.4.2 Aspektkonfiguratoren

Die genannten Teamklassen decken funktionell den Rahmen eines Logging der Benutzeraktivitäten sowie der lokalen Compiere-Einstellungen ab. Um diese Funktion gezielt einsetzen zu können, wurden Konfiguratoren implementiert, mit welchen gezielte Änderungen am Aspektverhalten vorgenommen werden können. Sie sind in Form von grafischen Eingabemasken im *Compiere Monitor* integriert und sollen in diesem Abschnitt näher beschrieben werden.

- ***Workflow-Erfolgskriterien***

  Die Workflow-Erfolgskriterien definieren für jedes der Workflowknoten Kriterien, mit welchen geprüft werden kann, ob der jeweilige Workflowknoten erfolgreich abgearbeitet wurde. Die Charakteristik dieser Kriterien wird dabei über den Typ des Workflowknoten bestimmt. So können z.B. Benutzer bei Knoten des Typs *Window* potentiell mehr Änderungen oder Aktivitäten ausführen als beim Anstossen eines *Process*. Wie schon im vorherigen Abschnitt beschrieben wurde, werden die definierten Erfolgskriterien bei Verlassen bzw. Beenden des jeweiligen Knotenelements evaluiert. Das Ergebnis gibt Aufschluss darüber, inwieweit der Benutzer die erwarteten bzw. vorgegebenen Aktionen innerhalb des Workflowknotenelements ausgeführt hat. Wird das Kriterium dabei nur teilweise oder gar nicht erfüllt, gilt der Workflowknoten als nicht erfolgreich ausgeführt und der gesamte Workflow wird dementsprechend als unvollständig gewertet.

  Ein Kriteriumelement kann bei einem Knoten des Typs *Window* z.B. das Ändern des Wertes eines Datenfeldes oder das Drücken eines Buttons sein. Die hierbei verwendbaren Fensterelemente werden direkt aus den Compiere-Tabellen der Datenbank bezogen und bieten damit eine Vielzahl von Kontextinformationen (Siehe 4.2.5). Knoten dieses Typs besitzen dadurch den größten Detailgrad bei der Definition ihrer Erfolgskriterien. Ein *Form* stellt hingegen eine Individualentwicklung

dar, d.h. er besitzt selbst keine Repräsentationen seiner Komponenten innerhalb der Datenbank. Vergleichbar detaillierte Kriteriumsdefinition stehen aus diesem Grund hier nicht zu Verfügung. Kriterien werden in diesem Fall über die Angabe von Methodenaufrufen sowie allgemeiner Fensteraktionen, wie das Öffnen und Schließen, definiert. Für den Typ *Process* werden die Möglichkeiten noch restriktiver, da dieser nur einen Dialog mit Optionen zum Drucken und zum Starten eines Prozesses beinhaltet. Alle anderen Workflowknotentypen besitzen hingegen keinerlei Definitionsmöglichkeiten eines Erfolgskriteriums, da diese keine Benutzerinteraktionen zulassen.

Jedes der genannten Kriteriumselemente kann Teil eines Erfolgskriteriums für einen Workflowknoten sein. Die Kombination dieser einzelnen Elemente eines Kriteriums findet unter Verwendung boolscher Operationen statt. So können verschiedene Elemente über die Operatoren *UND*, *ODER* und *NOT* miteinander verknüpft werden. Die Auswertung des Gesamtausdrucks ist Teil der Evaluierung der Erfolgskriterien eines Workflowknoten. Die Definition des Kriteriums findet auf grafischer Ebene in Form eines Baumes statt in welchem die Blätter den einzelnen Elementen entsprechen und die Astgabeln eine boolsche Operation darstellen. Damit ist eine genaue Definition von gewünschtem Benutzerverhalten innerhalb eines Workflowknotenelements möglich.

- ***Formularelemente***

  Wie bei den *Workflow-Erfolgskriterien* schon erwähnt wurde, spielen die Methoden der *Forms* eine wichtige Rolle. Sie dienen als Ausgangspunkt des Formular-Monitorings, mit dessen Hilfe anschließend Aussagen über das Benutzerverhalten innerhalb des Formulars getroffen werden können. Dementsprechend muss jede Formularmethode, die in einem Workflow-Erfolgskriterium definiert wurde, auch für das Monitoring aktiviert sein. Die Aktivierung ist Bestandteil dieses Konfigurators. Er ermöglicht die Definition von zu überwachenden Methoden eines Formulars mitsamt einer semantischen Beschreibung derselben. Aus diesen Definition werden über die Compiere-Aspektgeneratoren entsprechende Rollenklassen innerhalb der Teamklasse `CompiereFormMonitor` gebildet. Sie sind in Abbildung 6.1 zusammenfassend durch die Rollenklasse <*dynamische Rollen*> dargestellt.

- ***Compiere-Einstellungen***

  Das Überwachen der individuellen Benutzereinstellungen für Compiere, wie die Fenstergrösse oder die verwendeten Schriftarten und Farben, hilft beim Erkennen von wiederkehrenden Benutzeranforderungen und ermöglicht damit die Etablierung eines Standards, der allen Benutzerwünschen gerecht wird. Diese Benutzereinstellungen werden direkt nach der Initialisierung der Logging-Sitzung einmalig geladen und in Form von entsprechenden Logging-Einträgen in der Datenbank gespeichert. Welche der Einstellungen dabei mit einbezogen werden, wird über diesen Konfigurator gesteuert. Die Liste der möglichen Einstellungen wird hierbei aus den statischen Attributen der Compiere-Klasse `org.compiere.util.Ini` bezogen, von welchen einige auch innerhalb der Compiere-`properties`-Datei Anwendung finden.

- ***Logging-Profile***

  Um dem rollenbasierten Ansatz von Compiere zu folgen, wurden die Logging-Einstellungen des *Compiere Monitor* unter Verwendung von Profilen implementiert. Sie erlauben die Definition der zu überwachenden Elemente auf Basis der Benutzer-, Rollen-, Klienten- und OrganisationsID des angemeldeten Compiere-Benutzers. Für jedes Profil können spezifische Aktivitäten zur Überwachung aktiviert werden. Somit lassen sich z.B. für verschiedene Rollen unterschiedliche Objekte überwachen. Die einzelnen Einstellungen entscheiden über die Überwachung einer Aktivität zur Laufzeit von Compiere. Sie werden innerhalb der Teamklassen über *Guards* an den entsprechenden Bindungen bzw. Abfragen an den passenden Codestellen implementiert (Siehe 6.4.1).

- ***Teamclass-Schedules***

  Ein weiterer Konfigurator dient der Planung der Aktivität von Teamklassen. Dabei können für jede Teamklasse Zeiträume definiert werden, in welchen sie aktiv sind. Ist eine Teamklasse zu einem Zeitpunkt nicht aktiv, so sind auch alle Rollenklassen und von ihr erbenden Teamklassen inaktiv. Existieren keiner dieser optionalen Zeitraumdefinitionen, so ist eine Teamklasse zu jedem Zeitpunkt aktiv. Diese Technik erlaubt somit eine zeitgesteuerte Modellierung von Aspektverhalten. Implementiert sind sie über *Guards* (Siehe 5.3.3) auf Teamklassen-Ebene. Hierbei werden zur Laufzeit der Aspekte die Zeiträume des entsprechenden Teamclass-Schedule aus der Datenbank herangezogen, um eine Prüfung auf Aktivität zum Zeitpunkt der Nutzung eines Teamklassenelements vorzunehmen. Die einzelnen Zeiträume sind durch diesen Konfigurator frei zu definieren und erlauben so eine flexible Anpassung.

Alle Einstellungen der genannten Aspektkonfiguratoren werden persistent in der Datenbank gehalten. Je nach Konfigurator finden sie während der Generierung durch den Compiere-Aspektgenerator oder auch direkt zur Laufzeit der Aspekte Anwendung.

So werden die Konfiguratoren *Formularelemente* sowie *Compiere-Einstellungen* innerhalb des Compiere-Aspektgenerators verwendet. Sie dienen der Erstellung angepasster Team- bzw. Rollenklassen. Änderungen an den Einstellungen dieser beiden Konfiguratoren erfordern daher von dem Benutzer eine Neugenerierung der Aspekte durch den Compiere-Aspektgenerator.

Alle anderen Konfiguratoren beinhalten Einstellungen, welche direkt innerhalb des Aspektcodes genutzt werden. Sie sind integraler Bestandteil der Team- und Rollenklassen. So wird z.B. bei Eintreten eines Ereignisses geprüft, ob der zugehörige Ereignistyp für den aktuell angemeldeten Compiere-Benutzer für ein Logging aktiviert ist. Die persistenten Einstellungen in der Datenbank, die in diesem Fall durch den Konfigurator *Logging-Profile* definiert werden, entscheiden hierbei, ob das Ereigniss geloggt wird oder nicht. Diese Einstellungen werden typischerweise auf dem Compiere-Klienten nur einmal während des Startvorgangs geladen und lokal vorgehalten. Sie dienen fortan als Quelle aller Abfragen während der gesamten Laufzeit. Änderungen an den Einstellungen

der *Logging-Profile* erfordern für deren Aktivierung somit einen Neustart der entsprechenden Compiere-Klienten. Um dies auch dynamischer zu gestalten, wurde ein `Thread` implementiert, welcher während der gesamten Laufzeit des Klienten aktiv ist und die relevanten Tabellen regelmässig neu lädt. Dies erlaubt die Änderungen an den entsprechenden Konfiguratoren innerhalb des *Compiere Monitor* bei gleichzeitiger Übernahme dieser Änderungen durch alle laufenden Compiere-Klientanwendungen. Um den dabei entstehenden Datenverkehr zu verringern, besteht im *Compiere Monitor* die Möglichkeit zur Deaktivierung dieser Funktionalität sowie der Änderung des zeitlichen Intervalls, in welchem ein Neuladen der Tabellen stattfindet.

Daneben besitzt der *Compiere Monitor* auch eine Funktion zur Betrachtung der gespeicherten Logdaten. Dieser *Log-Viewer* dient der zeitlichen Einordnung der gespeicherten Ereignisse. Dabei werden die Ereignisse über die bereits erwähnte *Logging-Sitzung* gruppiert. Eine solche Sitzung bezeichnet den gesamten Zeitraum, in welcher ein Nutzer an einem Compiere-Klienten angemeldet ist. Der Log-Betrachter ist somit Werkzeug zur Auswertung der gesammelten Daten. Um den Rahmen der Arbeit nicht zu sprengen, wird aber auf eine detailliertere Beschreibung des *Log-Viewer* verzichtet.

6.5 Verwendete Techniken

Die bisherigen Verlauf dieses Kapitels lag die Aufmerksamkeit primär auf den unterschiedlichen Funktionalitäten des *Compiere Monitor*. Dieser Abschnitt befasst sich nun mit den Techniken zur Implementierung dieser Funktionen im speziellen sowie des *Compiere Monitor* im allgemeinen. Aus Gründen der besseren Verständlichkeit wird hierbei erneut nach funktioneller Sicht gruppiert.

- ***Java & Compiere***

  Die Frage nach der zu verwendenden Java-Version bei der Entwicklung des *Compiere Monitor* wurde geprägt durch die Gegebenheiten von ObjectTeams und Compiere. Über einen längeren Zeitraum der Entwicklungsphase hinweg wurde durch ObjectTeams/Java nur Java 1.4.x, in Form des *Object Teams Development Tooling* 0.8.x, unterstützt. Dies implizierte die Verwendung einer Compiere-Version, die ebenfalls auf dieser Version basiert. Zur Anwendung kam daher Compiere in dem *Release 2.5.2e*, welches das letzte unter Verwendung dieser Java-Version ist. Mit dem *Release 2.5.3a* wurde Compiere auf Java 1.5 umgestellt.

  Während der Entwicklung des *Compiere Monitor* wurden jedoch weitere Entwicklungsschritte seitens ObjectTeams/Java getätigt. Dabei wurde unter anderem eine Unterstützung für Java 1.5.x im *Object Teams Development Tooling* implementiert und so die Möglichkeit geschaffen, die neueren Compiere-Releases zu nutzen. Aus Konsistenzgründen gegenüber dem schon entwickelten Code wurde jedoch auf eine Nutzung der neuen Sprachfeatures von Java 1.5 für den *Compiere Monitor* verzichtet. Nichtsdestotrotz wird für den *Compiere Monitor* die Virtual Machine

von Java 1.5 genutzt. Nicht zuletzt deshalb, weil die verwendete XML-Technik des Import- und Export diese voraussetzt.

- ***Team-Creator und Compile-Check***

  Der Compile-Check des *Compiere-Monitor* dient der Überprüfung von Teamklassen auf Kompilierbarkeit hin. Er nutzt dabei die Technik des *Team-Creator*, welcher auf Compiere-Klienten für die Kompilierung der Teamklassen zuständig ist. Zum Kompilieren verwendet der *Team-Creator* dabei die *Eclipse Java Development Tools (JDT)*[3] und hier im speziellen die *JDT-Core*-Komponente. In Kombination mit den nötigen Jar-Archiven des ObjectTeams/Java ermöglicht sie das kompilieren der Teamklassen. Dafür werden diese zuvor durch den *Aspect-Builder* generiert und lokal abgespeichert. Anschließend werden diese `.java`-Dateien dann mittels des Aufrufes der statischen Methode `org.eclipse.jdt.internal.compiler.batch.Main.compile(..)` kompiliert. Die dabei auftretenden Fehler werden zum einen auf der Systemconsole ausgegeben und darüber hinaus innerhalb des Compile-Checks des *Compiere Monitor* sichtbar gemacht.

- ***Import und Export***

  Wie in Abschnitt 6.3 bereits erwähnt wurde, basiert die Funktion des Im- und Exports auf der XML-Technologie. Sie ist ein offener Standard und erlaubt die leichte Wiederverwendung der gespeicherten Daten. Im *Compiere Monitor* wurde diese Technik mittels *Java Architecture for XML Binding (JAXB)*[4] implementiert. Sie erlaubt die Bindung von Java-Klassen an ein XML-Schema und damit die Translation von XML-Daten in eine Objektstruktur. Zur Anwendung kam dabei die zum Zeitpunkt der Arbeit aktuelle JAXB-Version 2.0. Sie benutzt viele der neuen Sprachfeatures von Java 1.5 und bildete damit die bereits erwähnte Anforderung der Nutzung von Java 1.5 für den gesamten *Compiere Monitor*.

- ***JAR-Archive und Classbrowser***

  Der *Compiere Monitor* dient zur Verwaltung und Erstellung von ObjectTeams/Java-Aspekten. Zur Erfüllung dieser Aufgabe ist es nötig, Informationen über Klassen und deren Subelementen, wie Methoden und Attribute, zu besitzen. Zu diesem Zweck können im *Compiere Monitor* Jar-Archive eingelesen werden. Aus ihnen werden alle `.class`-Dateien ausgelesen und auf die nötigen Informationen hin analysiert. Dies geschieht mittels *Java-Reflection*. Die ausgelesenen Daten bilden die Grundlage für die spätere Verwendung in den Paket-, Klassen-, Methoden- und Attributbrowsern. Diese Browser finden innerhalb des *Compiere Monitor* häufige Anwendung, z.B. bei der Auswahl von Basismethoden während der Definition von Rollenbindungen.

  Viele durch die Datenbank abgebildeten ObjectTeams-Elemente haben Bezüge zu Klassen, Methoden oder Attributen. So haben zum Beispiel Rollen durch die

[3] http://www.eclipse.org/jdt
[4] https://jaxb.dev.java.net

`playedBy`-Relationen einen Bezug zu einer Basisklasse oder Bindungen einen zu ihrer Basismethode. Innerhalb der Datenbank werden sie alle in textueller Form gespeichert, d.h. das Bezugsobjekt wird nicht als ganzes Objekt sondern mittels eines eindeutigen Identifier persistent gehalten. So wird z.B. eine Klassenmethode mittels ihrer Methodensignatur gespeichert, anstelle das gesamte Methodenobjekt serialisiert abzuspeichern. Dies hat zwei Gründe. Zum einen sind die beim *Reflection* eingesetzten Objekte, wie `java.lang.Class` oder `java.lang.reflect.Method`, alle nicht serialisierbar und darüber hinaus auch `final`, d.h. nicht durch Vererbung erweiterbar. Selbst wenn dies nicht so wäre, spart die genannte Methode trotzdem Speicherplatz und damit Datentransfer ein, da serialisierte Objekte einen wesentlich höheren Bedarf an zu speichernden Daten aufweisen.

Die genannten Identifier hängen jedoch von den zugrundeliegenen Objekten ab. So kann z.B. die Methodensignatur je nach Version ihrer Klasse variieren. Um diese Inkonsistenzen zu vermeiden erfordert dies, dass alle Elemente dieselben Jar-Archive als Grundlage ihrer verfügbaren Klassen benutzen. Resultierend daraus werden eingelesene JAR-Archive nicht lokal gespeichert, sondern ebenfalls zentral in der Datenbank abgelegt und von dort auch wieder ausgelesen.

Da Jar-Archive, wie im Falle der benötigten Java-Runtime-Bibliothek `rt.jar`, aber relativ gross sein können, wurde im *Compiere Monitor* ein lokaler Cache implementiert. Eingelesene Jar-Archive werden dabei, neben der Speicherung in der Datenbank, auch lokal in Form einer temporären Datei abgespeichert. Zum Zwecke der Wiedererkennung wird zu einem eingelesenen Jar-Archiv eine *Cyclic Redundancy Check (CRC)*-Prüfsumme ermittelt. Sie wird sowohl in der Datenbank als auch im lokalen Cache gespeichert. Wird die Tabelle der Jar-Archive neu eingelesen, so dient die Spalte der CRC-Prüfsumme zur Identifikation des einzulesenden Jar-Archivs. Die Spalte des eigentlichen Jar-Objektes wird bei diesem ersten Einlesezyklus ausgelassen. Stattdessen wird die CRC-Prüfsumme der eingelesenen Zeile genutzt, um im Cache nach dem Jar-Objekt zu suchen. Wird kein Objekt mit dieser CRC-Prüfsumme im Cache gefunden, so wird in einem zweiten Einlesezyklus nachträglich das vorher ausgelassene Jar-Objekt aus der Datenbank geladen. Wird hingegen ein Objekt mit dieser CRC-Prüfsumme im Cache lokalisiert, so wird dieser zweite Zyklus übersprungen und das Objekt direkt aus dem Cache gelesen. Dieses Vorgehen verringert das Volumen der Datenbankkommunikation und spart damit Zeit beim Einlesen der einzelnen Jar-Archive ein. Um den Rahmen dieser Arbeit nicht zu sprengen, wird auf eine detailliertere Erläuterung der Implementierung des Cache verzichtet.

- ***Datenpersistenz***

  Der Datenpersistenz kommt in verteilten Applikationen mit entfernter Datenhaltung eine besondere Rolle zu. Aus diesem Grund gibt es bereits eine Reihe von Persistenz-Frameworks für die Entwicklung verteilter Applikationen in Java. *Hibernate*[5] ist ein Beispiel für solch ein Framework. Es sorgt für ein Mapping von

[5] http://www.hibernate.org

Java-Klassen auf Datenbankobjekte. Die Verwendung solch einer Lösung birgt viele Vorteile, aber auch Nachteile, welche gegeneinander aufgewogen werden müssen. Die Entscheidung bei der Entwicklung des *Compiere Monitor* fiel dabei gegen die Verwendung eines Persistenz-Frameworks aus. Stattdessen wurde selbst eine Persistenzschicht implementiert, welche Objekte der Datenbank stark generisch behandelt. Eine ähnliche Funktionalität wäre nur mit ungerechtfertigtem Mehraufwand in die abbildenden Klassen von Persistenz-Frameworks, speziell *Hibernate*, zu implementieren. Des Weiteren erleichtert die derzeitige Implementierung das Ändern von Tabelleneigenschaften. So müssen hier z.B. bei Änderungen keinerlei Deskriptoren geändert werden. Natürlich bieten größere Persistenzschichten wie *Hibernate* auch eine ganze Reihe von Vorteilen, wie ein automatisches, kaskadiertes Löschen. Für die Entwicklung des *Compiere Monitor* und der Zielsetzung dieser Arbeit sind diese jedoch nebensächlich. Die genannten Punkte bilden die Rechtfertigung für die Entscheidung zu Gunsten der Implementierung einer eigenen Persistenzschicht, die weiterführend noch in Abschnitt 6.7 behandelt wird.

Neben den genannten Techniken wurden des Weiteren noch Techniken zur Erfüllung sekundärer Aufgaben verwendet, die aber keiner ausführlichen Erläuterung bedürfen. Diese sind unter anderem *JUnit*[6] zum Ausführen von Unit-Tests, welche jedoch aus zeitlichen Gründen marginal ausgebildet sind, und *Log4j*[7] zur Unterstützung der Loggingfunktionalitäten innerhalb des *Compiere Monitor*.

6.6 Datenbankmodell

Der *Compiere Monitor* ist eine verteilte Anwendung, welche auf der Verwendung von Datenbanken basiert. Diese dienen der Speicherung sämtlicher Datenobjekte mit Ausnahme der lokalen Einstellungen der Applikation selbst. Das zugrunde liegende Datenmodell soll in diesem Abschnitt näher vorgestellt werden.

Das Modell besitzt insgesamt 61 statische Tabellen wovon 12 schreibgeschützte Compiere-Tabellen sind. Jede Tabelle des Compiere Monitor trägt, zur besseren Unterscheidung gegenüber den Compiere-Tabellen, im Namen den Prefix `ASPECT_MGMT_`. Im Sinne der Vollständigkeit wird dieser im Modell beibehalten. Die Tabellen selbst lassen sich funktionell in 4 Gruppen aufteilen:

- ***Aspektmanagement***

  Tabellen dieser Gruppe beinhalten Daten der Aspektverwaltung. Sie dienen der Speicherung aller ObjectTeams/Java-Objekte, wie Team- oder Rollenklassen, sowie deren spezifischen Eigenschaften, z.B. den Aktivierungszeiträumen eines Teams.

- ***Compiere-spezifische Elemente***

[6] http://www.junit.org
[7] http://logging.apache.org/log4j

Die in Abschnitt 6.4 betrachteten Aspektgeneratoren generieren Aspekte zum Logging verschiedener Ereignisse. Die Ergebnisse dieses Logging werden in Tabellen gespeichert, welche dieser Gruppe angehören. Zusätzlich beinhaltet sie die Tabellen zur Speicherung der Einstellungen der Aspektkonfiguratoren (Siehe 6.4.2) sowie sämtliche, am Anfang erwähnten Compiere-Tabellen. Letztere dienen lediglich der semantischen Einordnung von Objekten in die Compiere-Umgebung und sind daher schreibgeschützt. So verweist z.B. ein Eintrag einer Logging-Sitzung auf Einträge der Tabellen `AD_ORG`, `AD_CLIENT`, `AD_ROLE` und `AD_USER`. Sie ermöglichen eine Identifikation des Benutzers, welcher die Sitzung über den Start des Compiere-Klienten initiiert hat.

- ***Konfiguration***

  Der *Compiere Monitor* besitzt eine Vielzahl von Objekten, die als Grundlage weiterführender Funktionalitäten dienen. So besitzt z.B. jede Methode eine bestimmte Sichtbarkeit. Die unterschiedlichen Methodensichtbarkeiten sind hierbei Zeilen der Tabelle `ASPECT_MGMT_METHODVISIBILITY` aus der Gruppe *Konfiguration* und werden durch die Tabellen der Team- und Rollenmethoden verwendet. Diese Tabelle stellt somit grundlegende Daten bereit, die Voraussetzung für die erfolgreiche Definition einer Methode sind. Diese und andere Tabellen wurden aufgrund ihrer Eigenschaften in dieser Gruppe zusammengefasst.

- ***Dynamische Logging-Tabellen***

  Das dynamische Logging dient der Funktion, Tabellen frei definieren zu können, um diese anschließend zum Zwecke des Logging zu nutzen. Dementsprechend sind diese Tabellen nicht statisch, d.h. nicht fest im Datenmodell verankert. Lediglich 2 statische Tabellen dienen der Verwaltung dynamischer Logging-Tabellen. Sie beinhalten die Eigenschaften, wie den Datentypen oder Namen, zu jeder dynamischen Spalte und Tabelle. Mit ihrer Hilfe wird der Zugriff auf die dynamischen Tabellen ermöglicht.

Abbildung 6.2 zeigt beispielhaft einige der Tabellen des Datenmodells. Im Speziellen werden hier Tabellen aus der Gruppe des *Aspektmanagements* dargestellt, welche die gesamten Team- und Rollenklassen abbilden. Da diese weitgehend selbsterklärend sind, wird auf eine detaillierte Erläuterung verzichtet.

Die Tabellen des Datenmodells werden durch das verwendete Datenbankmanagementsystem abgebildet. Dieses gibt die Möglichkeiten des Modells vor, wie zum Beispiel die maximale Länge der Tabellennamen oder die zu verwendenden Datentypen. Verschiedene Techniken des *Compiere Monitor* erlauben dabei den Austausch des Datenbankmanagementsystems. Sie werden ein Bestandteil des folgenden Abschnittes sein.

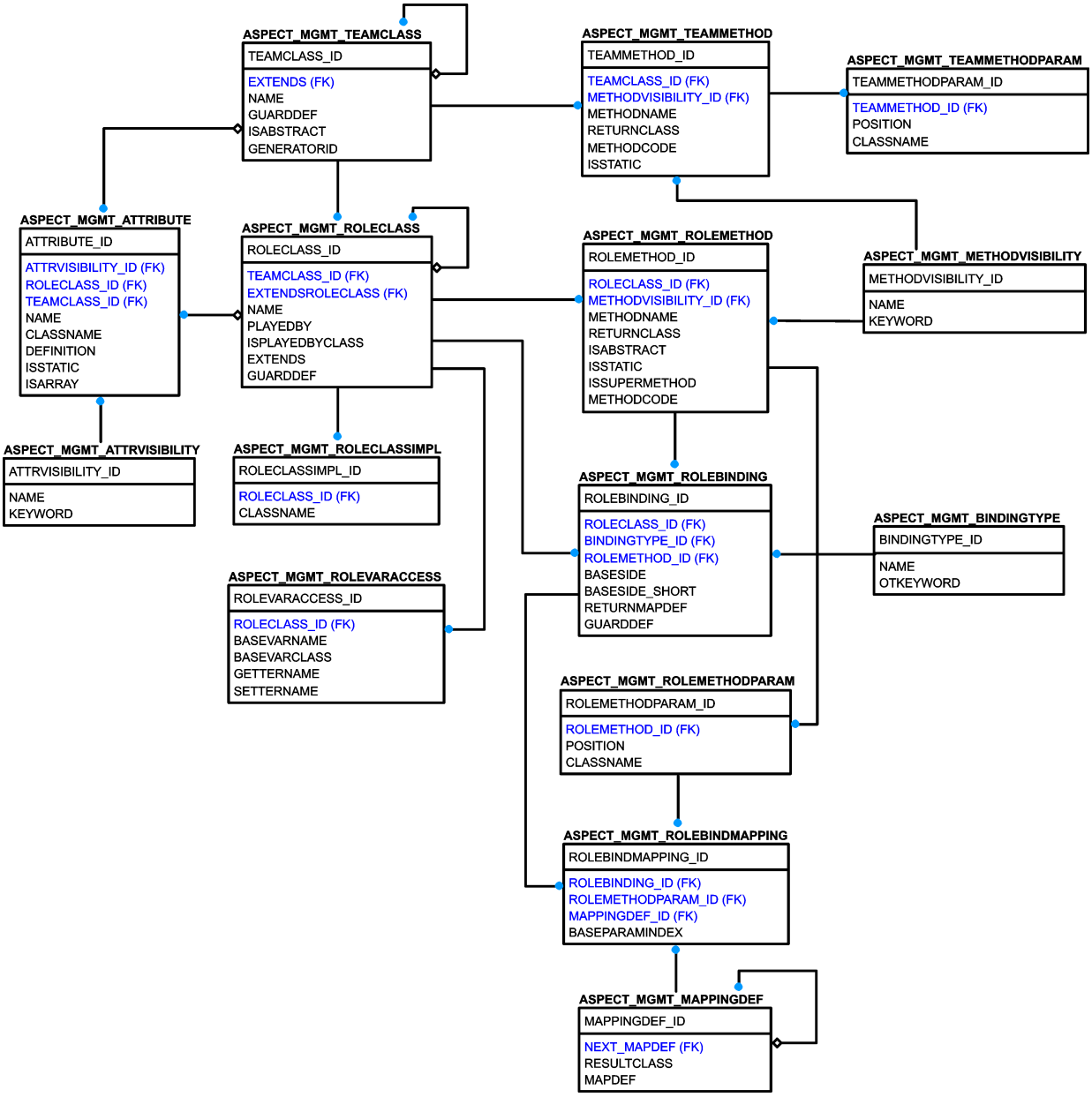

Abbildung 6.2: Vereinfachtes Entity-Relationship-Modell für Tabellen des Aspektmanagement zur Speicherung der Team- und Rollenklassen

6.7 Implementierung

Die bisherigen Abschnitte dieses Kapitel schilderten die Anforderungen und Zielsetzungen an die Entwicklung des *Compiere Monitor*. Sie gingen in 6.5 des Weiteren auch auf die verwendeten Techniken ein und deuteten damit schon Elemente der Implementierung an. Dieser Abschnitt soll diese nun vervollständingen und eine detailliertere Übersicht geben. Schwerpunkt liegt dabei auf den Elementen, die besonders von der Verwendung objektorientierte Konzepte profitieren sowie auf den implementierten *Design Patterns*.

6.7.1 Allgemeine Konzepte

Thema dieses Abschnittes sind die Klassenstrukuren, die für den *Compiere Monitor* besonders wichtige Funktionalitäten übernehmen. Sie werden im Einzelnen vorgestellt und näher erläutert. Aus strukturellen Gründen wird hier nun zuerst die generelle Abbildung von Tabellen erklärt.

Abbildung von Tabellen

Der *Compiere Monitor* bezieht seine Daten aus einer Datenbank. Diese werden für die Dauer der Laufzeit bzw. bis zum nächsten Neuladen der Tabelle im Speicher abgelegt. Zu diesem Zweck gibt es Klassen zur Abbildung dieser Daten. Sie werden nun beschrieben und finden in Abbildung 6.3 ihre grafische Repräsentation.

- `Table`

  Jede Tabelle der Datenbank, die durch den *Compiere Monitor* genutzt wird, besitzt eine eigene Tabellenklasse. Sie erben jeweils von der abstrakten Klasse `Table` und werden in Abbildung 6.3 zusammenfassend bezeichnet als *Tabellenklassen*. Diese Klassen beinhalten die Rules (Siehe 6.3) sowie die spezifischen Informationen über die Struktur der durch sie abgebildeten Tabelle. Sie bieten darüber hinaus jedoch kaum Funktionalitäten. Vielmehr implementieren sie abstrakte Methoden der Klasse `Table`. Diese abstrakten Methoden werden innerhalb der Klasse `Table` genutzt, um die generischen Methoden auf den individuellen Tabellenstrukturen anwenden zu können. So besitzt die Klasse `Table` z.B. lediglich Informationen darüber, dass es Spalten gibt. Die genauen Ausprägungen, wie die Spaltenanzahl oder deren Datentyp und Namen, sind ihr jedoch unbekannt. Informationen wie diese werden erst durch die Implementierungen der erbenden Klassen gegeben. Sie dient somit als Container verschiedenster Funktionen und Objekte, die zusammen erst den generischen Zugriff auf jedes Tabellenobjekt gewährleisten.

- `Column`

  Die innere Klasse `Column` der `Table`-Klasse dient der Kapselung der spezifischen Eigenschaften einer Tabellenspalte. Dies umfasst den Spaltennamen, dessen Datentyp sowie die Eigenschaft, ob die Spalte einen *NULL*-Wert annehmen kann.

Optional steht noch für die unterstützenden Datentypen die Eigenschaft der maximalen Länge zur Verfügung. Sie beschränkt die Eingabe von Werten auf eine bestimmte Länge, wie z.B. die Textlänge eines textuellen Datentyps. Diese Klasse dient lediglich der Modellierung von Tabelleneigenschaften innerhalb der, von `Table` erbenden, Tabellenklassen. In jeder Tabelle gibt neben möglichen anderen genau ein `Column`-Objekt, welcher als Primärschlüssel dient. Er dient der eindeutigen Identifizierung einer jeden Zeile.

- `Row`

  Die Klasse `Row` repräsentiert genau eine Zeile aus einer Tabelle. Durch diese starke Bindung an exakt eine Tabelle wurde sie ebenfalls als innere Klasse der Klasse `Table` implementiert. Die Klasse `Table` enthält dabei eine beliebige Anzahl von Instanzen der Klasse `Row`. Zusammen spiegeln sie damit sämtliche Zeilen einer Tabelle wieder. Als Abbildung einer Tabellenzeile enthält sie ebenfalls die einzelnen Werte der Tabellenspalten. Sie werden als Objekte des Typs `CMDataType` gespeichert und mit Hilfe des Namens der Spalte, zu welchen sie zugehörig sind, identifiziert.

- `CMDataType`

  Die abstrakte Klasse `CMDataType` ist Superklasse der Wrapperklassen, welche zur Speicherung der einzelnen Spaltenwerte einer Tabellenzeile genutzt werden. Jede dieser Wrapperklassen dient der Kapselung von Java-Datentypklassen, wie `Integer` oder `Boolean`, und bietet eine Reihe von Datentyp-spezifischen Methoden. Darüber hinaus implementiert jede der Subklassen eine Reihe abstrakter Methoden der Klasse `CMDataType`. Sie dienen der Einbettung individueller Implementierungen unter Verwendung generischer Zugriffsmethoden. So implementiert z.B. jede der Subklassen die Setter-Methode `setValue(Object obj)` entsprechend des zu wrappenden Datentyps. Diese Technik führt den generischen Ansatz der Klasse `Table` fort und erleichtert den Umgang mit den Tabellendaten.

  Die einzelnen Wrapperklassen sollen nun im Kurzen genannt werden:

  - `CMBoolean, CMString, CMDate, CMLong, CMFloat`

    Diese Klassen werden zusammenfassend genannt, da sie alle ähnlichen Charakters sind und selbst keine spezifische Rolle im *Compiere Monitor* übernehmen. Entsprechend ihrer Bezeichner sind sie Wrapper für die Java-Datentypklassen `Boolean`, `String`, `Date`, `Long` und `Float`.

  - `CMBlob`

    Diese Klasse ist ein Wrapper für einen Wert des Typs `java.lang.Object`. Zum Zwecke der Speicherung in der Datenbank implementiert sie zusätzlich die Schnittstelle `java.io.Serializable`. Dabei wird das `CMBlob`-Objekt in ein `byte[]`-Array transformiert und in Form eines Binärstreams die Datenbank geschrieben. Diese Wrapperklasse erlaubt somit das Speichern sämtlicher Objekte, welche selbst die Schnittstelle `java.io.Serializable` implementerien. Objekte die diese nicht implementieren, werden vorher durch die

`setValue(...)`-Implementierung dieser Klasse in den serialisierbaren Typ `java.lang.String` transformiert. Dieser Datentyp wird unter anderem bei Spalten verwendet, deren Werte über die herkömmlichen Größenbeschränkungen anderer Datentypen hinausgehen. So wird z.B. der Code einer Team- oder Rollenmethode in einem `CMBlob` gespeichert, da Code erstellt werden kann, der über die Maximallänge eines `CMString` hinausgeht. Daneben wird er unter anderem auch verwendet, um die bereits erwähnten Jar-Archive zu speichern.

– `CMInteger`

  Werte des Typs `java.lang.Integer` werden über diesen Wrapper gekapselt. Zusätzlich sind `Column`-Objekte, welche die Spalte der Primärschlüssel einer Tabelle repräsentieren, stets vom Typ `CMInteger`. Sie gewährleisten eine Unterscheidung der Zeilen anhand eines numerischen Wertes in einem ausreichend großem Zahlenbereich. Spalten, die eine Referenz auf Zeilen anderer Tabellen beinhalten, verwenden die Primärschlüssel der Zielzeile als jeweiligen Referenzzeiger und sind daher ebenfalls vom Typ `CMInteger`.

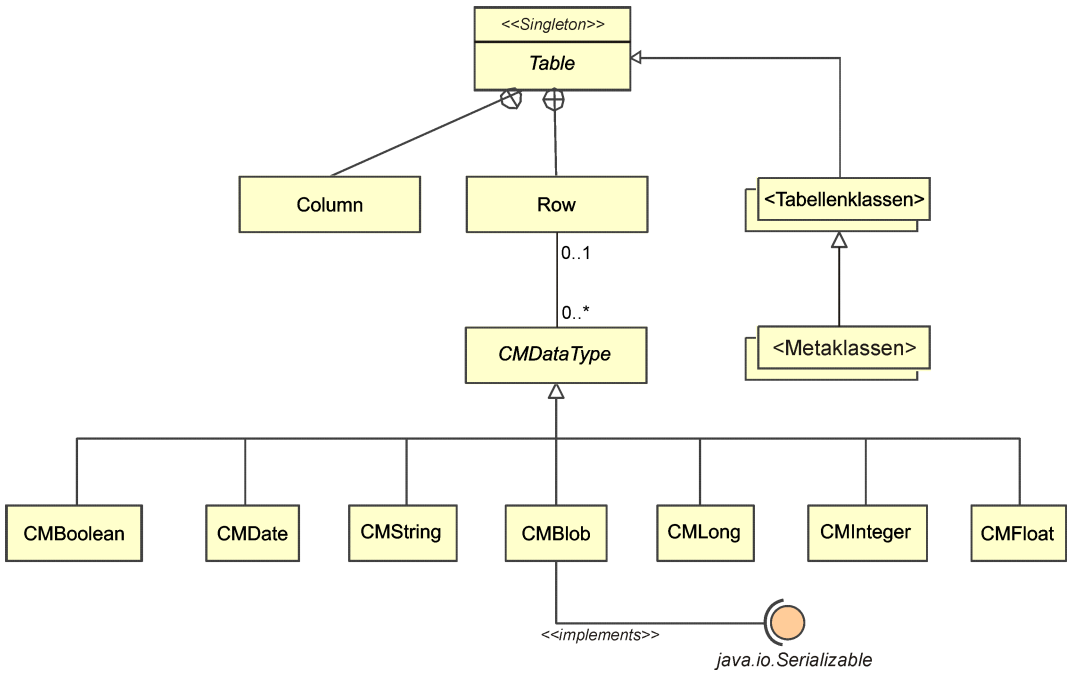

Abbildung 6.3: Klassendiagramm der Komponenten der Tabellenabbildung

Die Speicherung aller `Row`-Objekte innerhalb eines `Table`-Objektes ist mittels eines `java.util.Hashtable` implementiert, wobei der Primärschlüssel der Zeile als Indexwert dient. In einem `Row`-Objekt werden die einzelnen Objekte vom Typ `CMDataType` ebenfalls über eine `java.util.Hashtable` gespeichert. Hier dienen jedoch die Spaltennamen als Indices. Zusammen ergeben sie so eine Art Matrix aus Zeilen und Spalten und bilden damit die jeweilige Tabelle ab.

Ein Nachteil bei der Entscheidung gegen ein Persistenzframework (Siehe 6.5) ist das Fehlen eines Objektbezuges. Der *Compiere Monitor* arbeitet intern nicht mit ObjectTeams-bezogenen Objekten, wie z.B. mit Instanzen einer Klasse, die jeweils eine Bindung repräsentiert, sondern legt eine Art *Metaschicht* über die Tabellendaten. Diese Metaschicht ist durch eine Vielzahl von Klassen implementiert, wobei jede genau einer Tabellenklasse zugehörig ist und von dieser erbt. In Abbildung 6.3 werden sie als *Metaklassen* dargestellt. Jede dieser Metaklassen erweitert die dazugehörige Tabellenklasse um Funktionalitäten, welche die Tabellendaten mit Relationsbezügen und erweiterten Semantiken anreichert. Sie ähnelt dabei der Komposition von Daten durch die Klassen eines Persistenzframeworks, arbeitet jedoch stets nur mit den bereits genannten Elementen einer Tabelle, d.h. den `Row`- und `CMDataType`-Objekten.

Diese generische Sicht auf Tabellen- und Wrapperklassen erleichtert in ihrer Kombination den Umgang mit den Tabellendaten und fördert die Entwicklung weiterer generischer Komponenten. Einige werden in diesem Abschnitten noch weitere Erwähnung finden.

Filter

Zur Erleichterung des Umgangs mit den einzelnen Zeilen- und Spaltenobjekten einer Tabelle, wurden sogenannte *Filter* eingeführt. Sie ähneln dem Konzept der *Views* in Datenbankmanagementsystemen. Jedes dieser Filter ist dabei eine Klasse, welches das Interface `IFilter` implementiert. Dieses definiert eine Methode zur Überprüfung der Zugehörigkeit eines Objektes zu der gewünschten Objektmenge. Bezogen auf Tabellenobjekte können so Werteprüfungen auf den Spaltenwerten einer übergeben Zeile getätigt werden. Sie sind vergleichbar mit den *WHERE <column>=<value>*-Statements in der SQL.

Neben Filterklassen aus anderen Kontexten, bietet der Compiere Monitor zwei Arten der tabellenbezogenen Filter. Dies ist zum einen `FilterRowColumnLike`. Die Zielobjektmenge besteht hier aus Zeilen, die einen bestimmten Wert innerhalb einer Spalte besitzen. Der zweite Filter, `FilterRowColumnNotLike`, hat eine dazu inverse Bedingungsdefinition. Hier muss eine Zeilen einen Spaltenwert besitzen, der von dem im Filter definierten Wert abweicht. Um eine flexible Kombination dieser Filter zu erlauben, wurden darüber hinaus zwei weitere Filter implementiert, `FilterCombineAnd` und `FilterCombineOr`. Ihre Implementierung des Interface `IFilter` dient der boolschen Verknüpfung von zwei weiteren Filter-Objekten. Sie verketten diese Filter über eine *UND*- bzw. *ODER*-Relation. Da diese boolschen Filter ebenfalls die Schnittstelle `IFilter` implementieren, können sie selbst Operand eines boolschen Filters sein. Das erlaubt eine vielfältige Nutzung dieser Filter.

Als kurzes Beispiel soll hier die Tabelle `ASPECT_MGMT_ROLECLASS` der Abbildung 6.2 dienen. In dieser Tabelle gibt es die Spalten `TEAMCLASS_ID` und `PLAYEDBY`. Ein Filter, angewandt auf diese beiden Spalten, kann so alle Zeilenobjekte der Rollenklassen zurückgeben, welche Teil einer angegebenen Teamklasse sind und die in playedBy-Beziehung zu einer Basisklasse *baseA* oder *baseB* stehen. Listing 6.5 verdeutlicht die Definition dieses Beispielfilter unter Verwendung von Pseudo-Java:

Listing 6.5: Beispiel einer Definition von Filternklassen (Pseudo-Java)

```
1 FilterCombineAnd( FilterRowColumnLike(TEAMCLASS_ID, <Teamklasse>),
2   FilterCombineOr(
3     FilterRowColumnLike(PLAYEDBY, baseA) ,
4     FilterRowColumnLike(PLAYEDBY, baseB)
5   )
6 )
```

Die Filter unterstützen damit den gezielten Einsatz von Methoden der Metaklassen einer jeden Tabelle und helfen bei der variablen Gestaltung des Datenzugriffs.

Rule-Monitor

Der *Rule-Monitor* ist eine Komponente zur Überwachung der logischen Constraints aus Abschnitt 6.3. Logische Constraints werden im *Compiere Monitor* über Subklassen der abstrakten Klasse `Rule` modelliert. Diese bietet die nötigen Methoden zur Evaluierung eines Constraint, d.h. der Prüfung, ob die Bedingung des `Rule`-Objektes erfüllt wurde. Je nach Eigenschaft eines `Rule` findet eine Evaluierung innerhalb eines einzigen `Row` oder aber auch über ein gesamtes `Table`-Objekt statt. So benötigt z.B. ein Rule für die Einhaltung eines bestimmten Datenformats eines Spaltenwertes nur eine Evaluierung auf der Zeile des zu prüfenden Wertes. Prüfungen, wie das Verhindern doppelter Spalteneinträge, müssen hingegen über das gesamte `Table`-Objekt laufen, damit ein Erkennen aller Constraintverletzungen möglich ist.

Zusätzlich gibt es noch eine dritte Ebene - die der multi-dimensionalen Evaluierung. Hierbei wirkt ein `Rule` auf mehr als einer Tabelle. Dementsprechend können Änderungen an einem Wert innerhalb einer Tabelle auch zu Verletzungen in den Constraints anderer Tabellen führen. Für eine vollständige Evaluierung müssen alle diese Tabellen einer Prüfung hinsichtlich ihrer Constraints unterzogen werden.

Die Funktionsweise dieser Evaluierungen soll nun anhand der Abbildung 6.4 näher erläutert werden. In ihr sind einige konkrete `Rule`-Subtypen zu erkennen, die selbst Hierarchien bilden oder komplexerer Natur sind. Daneben gibt es noch weitere `Rule`-Objekte, welche aufgrund ihrer einfachen Implementierung zusammenfassend abgebildet sind als *<Rules>*.

Innerhalb eines `Row`-Objektes stellen Objekte des Typs `CMDataType` die Werte der jeweiligen Zeilenspalten dar. Ändert sich eines dieser Werte, so kann dies zu Änderungen an den Zuständen der Constraints hinsichtlich ihrer Erfüllung führen. Aus diesem Grund wird bei Änderung eines der Spaltenwerte eine erneute Evaluierung der Constraints vorgenommen. Da die Evaluierungen auf ganzen Tabellenzeilen arbeiten, besitzen Objekte des Typs `CMDataType` eine Referenz auf das `Row`-Objekt, welches sie beinhaltet. Diese erhalten die Instanzen erst bei Einbettung in das `Row`-Objekt, so dass eine Benutzung dieser Wrapperklassen auch ohne sie möglich ist. In diesem Fall besitzen sie keine Bindung an Tabellen und damit auch keine Constraints, welche sie erfüllen müssten.

Wird ein Wert geändert, so leitet die geänderte `CMDataType`-Instanz im *Rule-Monitor* eine Constraint-Revalidierung des `Row`-Objektes ein, welches es beinhaltet. Das Ergebnis

der erneuten Prüfung wird im Anschluss auf alle `CMDataType`-Objekte der überprüften Tabellenzeile übertragen, die innerhalb der revalidierten `Rules` Anwendung finden. Die beeinflussten Spalten werden dabei aus dem geprüften `Rules` selbst bezogen. Nur sie haben die Information darüber, welche Spalten durch sie geprüft werden.

Der *Rule-Monitor* dient so zur Überwachung des Zustandes bezüglich der `Rules` eines jeden `CMDataType`. Jedes dieser Spaltenwerte kann verschiedene logische Constraints in Form eines `Rule` verletzten. Um dem gerecht zu werden, hält der Rule-Monitor für jedes `CMDataType` eine Liste der verletzten `Rules` bereit. Konkret besteht diese Liste so aus Referenzen auf `Rules` der `Table`-Objekte, zu welchem die Zeile des `CMDataType` gehört. In dieser Liste werden nur verletzte `Rule`-Objekte gehalten. Im Umkehrschluss dazu werden `Rules` über die Gegenannahme als erfüllt eingestuft, wenn sie nicht in dieser Liste enthalten sind.

Zur Speicherung der Referenzen verwendet der Rule-Monitor ein `java.util.Hashtable`. Als Index dient hierbei das entsprechende `CMDataType`. Ist ein solches Objekt in dem `Hashtable` als Schlüssel vorhanden, so bedeutet dies, dass dieses Objekt derzeit eines oder mehrere `Rules` verletzt. Die Liste selber ist in Form eines `java.util.ArrayList` implementiert. Sie wird als Wert dem zugehörigen `CMDataType`-Schlüsselobjekt innerhalb des `Hashtable` zugewiesen. Ist diese Liste leer, d.h. tritt keinerlei Verletzung eines Constraint auf, so wird das Schlüsselobjekt, und damit auch die Werteliste selber, wieder aus der `Hashtable` entfernt. Das `Hashtable` enthält so stets nur die `CMDataTypes`, welche Constraintverletzungen aufweisen. Diese Vorgehensweise erspart das Anlegen eines `Hashtable`-Eintrags für jedes `CMDataType` und somit Rechenaufwand bei der Suche nach den verletzten `Ruleseines CMDataType`.

Die weitere Verwendung des beschriebenen Rule-Monitor wird im Verlaufe dieses Kapitels noch deutlicher zu erkennen sein.

Datenbank-Dialekte

In Kapitel 4.2.4 ist die Implementierung der Datenbankkommunikation von Compiere erläutert worden. Dabei wurden die verschiedenen Implementierungen der Schnittstelle `CompiereDatabase` vorgestellt. Sie dienen der Unterstützung mehrerer Datenbanken durch Compiere. Hierbei wird von einem SQL-Statement im Format einer Oracle Datenbank ausgegangen, welches anschließend durch die jeweilige Implementierung in das entsprechende Format der Zieldatenbank transformiert wird. Aufgrund einer anderen Herangehensweise beim Umgang mit Datenbankelementen ist dies jedoch ungenügend bei der Verwendung des *Compiere Monitor*. Um die zusätzlichen Methoden zu implementieren wurde eine Schnittstelle ähnlich dem `CompiereDatabase` geschaffen - `DBDialect`. Die implementierenden Klassen erlauben damit die Verwendung des *Compiere Monitor* auf verschiedenen Datenbanken. Aus zeitlichen Gründen wurde jedoch im Rahmen der Ausarbeitung nur eine Implementierung in Form der Klasse `DBDialectOracle`vorgenommen. Sie implementiert die nötigen Methoden für eine Verwendung auf Basis der Oracle Datenbanken.

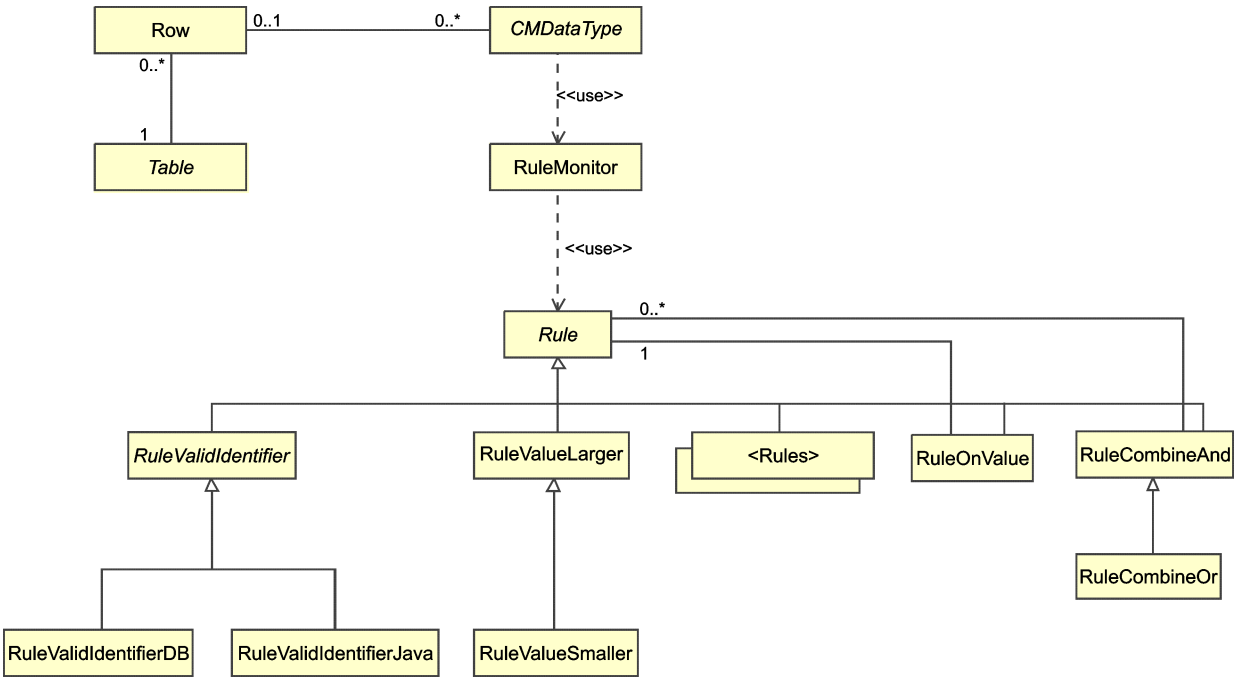

Abbildung 6.4: Klassendiagramm der Objekte des Rule-Monitor

Datenpräsentation

Es wurde in diesem Kapitel bereits eine Erläuterung der Gründe gegeben, die dazu führten, dass eine Entscheidung gegen die Verwendung eines Persistenzframeworks fiel. Eines davon war der generische Zugriff auf Datenbankobjekte. Dieser findet insbesondere bei der Implementierung grafischer Elemente zur Darstellung und Änderung der Spaltenwerte einer Tabellenzeile Anwendung. Sie sollen im Folgenden näher erläutert werden. Abbildung 6.5 dient dabei der Übersicht über die beteiligten Komponenten und deren Relationen zueinander.

Ziel bei der Entwicklung der grafischen Elemente war es, ein System sich selbst aktualisierender Elemente zu schaffen. Dies wurde durch eine Kombination verschiedener Techniken erreicht, von welchen einige erst im Anschluss in Abschnitt 6.7.2 Erwähnung finden werden. Das zentrale Element ist dabei die abstrakte Klasse `CMPanel`. Sie adaptiert die Klasse `javax.swing.JPanel` und dient als Container für weitere darstellende Komponenten. Diese werden durch ihre Subklassen definiert, wobei jedes der Darstellung der verschiedenen Ausprägungen der Klasse `CMDataType` dient. Eine detaillierte Beschreibung dieser `CMPanel`-Subklassen soll an dieser Stelle aber zu Gunsten einer allgemeinen Betrachtung ausbleiben.

Die Datentypen innerhalb einer Tabelle werden für ganze Spalten definiert. Aus diesem Grund wird ein `CMPanel`-Objekt nicht für einen einzigen Zeilenwert des Typs `CMDataType` erstellt, sondern für eine gesamte Tabellenspalte. Informationen über die aktuell zu nutzende Zeile sowie der Quelltabelle bezieht das `CMPanel`-Objekte aus Klassen, welche die

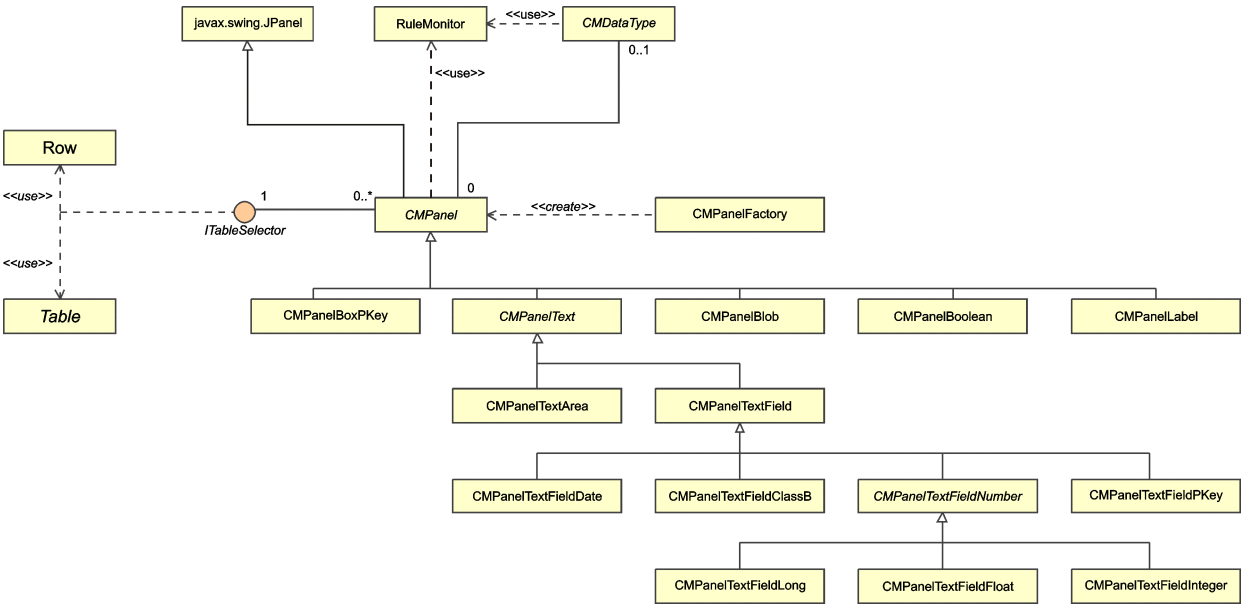

Abbildung 6.5: Komponenten der graphischen Darstellung von Tabellendaten

Schnittstelle `ITableSelector` implementieren. Zusammen mit der Information der anzuzeigenden Spalte, kann ein `CMPanel`-Objekt damit den aktuellen Zeilenwert in Form eines `CMDataType` ermitteln und anschließend gemäß seiner Implementierung anzeigen.

Die Objekte, welche `ITableSelector` implementieren entscheiden also über die aktuellen Zeilen und damit über die anzuzeigenden Daten. Sie sind zum Großteil selbst darstellende Elemente auf höherer Hierarchieebene. So z.B. ein Panel zur Darstellung aller Teamklassen, wobei das aktuelle `Row`-Objekt das der aktuell selektierten Teamklasse wäre und die Bezugstabelle die aller Teamklassen. Die Neuselektierung einer Teamklasse kann hier z.B. eine Aktualisierung aller enthaltenden `CMPanel`-Objekte auslösen. Dies bedeutet, dass jedes der `CMPanel`-Objekte das aktuelle `Row`-Objekt, was in diesem Fall die Zeile der aktuell selektierten Teamklasse wäre, von diesem Panel bezieht und anhand dieser Daten eine Aktualisierung seiner selbst durchführt. Der Aufruf für die Aktualisierung ist dabei parameterlos. Alle nötigen Daten, konkret sind das der Spaltenname sowie das `ITableSelector`-Objekt, werden dem `CMPanel` bereits bei dessen Initialisierung mit übergeben und bleiben für die gesamte Lebensdauer der Instanz konstant.

Zusätzlich zur Darstellung der Daten zeigt jedes `CMPanel` Informationen über verletzte logische Constraints des derzeit angezeigten `CMDataType` an. Je nach Implementierung des `CMPanel`, geschieht die Darstellung eines verletzten Constraint in Form von rot gefärbten Komponenten sowie mit einer zusätzlichen Ausgabe des verletzten `Rule` über den Tooltiptext der Komponenten. Zu diesem Zweck sendet es bei jeder Aktualisierung eine Anfrage bezüglich des Constraint-Status an den *Rule-Monitor*. Sie beinhaltet das derzeit angezeigte `CMDataType` des `CMPanel`. Mit diesem führt der *Rule-Monitor* eine Prüfung auf Vorhandensein eines verletzten Rule durch. Sie war Bestandteil des vorangegangenen Abschnitts.

Neben der Datenpräsentation des Zeilenwertes erlauben die meisten Subklassen, eine Ausnahme bildet z.B. die Klasse `CMPanelLabel`, auch das Ändern des angezeigten Wertes. Dabei werden die Änderungen sofort und direkt in das zugehörige `CMDataType`-Objekt des `CMPanel` geschrieben. Das hat Vor- wie auch Nachteile. Vorteilhaft ist so z.B. die Nutzung des *Rule-Monitor*. Ohne die direkte Änderung der Tabellendaten würden die aktuell editierten Daten nicht mit den tatsächlichen Daten innerhalb des `Row`-Objekts übereinstimmen. Eine Prüfung der Rules durch den *Rule-Monitor* würde somit falsche Resultate erzeugen. Nachteilig hingegen ist es bei der Implementierung von Komponenten, welche getätigte Änderungen an Zeilen rückgängig machen können. Diese müssen den Zustand vor Beginn der Änderung speichern um diesen im Anschluss wieder herstellen zu können. Das Gewicht der erreichten Vorteile überwiegt letztendlich jedoch das der wenigen Nachteile und macht diese somit irrelevant.

Die Technik der automatischen Aktualisierung der `CMPanel` und der Rules des `CMDataType` wird in Abschnitt 6.7.2 detailliert erläutert. Es vervollständigt die Gesamtsicht auf die Implementierung der beschriebenen `CMPanels`.

6.7.2 Design Patterns

Die im vorherigen Abschnitt erläuterten Implementierungen mittels allgemeiner Konzepte, beschrieben die Funktionalität der Kernkomponenten des *Compiere Monitor*. Aus strukturellen Gründen wurden dort Elemente ausgespart, welche zur Implementierung der Design Pattern gehören. Sie werden in diesem Abschnitt behandelt und ergänzen damit die Sicht auf die bereits genannten Implementierungen. Hierzu werden im Folgenden die verwendeten Design Patterns aufgezählt und jeweils mit einer Lokalisierung innerhalb der Implementierung vervollständigt. Eine umfassende Beschreibung der Design Pattern selbst wird an dieser Stelle jedoch nicht vorgenommen. Sie ist unter anderem [BMR+96] zu entnehmen.

Singleton

Das Singleton-Pattern wird bei Klassen genutzt, die während der gesamten Laufzeit nur einmalig instanziiert werden dürfen. Dazu werden alle Konstruktoren mit der Sichtbarkeit `private` ausgestattet und eine statische Referenz auf die einzig existente Instanz innerhalb dieser Klasse gehalten. Sie bleibt nach erstmaliger Instanziierung durch die statische Getter-Methode konstant und wird durch sie fortan zurückgegeben. Dieses Pattern erleichtert damit die globale Verwendung einmalig vorkommender Objekte. Die wichtigsten *Compiere Monitor*-Klassen, welche mittels des Singleton-Pattern implementiert wurden, sollen hier nun genannt werden. Aufgrund des einfachen Aufbaus dieses Pattern wird dabei jedoch auf eine erweiternde Darstellung in Form eines Klassendiagramms verzichtet.

- Die Klasse `Config` dient der Verwaltung der lokalen Einstellungen des *Compiere Monitor*, wie z.B. der Farbverwaltung oder der Lebensdauer eines Cache-Eintrags. Da es ausreicht diese Einstellungen nur einmalig aus der lokalen Datei auszulesen,

existiert von ihr global nur eine Instanz, deren Referenz mit der entsprechenden statischen Getter-Methode einzuholen ist.

- Tabellenobjekte dienen der Abbildung einer Tabelle aus der Datenbank. Sie laden die Tabellendaten zum Zeitpunkt ihrer Erstellung und speichern sie in Form von `Row`-Objekten. Das gesamte System des *Compiere Monitor* baut auf diesen Daten auf. Damit jede Komponente auf demselben Stamm von Daten arbeitet muss sichergestellt sein, dass jeweils nur ein Tabellenobjekt pro Tabelle existiert. Dies wird durch die Implementierung des Singleton-Patterns in jeder der Tabellenklassen erreicht.

- In Abschnitt 6.5 ist bereits die Technik des lokalen Cache im Kontext der Klassenbrowser vorgestellt worden. Diese wurde im Compiere Monitor mittels des Singleton-Patterns implementiert. Dazu gibt es die Klasse `LocalCache`, welche bei erstmaliger Verwendung der Singleton-Getter-Methode instanziiert wird und dann während der gesamten Laufzeit der Applikation bestehen bleibt. Während dieser Instanziierung lädt sie die Objekte aus dem lokalen Cache und hält sie innerhalb einer Liste vor. Wird diese Liste verändert, so wird der Cache zu gegebener Zeit lokal gespeichert. Das Singleton-Pattern trägt hier dazu bei unnötige Instanziierungen und Leseprozesse zu verhindern

Model-View-Controller

Das Pattern des *Model-View-Controller (MVC)* hat als Ziel, die Assoziationen zwischen den Komponenten der Daten-, Kontroll- und Präsentationsschicht aufzulockern. Gleichzeitig werden die Funktionen, wie das Aktualisieren aller Komponenten der Präsentationsschicht bei Änderungen der dargestellten Daten, gestärkt. Dieses Konzept wurde auch im *Compiere Monitor* implementiert. Abbildung 6.6 zeigt die Einbettung wichtiger Komponenten innerhalb das MVC-Pattern und dient als Grundlage der folgenden Betrachtungen.

Das Hauptaugenmerk liegt nun zuerst auf der Betrachtung des Controller `ObservableTable`. Er dient der Aktualisierung von Elementen der Präsentationsschicht bei Änderungen von Elementen der Modellebene. Die Elemente der Präsentationsschicht können dabei unterschiedlicher Ausprägung sein. Zu diesem Zweck wurde hierfür im *Compiere Monitor* die Schnittstelle `IObservable` eingeführt. Sie beinhaltet eine Methode zur Aktualisierung der implementierenden Klasse und stellt für den Controller `ObservableTable` alle Elemente der Präsentationsschicht dar. In der Abbildung wurde zum besseren Verständniss das, bereits vorgestellte, `CMPanel` als implementierende Klasse eingefügt. Dessen Subtypen implementieren alle diese Methode je nach deren eigener Ausprägung. Die Nutzung dieser Art der Aktualisierung erfordert von den Objekten eine vorherige Registrierung am entsprechenden Controller. Konkret bedeutet dies am Beispiel der `CMPanel`-Objekte, dass sich diese innerhalb ihrer Konstruktoren durch den Aufruf entsprechender Methoden am Controller `ObservableTable` registrieren.

Der Controller selbst bietet während der Registrierung bzw. späteren Aktualisierung eine Unterscheidung auf 3 Ebenen, die den dargestellten Modellelementen aus Abbildung

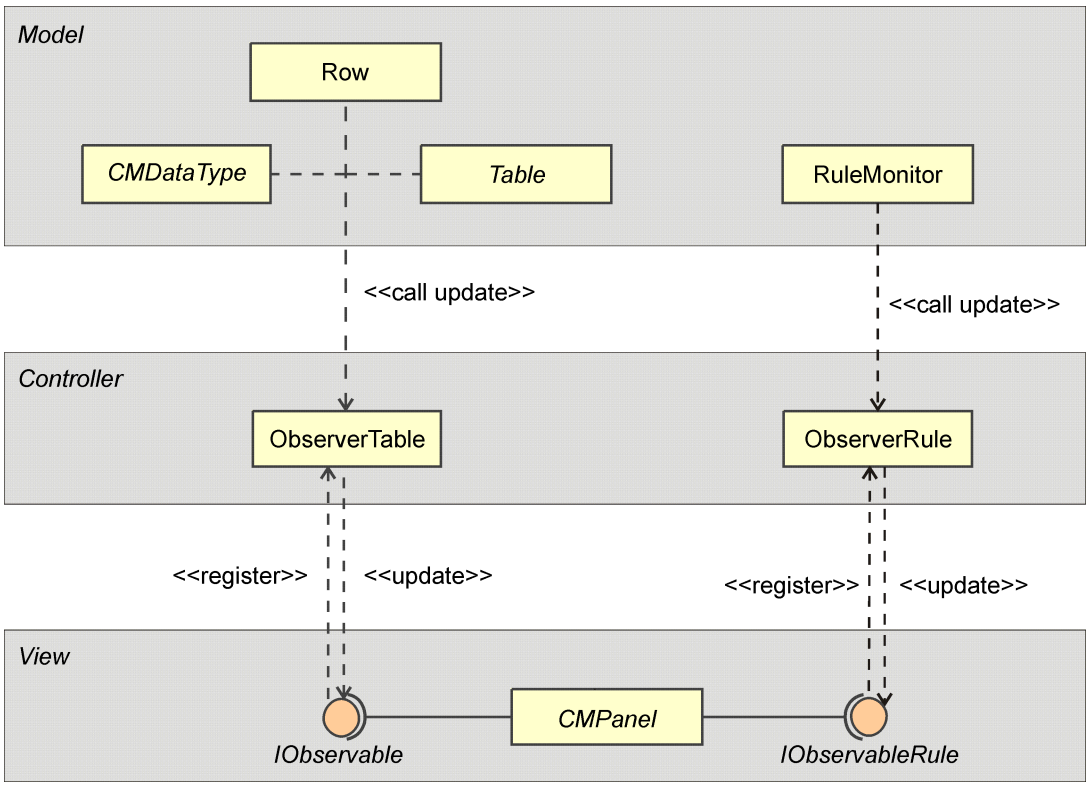

Abbildung 6.6: Anwendung des Model-View-Controller-Pattern im Compiere Monitor

6.6 entsprechen:

- `1. Ebene: Spaltenwerte`

  Wie schon in diesem Kapitel Erwähnung fand, sind Objekte des Typs `CMDataType` typischerweise innerhalb eines `Row`-Objektes anzutreffen, da diese dort die Rolle eines Spaltenwertes übernehmen. Werden Änderungen an diesem Objekt getätigt, so spiegelt dies eine Änderung an einem Spaltenwert einer Zeile wieder. Elemente der Präsentationsschicht, die zur Darstellung solcher Spaltenwerte dienen, wie alle Subtypen von `CMPanel`, müssen bei einer solchen Datenänderung aktualisiert werden. Zu diesem Zweck werden sie am Controller unter Angabe der Tabelle und Spalte registriert. Sie werden nun bei Änderungen eines Spaltenwertes durch den Controller aktualisiert. Jedes `CMDataType` einer Zeile sendet dafür bei dessen Datenänderung eine Aktualisierungsaufforderung an den Controller `ObserverTable` mit den Informationen über seine eigene Quelltabelle sowie dem Namen der Spalte, welcher er angehört. Der Controller ermittelt anschließend aus den registrierten Objekten diejenigen, welche sich an dieser Tabelle und Spalte registriert haben und aktualisiert sie über die entsprechende Methode der Schnittstelle `IObservable`.

- `2. Ebene: Zeilen`

  Es gibt Elemente der Präsentationsschicht, die sensibel auf Änderungen sämtlicher Spaltenwerte einer Zeile reagieren, d.h. über die begrenzende Sicht auf einen einzel-

nen Spaltenwert der 1.Ebene hinausgehen. Diesen wurde mit der zweiten möglichen Controller-Ebene entsprochen. Empfängt der Controller eine Aktualisierungsaufforderung eines Spaltenwertes, d.h. aus der 1. Ebene, so aktualisiert er ebenfalls alle Elemente des Typs `IObservable`, welche an eine Zeile gebunden sind.

Neben diesen Aktualisierungen aus 1.Ebene umfasst die 2.Ebene darüber hinaus noch weitere Aktualisierungsaufrufe, die z.B. bei dem Löschen einer Zeile getätigt werden. Die Löschaktion ändert durch ihren Bezug auf eine gesamte Zeile keine einzelnen Spaltenwerte und führt daher zu keinem Aufruf auf 1.Ebene. Änderungen diesen Typs können mit einer Registrierung eines `IObservable`-Objektes auf Ebene eines gesamten `Row` abgedeckt werden. Für eine Registrierung an dem Controller ist hierbei nur die Angabe der Bezugstabelle nötig. Jedwede Änderung an einer der Tabellenzeilen führt dementsprechend zu einer Aktualisierung auf dieser 2.Ebene.

- `3. Ebene: Tabellen`

  Die oberste aller Ebenen bildet die der Tabellen. Anders als Ebene 2 erweitert sie keine der beiden Unterebenen. Sie bildet vielmehr einen geschlossenen Fokus auf Ereignisse, welche die Tabelle betreffen, ohne Änderungen an Spalten- oder Zeilenobjekten mit einzuschließen. Änderungen, wie das Löschen einer Zeile werden durch ihren Bezug auf eine schon existierende Zeile auf dieser Ebene nicht beachtet. Anders verhält es sich bei dem Hinzufügen neuer Zeilen oder dem Löschen der gesamten Tabelle. Sie werden beide als Änderungen an der Tabelle selbst aufgefasst und führen damit zu einem Aktualisierungsaufruf dritter Ebene am Controller. Für eine Registrierung an diesem ist auch hier lediglich die Angabe der Tabelle nötig.

Die dargelegte Abstufung auf Modellebene erlaubt eine differenzierte Nutzung des Controller `ObservableTable`. Je nach Anforderungen auf Seiten der Präsentationsschicht können so am Controller Registrierungen auf unterschiedlichen Ebenen vorgenommen werden.

Eine weitere Verwendung findet das MVC-Pattern im Kontext der *logischen Constraints* und des *Rule-Monitor*. Wie in Abschnitt 6.7.1 ersichtlich wurde, können durch eine Änderung eines Spaltenwertes mehr als ein `Rule` verletzt werden. Neben der Aktualisierung der Elemente der Präsentationsschicht, die diesen Spaltenwert darstellen, müssen so nun auch die Elemente aktualisiert werden, deren `Rules` durch die Änderung verletzt wurden. Implementiert wurde diese Funktion mit Hilfe des Controller `ObserverRules`. Er erlaubt das Aktualisieren der Kontextinformationen, d.h. der Elemente, welche die Verletzung eines `Rule` anzeigen, von Objekten über die Verwendung der Schnittstelle `IObserveableRule`. Zu diesem Zweck registrieren sich die entsprechenden Objekte unter Angabe der für sie geltenden `Rules` an diesem Controller. Ändert sich innerhalb des *Rule-Monitor* der Status eines `Rule`, führt dieser eine Kontextaktualisierung für den sich ändernden `Rule` am Controller aus. Dieser führt daraufhin an allen `IObserveableRule`-Objekten, die sich für diesen `Rule` am Controller registriert haben die entsprechende Methode zur Aktualisierung ihrer Kontextdaten aus.

Beide genannten Implementierungen finden eine weitere Darstellung in Form des Sequenzdiagramms in Abbildung 6.7. Es zeigt einen vereinfachten Informationsfluss bei einer Änderung an einem `CMPanelBoolean`-Objekt. Dieses `CMPanel` besitzt zur Datenänderung zwei Elemente des Typs `javax.swing.JRadioButton`, wobei die Methode `itemStateChanged(e)` eine Selektionsänderung signalisiert. Im Sequenzdiagramm wird dabei vereinfachend nur eine Änderung auf den Wert `true` betrachtet. Nach vollzogener Selektionsänderung führt das `CMPanel` die entsprechende Datenänderung auf dem, durch ihn dargstellten `CMDataType` durch. Dieses sendet anschließend eine Aktualisierungsanforderung an die Controller `RuleMonitor` sowie `ObserverTable`, die dann die schon beschriebenen Aktivitäten ausführen.

Command

Das Command-Pattern dient der Kapselung von auszuführenden Code, welche hier als Kommandos bezeichnet werden. Zusätzlich lockert es die Kopplung zwischen dem ausführenden Objekt und dem auslösendem Objekt. Es behandelt dazu Kommandos als eigene Objekte, die vom dem Auslöser zur Ausführung an eine weitere Komponente übergeben werden.

Dieses Konzept findet ebenfalls im *Compiere Monitor* Anwendung, wenn auch quantitativ nur schwach ausgeprägt. Sie wird genutzt, um Kommandos nebenläufig laufen zu lassen, d.h. innerhalb eines `Threads`. Es wird insbesondere durch Komponenten der grafischen Benutzeroberfläche in Anspruch genommen. Dort dient er der Trennung umfangreicher Kommandos von dem *AWT-Thread* zur grafischen Darstellung und verhindert so dessen Blockierung während der Ausführung dieses Kommandos. Er ist damit funktional vergleichbar mit dem *Java-SwingWorker*, bietet jedoch darüber hinaus individuelle Funktionalitäten für den *Compiere Monitor*.

Das Starten eines nebenläufigen Kommandos wird über die Klasse `CommandRunner` vorgenommen. Dieses implementiert die Schnittstelle `java.lang.Runnable` und erlaubt damit die Ausführung seiner selbst innerhalb eines `java.lang.Thread`. Es bietet des Weiteren statische Methoden zum Start eines Kommandos. Ein solches Kommando wird durch ein Objekt repräsentiert, das die Schnittstelle `ICommand` implementiert. Diese besitzt eine Methode, dessen Implementierung den Inhalt eines Kommando definiert. Die Kommandos selber werden in den Klassen des *Compiere Monitor* größtenteils in Form von inneren oder anonymen Klassen implementiert. Während des Starts eines Kommandos mittels der genannten statischen Methoden erzeugt `CommandRunner` Instanzen seiner selbst, wobei ihm dabei das `ICommand`-Objekt übergeben wird. Mit dieser Instanz wird jeweils ein neuer `Thread` erzeugt und gestartet. Eine Blockierung des aufrufenden Objekts wird somit verhindert. Abbildung 6.8 stellt die genannten Komponenten noch einmal grafisch dar.

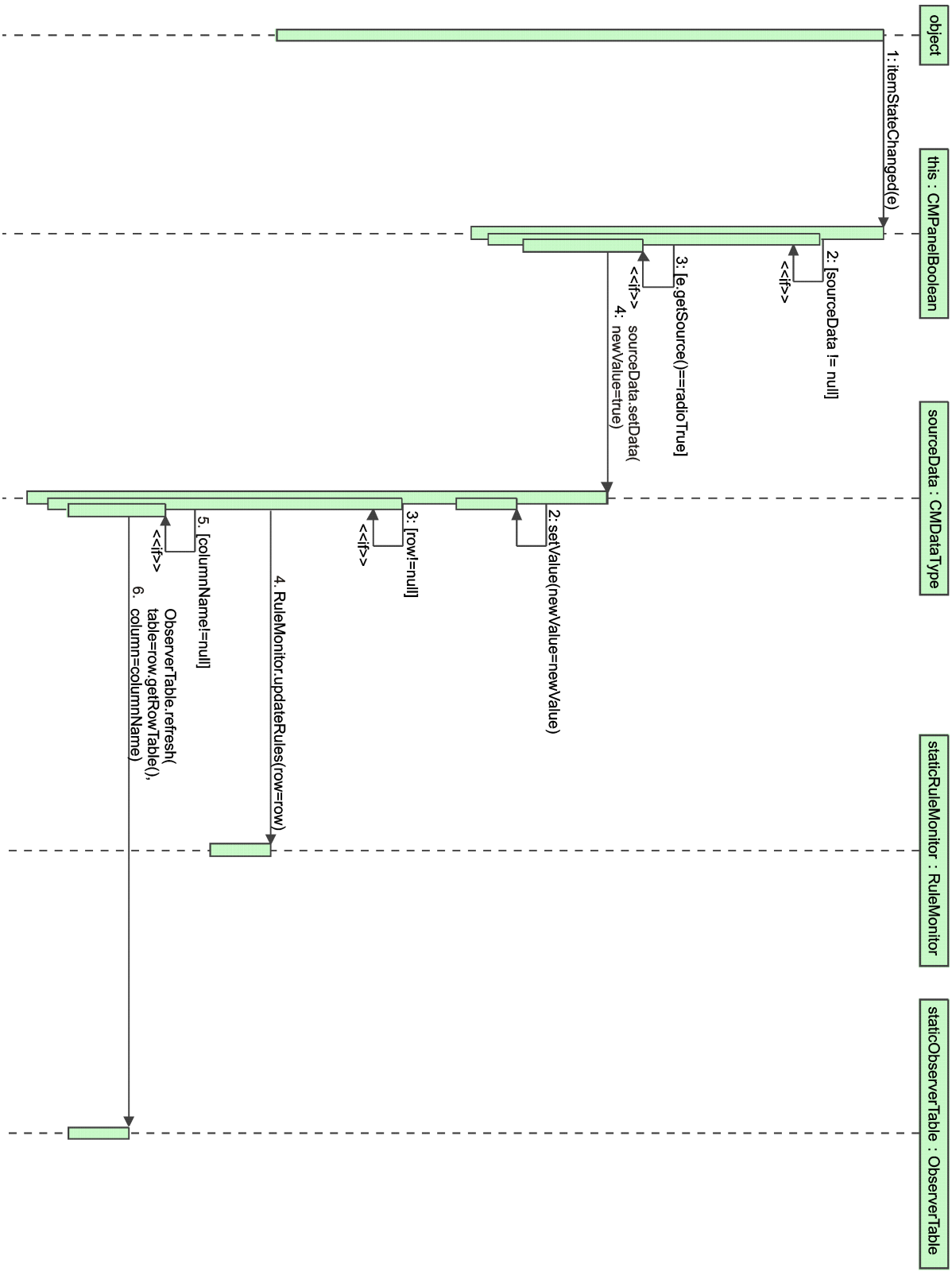

Abbildung 6.7: Kommunikationsstruktur bei Datenänderung eines CMDataType durch ein CMPanelBoolean

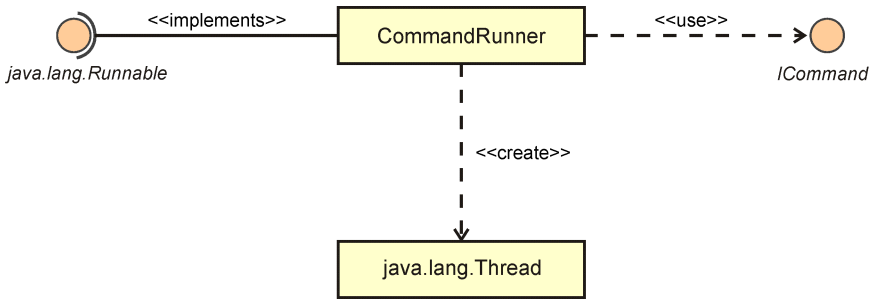

Abbildung 6.8: Implementierung des Command-Pattern im Compiere Monitor

Observer

Das Observer-Pattern, das schon in vergleichbarer Form als *Controller* innerhalb des MVC-Pattern genutzt wurde, erlaubt die Registrierung eines Objektes an einem zweiten Objekt, dem *Observer*. Treten Änderungen innerhalb des *Observer*-Objektes auf, so werden alle an ihm registrierten Objekte aktualisiert.

Im *Compiere Monitor* wird dieses Pattern nur an wenigen Stellen genutzt, da sich durch die starke Anlehnung an Datenmodelle das MVC-Pattern in diesem Fall besser eignet. Es findet z.B. innerhalb der Klasse `CMTable` Anwendung, welches in Abbildung 6.9 zu sehen ist und im Folgenden näher betrachtet wird. Es stellt ein `Table`-Objekt mit seinen `Row`-Objekten grafisch innerhalb eines `javax.swing.JTable` dar. Selektionsänderungen an der Tabelle erfordern hier möglicherweise die Aktualisierung von zusätzlichen Komponenten. Zu diesem Zweck bietet die Klasse `CMTable` Observerfunktionalitäten in Form von Methoden zur Registrierung weiterer, `ITableSelectorSetter` implementierender Klassen. Diese Schnittstelle erlaubt über eine Methode die Neuselektion eines `Row`-Objektes. Sie wird bei der Auswahl einer neuen Zeile innerhalb des `javax.swing.JTable` aus der Klasse `CMTable` aufgerufen, um alle registrierten Objekten über diese Neuselektierung zu informieren.

Konkret wird in Abbildung 6.9 die `ITableSelectorSetter`-implementierende Klasse `DisplayTableRead` bzw. dessen Subklasse `DisplayTableReadWrite` als registrierendes Objekt dargestellt. Beides sind Klassen zur detaillierten visuellen Darstellung einer konkreten Tabellenzeile. Zusätzlich erlauben sie das Umschalten auf eine Tabellensicht, welche mittels `CMTable` implementiert ist. Damit Änderungen der selektierten Zeile innerhalb des `CMTable` ebenfalls auf die Detailsicht übernommen werden, wird `DisplayTableRead` bzw. `DisplayTableReadWrite` mittels der Observermethoden an `CMTable` registriert. Fortan werden Selektionsänderungen innerhalb der Tabellensicht mittels der Schnittstellenmethoden an die Detailsichten übertragen und so eine Übereinstimmung der angezeigten Daten erreicht ohne eine Kopplung der einzelnen Komponenten zu benötigen.

Factory

In Abschnitt 6.7.1 wurde bereits das Thema Datenpräsentation behandelt. Abbildung 6.5 diente hierbei als grafisches Modell der erläuterten Komponenten und zeigte zusätz-

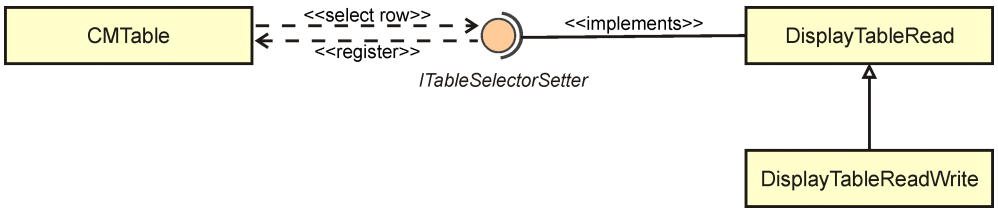

Abbildung 6.9: Observer-Pattern innerhalb des Compiere Monitor

lich die Klasse `CMPanelFactory`. Sie stellt eine Implementierung des Factory Pattern dar und ist das einzige seiner Art im *Compiere Monitor*. Es dient der Erstellung einer passenden `CMPanel`-Subklasse anhand eines Spaltennamens und `ITableSelector`-Objekts. Mit diesen wird das `Column`-Objekt der Spalte ermittelt. Je nach Typ und Ausprägung des `CMDataType` der Spalte wird ein passendes `CMPanel`-Objekt zur Betrachtung und Editierung der Spaltenwerte erstellt. Als Ausprägung werden hierbei spezifische Eigenschaften eines `CMDataType`-Objektes bezeichnet. So können z.B. Werte des Type `CMString` eine maximale Zeichenlänge besitzen. Sie wird innerhalb der Factory-Klasse verwendet, um konkret zwischen `CMPanelTextField` und `CMPanelTextArea` zu entscheiden. Beide bieten unterschiedliche grafische Komponenten, wobei sich letztere besser für längere Texte eignen. Ausgehend von der maximalen Zeichenlänge des `CMString`-Objekts entscheidet `CMPanelFactory` hier über das zu erstellende `CMPanel`.

Das Factory Pattern zentralisiert die Erstellung passender `CMPanel`-Objekte und unterstützt damit eine gezielte und effiziente Anpassung der `CMPanel`-Verwendung. Daneben vereinfacht sie die Erstellung eines Editors für einen Spaltenwert, da dem Entwickler die Entscheidungsfindung des passenden `CMPanel`-Objektes abgenommen wird.

6.8 Zusammenfassung

Die Implementierung des *Compiere Monitor* war eines der zentralen Zielsetzungen dieser Arbeit und mit einem letztendlichen Umfang von über 80.000 Code- und Kommentarzeilen darüber hinaus auch der aufwendigste Teil. Er ist als Erweiterung zum ERP-System Compiere (Siehe 4) entwickelt worden mit dem Ziel, dessen Wandlungsfähigkeit zu erhöhen. Das dabei erzielte Ergebnis wird Thema des Abschnitts 7.1 sein.

In diesem Kapitel wurden eine Reihe von Details zur Implementierung des *Compiere Monitor* gegeben. Sie alle dienen dem Erreichen der Zielsetzungen aus Abschnitt 6.2 und weiterer Funktionalitäten, von denen die Wichtigsten in Abschnitt 6.3 erwähnt worden sind. An dieser Stelle sollen nun, im Sinne eines Ausblicks auf mögliche Weiterentwicklungen, einige der offen gebliebenen Komponenten genannt werden:

- Compiere nutzt selbst eine rollenbasierte Rechteverwaltung. Je nach angemeldetem Benutzer können unterschiedliche Aktivitäten ausgeführt werden. Der *Compiere Monitor* implementiert dagegen bisher keinerlei Sicherheitsmechanismen. Eine weiterführende Implementierung in diesem Sinne würde eine Stärkung der Anwendbarkeit in der Praxis beitragen.

- Eine wichtige Funktionalität des *Compiere Monitor* ist die Verwaltung der Team- und Rollenklassen. Dazu gehört ebenso das Verändern von Rollenmethoden, speziell des Methodencodes. Der *Compiere Monitor* bietet hier derzeit nur ein einfaches *Syntax-Highlighting*, d.h. eine farbliche Markierung sprachspezifischer Schlüsselwörter. Der aus Entwicklungsumgebungen bekannte Umfang der Benutzerunterstützungen ist hier nicht zu finden. Lediglich die Methodenauswahl voll-qualifizierter Klassennamen wurde bereits implementiert. Dieser Punkt bietet aufgrund seiner umfangreichen Natur besonders viel Potential für weiterführende Entwicklungen.

- Typischerweise erlauben Entwicklungsumgebungen den Import fremder Klassen innerhalb einer Klasse. Dies ist im *Compiere Monitor* nicht möglich. Daraus resultiert, dass sämtliche Klassenangaben den voll-qualifizierten Klassennamen verwenden müssen, um eine Auflösung der Klasse zu gewährleisten. Die Integration einer Import-Möglichkeit würde hier den Entwicklungskomfort steigern.

- Die Menge unterstützter Datenbankmanagementsysteme ist im Compiere Monitor auf *Oracle Database* begrenzt. Um den Umfang der praktischen Anwendbarkeit zu erhöhen, können hier weiterführende Implementierungen für andere Systeme vorgenommen werden.

- Im Kontext des *Im- und Export* von Abschnitt 6.3 wurde bereits das Fehlen eines Sekundärschlüssels erwähnt. Die Implementierung der Unterstützung eines solchen würde das Problem des Imports lösen und darüber hinaus sicher auch zukünftigen Anforderungen entgegenkommen.

- Im Entwicklungsprozess des Compiere Monitor ist die Testphase aus zeitlichen Gründen nur minimal ausgefallen. Eine mögliche Weiterentwicklung kann und sollte hier einen Ausbau der wenigen JUnit-Testklassen forcieren.

Trotz all dieser offenen Punkte umfasst der Compiere Monitor in vollem Umfang die angestrebten Funktionalitäten. Er demonstriert den praktischen Nutzen der aspektorientierten Programmierung im Kontext der Wandlungsfähigkeit von ERP-Systemen. Das nun folgende Kapitel beinhaltet eine abschließende Bewertung der gesamten Arbeit und geht dabei auf die erzielten Vor- und Nachteile ein. Es trägt somit auch zur Gesamtbeurteilung des *Compiere Monitor* bei.

7 Ergebnis und Zusammenfassung

Das Ziel dieser Arbeit liegt in der Prüfung, inwieweit das Paradigma der Aspektorientierung eine Steigerung der Wandlungsfähigkeit von ERP-Systemen bewirken kann. Zu diesem Zweck führten die Kapitel 2-4 in den Bereich der ERP-Systeme generell ein und gaben einen konkreten Einblick in die Implementierung des verwendeten ERP-Systems *Compiere*. Darüber hinaus wurden in Kapitel 3 die Methoden zur Evaluierung der Wandlungsfähigkeit gegeben. Sie werden in diesem Kapitel Anwendung finden, um den letztendlich erreichten Grad der Verbesserung der Wandlungsfähigkeit durch die Verwendung der Aspektorientierung zu ermitteln. Die dabei genutzten Techniken wurden in den Kapiteln 5-6 vorgestellt. Sie stellen den *Compiere Monitor* im Detail dar und geben dabei gleichzeitig einen Einblick in die, durch *ObjectTeams/Java* implementierten, Aspektfunktionalitäten. Sie sind in dieser Arbeit das Instrument zur Stärkung der Wandlungsfähigkeit von *Compiere*. Die Frage, ob eine Steigerung eintrat und in welchem Grad die Aspektorientierung in Form von *ObjectTeams/Java* dabei gegenüber der Objektorientierung Vorteile bringt, ist Thema dieses abschließenden Kapitels.

Als erstes soll nun eine Evaluierung der Wandlungsfähigkeit von Compiere stattfinden.

7.1 Evaluierung der Wandlungsfähigkeit von Compiere

> „Als Wandlungsfähigkeit bezeichnet man die Fähigkeit, sich bzw. etwas zu verändern um, sich an Veränderungen anzupassen.“ (Siehe 3.1)

Diese Definition der Wandlungsfähigkeit konkretisiert sich im Umfeld von ERP-Systemen auf die Anforderung, dass sich das ERP-System idealerweise selbstständig anpassen können muss. Voraussetzung dafür ist jedoch das Wissen über benötigte Änderungen. Das Erkennen des Änderungsbedarfs sowie die darauffolgende Anpassung sind damit beides Funktionalitäten eines wandlungsfähigen Systems. Der *Compiere Monitor* unterstützt hier besonders das Erkennen des Änderungsbedarfs. Die genauen Einflüsse der Verwendung des *Compiere Monitor* auf die Wandlungsfähigkeit von Compiere werden im Folgenden nun mittels einer Berechnung des erreichten Grades der Vorher/Nachher-Wandlungsfähigkeit ermittelt. Zum Zwecke eines direkten Vergleichs folgt daraufhin eine Gegenüberstellung der Ergebnisse dieser Berechnung innerhalb des, in Abschnitt 3.2.4 vorgestellten Portfolios.

7.1.1 Ermittlung der Einflüsse des Compiere Monitor

Die Methodik zur Ermittlung der Wandlungsfähigkeit eines ERP-Systems war Thema des Kapitel 3.2.3. Sie gestaltet sich über die Beantwortung einer Reihe von Fragen verschiedener Kriterien, die im Anschluss mittels einer Berechnung zu einem diskreten Ergebniswert führen. Die Berechnung selbst wird dabei durch den verwendeten *Adaptability Analyzer* (Siehe 3.2.3) vorgenommen und ist daher an dieser Stelle irrelevant.

Die konkreten Auswirkung des *Compiere Monitor* auf die Wandlungsfähigkeit werden anhand der sich ändernden Antworten auf die einzelnen Fragen, sowie über die bereits erwähnte Gegenüberstellung im Ergebnisportfolio gezeigt. Da die Fragen selbst Bestandteil des *Adaptability Analyzer* sind, werden hierbei nur die Fragen für eine nähere Betrachtung herangezogen, deren Antworten durch die Aspekte des Compiere Monitor beeinflusst werden. Sie sind es, die zu der letztendlichen Veränderung der Bewertung der Wandlungsfähigkeit von Compiere führen werden. Diese Fragen werden nun näher betrachtet. Zusätzlich dazu wird zu jeder Frage die Antwort in Form ihrer Bewertung für ein Compiere mit (*Nachher*) und ohne (*Vorher*) Verwendung der Aspekte des *Compiere Monitor* mit aufgelistet. Auf eine Strukturierung, die über die oberste Aufteilung in *technische* und *geschäftsspezifische Wandlungsfähigkeit* hinausgeht und die Schichten bzw. Kriterien beinhaltet, wird hierbei aber verzichtet.

- *Technische Wandlungsfähigkeit*
  - ***Enthält das System Funktionen der Selbstorganisation, wie z.B. Selbstmanagement ?***

    Bewertung: Vorher=0, Nachher=2

    Compiere bietet selbst keine Möglichkeiten zur Überwachung des System- und Benutzerverhaltens. Daraus resultiert die mangelhafte Fähigkeit zum Management seiner selbst über das Erkennen von auftretendem Änderungsbedarfs. Der *Compiere Monitor* implementiert für Compiere hier nicht die direkte Fähigkeit zu einer Selbstorganisation. Er unterstützt jedoch das Erkennen des Änderungsbedarfs und schafft somit die Voraussetzung für eine weiterführende Implementierung im Sinne der Selbstorganisation. Aufgrund dieser vorbereitenden Unterstützung, erhält Compiere im Verbund mit dem *Compiere Monitor* hier 2 Punkte.
  - ***Erkennt das System Veränderungsbedarf bei Funktionen ?***

    Bewertung: Vorher=0, Nachher=2

    Der Fokus dieser Frage liegt auf den Funktionen der Applikationsschicht eines ERP-Systems (Siehe 2.4). Hierbei entsteht Veränderungsbedarf, wenn Funktionen keine Anwendung mehr finden oder sie falsche bzw. mangelhafte Ergebnisse liefern. Der *Compiere Monitor* kann hier durch das Logging der Benutzeraktivitäten dazu beitragen, nicht genutzte Funktionen zu identifizieren um so Veränderungsbedarf zu ermitteln.
  - ***Gibt es die Möglichkeit der Selbstdiagnose ?***

Bewertung: Vorher=0, Nachher=2

Auch hier wird durch den *Compiere Monitor* eine Steigerung in Form der Möglichkeit zur Überwachung verschiedenster Benutzeraktivitäten und Systemeigenschaften erreicht. Sie bilden die Grundlage, auf welcher eine spätere, durch den *Compiere Monitor* nicht implementierte Diagnose bzw. Auswertung stattfinden kann.

- ***Passt sich die Oberfläche an den Benutzer an ?***

  Bewertung: Vorher=0, Nachher=2

  Der *Compiere Monitor* ermöglicht eine Überwachung der individuellen Compiere-Einstellungen eines jeden Benutzers. Dies beinhaltet z.B. die zu verwendenden Farben und Schriftarten der Compiere-Clientanwendung. Sie bilden die Grundlage für eine automatische Anpassung der Benutzeroberfläche durch Compiere und führen zu einer gesteigerten Bewertung von 2 Punkten.

- *Geschäftsspezifische Wandlungsfähigkeit*

  - ***Können neue Segmentierungsmöglichkeiten hinzugefügt werden ?***

    Bewertung (Automatisierungsgrad / Wissen): Vorher=1/0, Nachher=3/3

    Als Segmentierungsmöglichkeiten werden allgemeine Kriterien bezeichnet, nach denen eine Bildung von Subsystemen, wie z.B. eine Aufteilung eines Lagers oder einer Filiale, erfolgen kann. Das abgetrennte Subsystem bildet hierbei selbst wieder ein autonomes System. Im Weiteren wird hier speziell auf eine Subsystembildung im Kontext von Tabellen eingegangen. Compiere besitzt die Möglichkeit der freien Definition von Tabellen, jedoch unter der Prämisse der Verwendung ein und desselben Datenbanksystems. Der *Compiere Monitor* schafft durch die Trennung von Compiere hier potentiell die Möglichkeit der Verwendung anderer Datenbanksysteme. Die Aspekte des *Compiere Monitor* können dabei gezielt Compiere-Methoden ersetzen und diese unter Verwendung einer abweichenden Datenbank implementieren. Theoretisch lassen sich so Tabellen auf andere Datenbanken auslagern. Allein durch diese Möglichkeit schafft hier die Verwendung der Aspektorientierung in Form des *Compiere Monitor* eine Verbesserung der Wandlungsfähigkeit.

  - ***Gibt es eine konfigurierbare Workflow-Engine ?***

    Bewertung (Anpassungsfähigkeit / Wissen): Vorher=1/0, Nachher=3/4

    In Kapitel 4 wurde bereits die Implementierung der Workflows in Compiere vorgestellt. Sie sind dabei eher als eine Empfehlungen auszuführender Aktivitäten anzusehen, d.h. sie definieren selbst keine vorgeschriebene Reihenfolge, an die der Benutzer gebunden ist. Der *Compiere Monitor* bietet hier nun die Möglichkeit, die Arbeitsschritte der Benutzer im einzelnen zu überwachen. Abweichungen von den empfohlenen Arbeitsschritten der einzelnen Workflows können somit erkannt werden. Er bietet dadurch die Möglichkeit, Workflows an häufig auftretende Abweichungen anzupassen und unterstützt so deren

praxisbezogene Gestaltung. Darüber hinaus werden durch die Aspekte des *Compiere Monitor* bereits abgearbeitete Workflowknoten, je nach Erfüllung ihrer Erfolgskriterien (Siehe 6.4.2) farblich markiert. Der Benutzer erhält damit einen Überblick über schon vollzogene Arbeitsschritte im Workflow.

Compiere bietet von sich aus keine dieser Funktionalitäten. Der *Compiere Monitor* schafft hier das Wissen über die Art und Weise der Verwendung eines jeden Workflows und ermöglicht damit erst die Anpassung der Workflows an das real auftretende Benutzerverhalten.

– ***Können die Workflows geändert bzw. konfiguriert werden ?***
Bewertung (Anpassungsfähigkeit / Wissen): Vorher=1/0, Nachher=3/4
Die Frage zur Konfigurierbarkeit der Workflows geht einher mit der vorherigen Frage nach einer konfigurierbaren Workflow-Engine. Die Workflows in Compiere sind an sich konfigurierbar. Jedoch entsprechen sie dabei nur Empfehlungen und richten sich selbst nicht an die tatsächlich auftretenden Benutzerschritte aus. Dies wird erst durch die Überwachung des *Compiere Monitor* ermöglicht. Die Konfiguration der Workflows wird somit erleichtert, da die tatsächlichen Benutzerschritte hier als Quelle von Verbesserungen dienen können.

Diese Fragen wirken sich auf das Gesamtergebnis ihrer jeweiligen Kriterien aus und damit auch auf die darüber liegenden Schichten. Eine detaillierte Auflistung aller erreichten Punktwerte dieser Ebenen wird hier aber nicht vorgenommen. Sie sind für das letztendlich erreichte Ergebnis nicht relevant. Eine Gegenüberstellung der sich ändernden Ergebnisse wird im Folgenden nun mit Hilfe eines Portfolios stattfinden.

7.1.2 Ergebnisportfolio

Der vorherige Abschnitt diente dem Überblick der Fragen, deren Antworten durch die Hinzunahme der Aspektorientierung in Form des *Compiere Monitor* eine Veränderung erfahren. Zusammen mit den nicht erwähnten Fragen, bilden sie die Grundlage zur Berechnung der Gesamtwandlungsfähigkeit von Compiere durch den *Adaptability Analyzer*. Die Betrachtung und Einordnung dieser Ergebnisse folgt nun.

Das Ergebnis der Bewertung der Gesamtwandlungsfähigkeit des ERP-System Compiere durch den *Adaptability Analyzer* lautet prozentual wie folgt:

- ***23%*** für Compiere
- ***34%*** für Compiere bei Verwendung der aspektorientierten Erweiterung in Form des Compiere Monitor (Im Folgenden auch abkürzend als *Compiere+* bezeichnet)

Anhand dieser Ergebnisse lässt sich bereits eine Steigerung der Wandlungsfähigkeit von Compiere durch die zusätzliche Verwendung des *Compiere Monitor* erkennen. Sie beträgt 11%.

Die Bedeutung dieses prozentualen Wertes lässt sich aus dem Portfolio aus Abschnitt 3.2.4 erschließen. Das Erreichen des Maximalwertes von 100 Prozent entspricht im Portfolio (Abb. 7.1) einer Erlangung einer maximalen Ausprägung sowohl der technischen- als auch geschäftsspezifischen Wandlungsfähigkeit und liegt somit in der äußersten Ecke des Quadranten *Ausgezeichnet*. Demgegenüber liegt der Quadrant *Durchschnittlich*, dessen unterste Ecke den Bereich einer Bewertung von 0 Prozent markiert. Dementsprechend bildet solch eine prozentuale Bewertung eine Kombination beider Ebenen der Wandlungsfähigkeit und bietet selber keine Möglichkeit eines Rückschlusses auf die einzelnen Werte der kombinierten Ebenen.

Diesem Zweck dient die zusätzliche Einordnung der erreichten Ergebnisse in das erwähnte Portfolio. Über die rein prozentuale Beurteilung hinaus, gewährt es Einblick in die Verteilung der erreichten Wandlungsfähigkeit auf die beiden Ebenen der technischen sowie geschäftsspezifischen Dimension. Abbildung 7.1 zeigt die erzielten Ergebnisse unter Verwendung solch eines Portfolios. Die erwähnte Steigerung der Wandlungsfähigkeit um 11% verteilt sich in diesem Portfolio auf die beiden Achsen der technischen bzw. geschäftsspezifischen Wandlungsfähigkeit. Dadurch wird eine Schwachstellenanalyse ermöglicht.

Im Fall der aktuellen Untersuchung ist zu erkennen, dass die Steigerung besonders auf der Ebene der technischen Wandlungsfähigkeit erreicht wird. Dies liegt daran, dass die Einwirkungen des *Compiere Monitor* lediglich der Überwachung dienen, selbst aber keinen direkten Einfluss auf geschäftsspezifische Compiere-Elemente, wie Prozesse ausüben. Der *Compiere Monitor* dient vielmehr als informationstechnische Grundlage weiterführender Implementierungen zur automatischen Anpassung von Compiere und steigert damit besonders dessen technische Wandlungsfähigkeit. Die Steigerung selbst gereicht noch nicht zu einer Verlagerung in einen der benachbarten Quadranten, lässt jedoch die Verbesserung zweifelsohne erkennen.

Eine abschließende Zusammenfassung über den Nutzen der Aspektorientierung für die Wandlungsfähigkeit von ERP-Systemen ist Thema des nächsten Abschnitts.

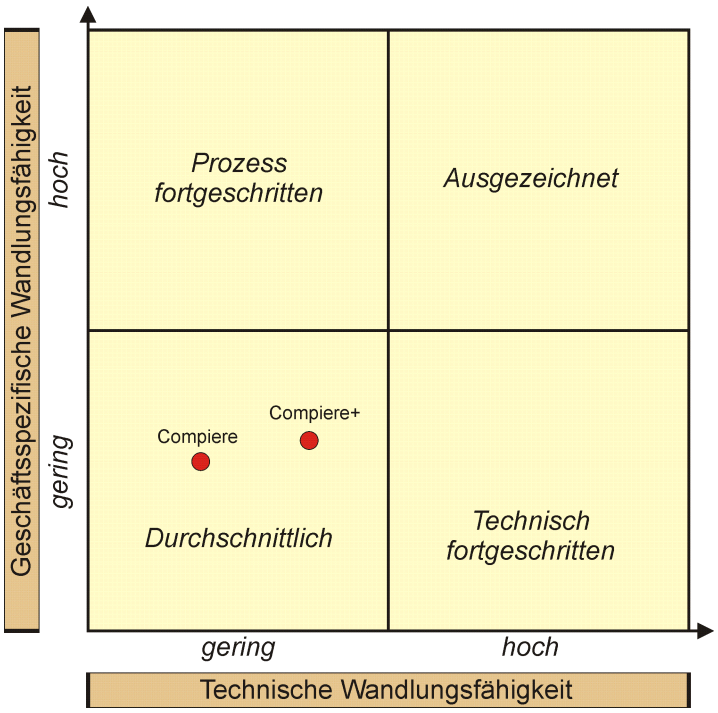

Abbildung 7.1: Ergebnisportfolio der Wandlungsfähigkeit von Compiere

7.2 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war eine Evaluierung der aspektorientierten Programmierung zur Steigerung der Wandlungsfähigkeit eine ERP-Systems. Am Beispiel der aspektorientierten Sprache *ObjectTeams/Java* und dem ERP-System *Compiere* wurden im Verlauf dieser Ausarbeitung geeignete Funktionalitäten ausgearbeitet, die zu einer Steigerung der Wandlungsfähigkeit des verwendeten ERP-Systems führten. Sie fanden ihre Implementierung in Form des *Compiere Monitor*. Neben der Verwaltung aller Aspekte in einer Datenbank, ermöglicht er darüber hinaus auch eine Konfiguration der einzelnen Aspekte sowie ein dynamisches Verhalten der Aspekte zu deren Laufzeit.

Die nun folgende Zusammenfassung wird strukturell in zwei Ebenen unterteilt. Als Erstes folgt eine funktionale Betrachtung, die im Anschluss durch eine technische Sichtweise erweitert wird.

In Kombination mit dem *Compiere Monitor* bietet dieses konfigurierbare sowie dynamische Aspektverhalten eine breite Palette von Anwendungsszenarien. Einige von ihnen wurden bereits während der Erläuterung zu den betrachteten Fragen zur Ermittlung der Wandlungsfähigkeit erwähnt (Siehe 7.1.1). Die gesammelten Daten des *Compiere Monitor* können so zum Beispiel zur Analyse des Benutzerverhaltens herangezogen werden. Mit ihrer Hilfe kann man anschließend häufig wiederkehrende Arbeitsschritte oder wenig genutzte Workflowelemente identifizieren, um diese daraufhin als Quelle möglicher Workflowänderungen zu nutzen. Es lassen sich auf diese Weise vorhandene Workflows

gezielt optimieren und verbessern.

Teil dieser Verbesserung kann auch eine vergleichende Herangehensweise an das aufgezeichnete Benutzerverhalten sein. So ist es z.B. einem erfahrenen Benutzer möglich, eine abweichende Arbeitsfolge gegenüber einem Workflow zu verwenden und damit zeitlich effizienter zu einem äquivalenten Ziel kommen. Mit Hilfe der Loggingfunktionalitäten des *Compiere Monitor* lassen sich solch optimierte Abweichungen vom bestehenden Workflow erkennen und entsprechend in den Workflow integrieren.

Weitere Anwendung kann der *Compiere Monitor* bei der Etablierung eines Firmenstandards finden. In Compiere kann jeder Benutzer unterschiedliche lokale Compiere-Einstellungen verwenden. Das Logging dieser einzelnen Benutzereinstellungen kann dazu beitragen, eine Konsensmenge der meistverwendeten Einstellungen zu bilden. Mit ihr ist es möglich, bei der Definition eines allgemeinen Standards, benutzerfreundliche Einstellungen zu finden. Die Verteilung dieses Standards kann im Anschluss daran mit Hilfe des *Compiere Monitor* und entsprechend entwickelter Aspekte vorgenommen werden.

Letztendlich decken jedoch die, durch den Compiere-Aspektgenerator (Siehe 6.4) erstellten Aspekte, nur eine begrenzte Menge der Anwendungsmöglichkeiten ab. Das Potential der Verwendung der Aspektorientierung zum Zwecke einer Steigerung der Wandlungsfähigkeit eines ERP-Systems wurde mit dieser Arbeit am Beispiel von *ObjectTeams/Java* und *Compiere* gezeigt, jedoch bei weitem nicht ausgeschöpft. Eine Implementierung weiterer Funktionalitäten mit Hilfe von *ObjectTeams/Java*, kann hier zweifelsohne zu einer weiteren Steigerung der Wandlungsfähigkeit führen.

Der *Compiere Monitor* bietet dafür eine mögliche Plattform. Er besitzt keinerlei Beschränkung auf die Aspekte der Aspektgeneratoren und erlaubt so eine leichte Integration weiterer Aspekte. Die Aspektgeneratoren sind hier vielmehr selbst als Zusatzkomponenten des eigentlichen Funktionskerns des *Compiere Monitor* zu sehen. Sie dienen lediglich einer gezielten Nutzung Compiere-spezifischer Aspekte mit der Unterstützung einer erleichterten Änderbarkeit mit Hilfe der Aspektkonfiguratoren aus Abschnitt 6.4.2.

Die bis hierhin erfolgte Zusammenfassung ist durch eine funktionelle Sicht auf die erreichten Ziele bestimmt. Als Kernthema hatte sie die Funktionen zur Stärkung der Wandlungsfähigkeit. Diese lassen sich jedoch auch über die Verwendung der herkömmlichen Objektorientierung implementieren. Aus diesem Grund folgt nun eine weitere Beurteilung unter dem Gesichtspunkt der technischen Realisierung.

Das Kapitel 3 stellte unter anderem die zunehmende Relevanz des Themas der Wandlungsfähigkeit im Kontext von ERP-Systemen dar. Zur Wahrung der Konkurrenzfähigkeit ihrer Produkte, müssen ERP-System-Anbieter in Zukunft verstärkt ihre Entwicklungsbestrebungen in dieser Richtung ausbauen. Im Fall von Compiere kann eine entsprechende Weiterentwicklung durch den Anbieter, d.h. der Compiere Inc., als auch durch die freie Entwicklergemeinde stattfinden. Bei einer Entwicklung mittels herkömmlicher objektorienterter Programmiersprachen durch die Gemeinde, wirkt sich dabei jedoch besonders die, in Abschnitt 2.5 genannte Eigenschaft der *Flexibilität* negativ aus. Änderungen durch die Gemeinde müssen nicht zwingend Bestandteil der zukünftigen Releases sein und gehen damit bei einem Wechsel auf neue Releases verloren. Sie bil-

den damit Produktindividualisierungen, die sich darüber hinaus auch negativ auf die Evolutionsfähigkeit gegenüber den offiziellen Releases auswirken.

Das Konzept der Aspektorientierung bietet hier die Möglichkeit der Programmierung neuer Funktionalitäten, ohne das Basisprodukt zu verändern. Der Basiscode kann somit durch neue Releases ersetzt werden, ohne den Verlust zusätzlich programmierter Funktionalitäten in Kauf nehmen zu müssen. So besitzt auch der *Compiere Monitor* mit seinen Aspektgeneratoren eine Unabhängigkeit gegenüber den verwendeten Compiere-Releases, solange diese, die in den Aspektbindungen verwendeten Methodensignaturen nicht umdefinieren. Dies ist einer der großen Vorteile der Aspektorientierung gegenüber der herkömmlichen Objektorientierung. Für den ERP-System-Anbieter bietet diese Technik ebenfalls vielerlei wirtschaftliches Potential. So wäre z.B. die Integration von Funktionalitäten in Form eines käuflichen Plugins möglich, das keine Anpassung des Basisprodukts erfordert und somit auch für ältere Releases angeboten werden könnte.

Der *Compiere Monitor* ist selbst ein gutes Beispiel für ein solches Plugin. Er ist ein eigenständiges Produkt, das Funktionalitäten zum Logging verschiedener Compiere-Aktivitäten besitzt ohne selbst Bestandteil des Compiere-Basisprodukts zu sein. Es ist möglich, die Funktionalitäten der dabei genutzten Aspekte ebenfalls mittels Objektorientierung zu implementieren. Dies bedingt dann jedoch die Änderung des Compiere-Basiscodes. Neben den bereits genannten Nachteilen durch eine solche Produktindividualisierung, würde eine Nachbildung der Aspekte mittels OOP verstärkt das Problem des *Scattering* und *Tangling* aus Abschnitt 5.1.1 aufwerfen. Die angestrebte Modularität würde damit geschwächt und zukünftige Weiterentwicklungen erschwert werden.

Die Verwendung der Konzepte von *ObjectTeams* brachte hier wesentliche Verbesserungen hervor. Sie halfen dabei, die benötigten Funktionalitäten zur Steigerung der Wandlungsfähigkeit von Compiere modular und flexibel zu gestalten, ohne Änderungen an dessen Basiscode zu erfordern. Im Sinne der Flexibilität fand dabei besonders das Konzept der *Guards* auf Team- und Bindungsebene Anwendung. Sie dienen dem De- und Aktivieren von Funktionalitäten zur Laufzeit von Compiere. Zur Gestaltung der Funktionalitäten selbst, kamen die Techniken fast aller `callin`- und `callout`-Bindungen zum Einsatz. Lediglich die `replace-callout`-Bindung fand keinerlei Verwendung. Dies liegt darin begründet, dass eine Zielsetzung dieser Arbeit in der bereits erwähnten Prämisse der Unveränderbarkeit des gesamten Compiere-Basiscodes bestand.

Die Kombination der genannten Techniken war Voraussetzung für die Implementierung der benötigten Funktionalitäten und demonstriert die reichhaltigen Möglichkeiten des *ObjectTeams*-Modells anhand eines praxisnahen Anwendungsfalls.

Die Entwicklung des *Compiere Monitor* war Teil einer aspektorientierten Implementierung von zusätzlichen Compiere-Funktionalitäten. Diese fanden bei der letztendlichen Evaluierung der aspektorientierten Programmierung zur Steigerung der Wandlungsfähigkeit Anwendung und resultierten in einem, als positiv zu bewertendem Ergebnis. Die Aspektorientierung erlaubt im Sinne der Wandlungsfähigkeit eine vorteilhafte Implementierung dieser Funktionalitäten und bietet das Potential für weitere Verbesserungen. Der

Compiere Monitor unterstützt dies durch seine Offenheit gegenüber neuen Aspekten. Er kann damit, über den Rahmen dieser Arbeit hinaus, als Grundstein weiterführender Entwicklungen mittels *ObjectTeams/Java* dienen. Die Zielsetzungen dieser Arbeit wurden somit erfüllt und durch die Vielseitigkeit des *Compiere Monitor* sogar übertroffen.

Für mich persönlich führte die Ausarbeitung dieser Diplomarbeit zu einem besseren Verständnis von ERP-Systemen im Allgemeinen sowie Compiere im Speziellen. So erforderte die Entwicklung geeigneter Aspekte zum Erreichen der gesetzten Ziele eine Vertiefung in die technische Implementierung des ERP-Systems Compiere. Die weitgehend theoretisch angesammelten Kenntnisse über ERP-Systeme und das Thema der Wandlungsfähigkeit wurden so durch die Erfahrungen der praktischen Anwendung ergänzt. Es ergibt sich damit ein umfassendes Gesamtbild über das Thema des *Enterprise Resource Planning*.

Das persönliche Interesse meinerseits fand seine Entsprechung hingegen besonders in der Behandlung der spannenden Thematik der aspektorientierten Programmierung. Sie wurde angewandt in Form von *ObjectTeams/Java* und bildete die Grundlage dieser Arbeit. Die Implementierung des *Compiere Monitor* beanspruchte darüber hinaus den Großteil des Bearbeitungszeitraumes und war von großem Interesse für mich. Besonders die Kombination datenbankbasierter Techniken mit den Konzepten von *ObjectTeams* bildete hier eine interessante Mischung.

Abschließend bleibt mir persönlich nur der Blick auf eine durchweg positiv verlaufene Ausarbeitung. Inwieweit jedoch das Thema Wandlungsfähigkeit letztendlich seinen Weg in die Praxis finden wird, bleibt abzuwarten. Aber nicht zuletzt zum Zwecke einer verbesserten Kundenzufriedenheit sind ERP-System-Anbieter dazu aufgerufen, ihre Produkte für den Kunden wandlungsfähiger zu gestalten. Mit dieser Arbeit wurde ein erster Schritt in diese Richtung für das ERP-System Compiere unternommen.

Literaturverzeichnis

[AG05] Andresen, Katja und Norbert Gronau: *An Approach to Increase Adaptability in ERP Systems.* In: *Managing Modern Organizations with Information Technology : Proceedings of the 2005 Information Recources Management Association International Conference*, 2005.

[AKS05] Andresen, Katja, Steffen Koslowski und Michael Stoffer: *Marktübersicht: ERP-Systeme – Software bis 5.000,- Euro.* ERP-Management, 3, 2005.

[ALG06] Andresen, Katja, Anne Lämmer und Norbert Gronau: *Vorgehensmodell zur Ermittlung der Wandlungsfähigkeit von ERP-Systemen.* In: *Multikonferenz Wirtschaftsinformatik 2006.* GITO-Verlag Berlin, 2006.

[Aspa] AspectJ: *Projektseite.* http://www.eclipse.org/aspectj.

[Aspb] AspectWerkz: *Projektseite.* http://aspectwerkz.codehaus.org.

[BA01] Bergmans, Lodewijk und Mehmet Aksit: *Composing Multiple Concerns Using Composition Filters*, 2001.

[BD04] Busch, Axel und Wilhelm Dangelmaier: *Integriertes Supply Chain Management : Theorie und Praxis effektiver unternehmensübergreifender Geschäftsprozesse.* Gabler, 2004.

[Bin02] Binder, Christof: *Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken.* Diplomarbeit, Technische Universität Berlin, 2002.

[BMR+96] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern-Oriented Software Architecture - A system of Patterns, Volume 1.* John Wiley & Sons, 1996.

[Bre03] Brendel, Michael: *CRM für den Mittelstand - Voraussetzungen und Ideen für die erfolgreiche Implementierung.* Betriebswirtschaftlicher Verlag Dr.Th. Gabler, 2003.

[Com] Compiere, Inc.: *Homepage.* http://www.compiere.org.

[CST03] Chiba, Shigeru, Yoshiki Sato und Michiaki Tatsubori: *Using HotSwap for Implementing Dynamic AOP Systems.* Technischer Bericht, Dept. of

Mathematical and Computing Sciences Tokyo Institute of Technology IBM Tokyo Research Laboratory, 2003.

[Dav00] Davenport, Thomas H.: *Mission Critical: realizing the promise of enterprise systems.* Harvard Business School Press, 2000.

[DJS04] Dalager, Christian, Simon Jorsal und Eske Sort: *Aspect Oriented Programming in JBoss 4.* Diplomarbeit, IT University of Copenhagen, 2004.

[Frü97] Frühwirth, Thom: *Constraint-Programmierung - Grundlagen und Anwendungen.* Springer, Berlin, 1997.

[GLA06] Gronau, Norbert, Anne Lämmer und Katja Andresen: *Entwicklung wandlungsfähiger Auftragsabwicklungssysteme.* In: *Wandlungsfähige ERP-Systeme - Entwicklung, Auswahl und Methoden*, Kapitel 3, Seiten 37–56. Gito-Verlag Berlin, 2006.

[Gro04] Gronau, Norbert: *Enterprise Resource Planning und Supply Chain Management.* Oldenburg Verlag München, 2004.

[Gro06] Gronau, Norber: *Wandlungsfähige Informationssystemarchitekturen - Nachhaltigkeit bei organisatorischem Wandel.* GITO-Verlag Berlin, 2006.

[GW05] Guliani, Gautam und Dan Woods: *Open Source for the Enterprise.* O'Reilly, 2005.

[Hac02] Hacker, Florian: *Aspektorientiertes Entwerfen mit "Aspectual Collaborations" Entwicklung eines grafischen Editors für die repository-basierte Entwicklungsumgebung PIROL.* Diplomarbeit, Technische Universität Berlin, 2002.

[Hat06] Hattwig, Jörg: *Open-Source-Software in der Praxis - Zarte Pflänzchen im Applikationsbereich.* IS Report, 6, 2006.

[Her02a] Herrmann, Stephan: *Composable Designs with UFA.* Technischer Bericht, Technische Universität Berlin, 2002.

[Her02b] Herrmann, Stephan: *Object Teams: Improving Modularity for Crosscutting Collaborations.* Technischer Bericht, Technische Universität Berlin, 2002.

[HS00] Homburg, Christian und F. Sieben: *Customer Relationship Management. Strategische Ausrichtung statt IT-getriebenem Aktivismus.* Management Know-how Papier M052 - Universität Mannheim, 2000.

[Hug03] Hugos, Michael: *Essentials of Supply Chain Management.* John Wiley & Sons, 2003.

[Hun03] Hundt, Christine: *Bytecode-Transformation zur Laufzeitunterstützung von Aspekt-Orientierter Modularisierung mit Object-Teams/Java.* Diplomarbeit, Technische Universität Berlin, 2003.

[JBo] JBoss, Inc.: *JBoss AOP Reference Documentation.* http://docs.jboss.com/aop/1.3/aspect-framework/reference/en/html/running.html.

[Jeh05] Jehle, Markus: *Gute Open-Soruce-Lösung für Mittelständler - Trotz Entwicklungssprung stößt das ERP-Paket Compiere an seine Grenzen.* Computerwoche, 41:24–25, 2005.

[KCA04] Kniesel, Günter, Pascal Costanza und Michael Austermann: *JMangler - A Powerful Back-End for Aspect-Oriented Programming.* In: Filman, R., T. Elrad, S. Clarke und M. Aksit (Herausgeber): *Aspect-oriented Software Development.* Prentice Hall, 2004.

[KLM+97] Kiczales, Gregor, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier und John Irwin: *Aspect-Oriented Programming.* Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997.

[Ks96] Kristensen, B. B. und K. Østerbye: *Roles: Conceptual Abstraction Theory and Practical Language Issues.* Theory and Practice of Object Sytems, 2:143–160, 1996.

[Lad03] Laddad, Ramnivas: *AspectJ in Action - Practical Aspect-Oriented Programming.* Manning Publications, 2003.

[Mos03] Mosconi, Marco: *Modularisierung und Adaptierung von Komponenteninteraktionen mit Object Teams und dem CORBA Komponentenmodell.* Diplomarbeit, Technische Universität Berlin, 2003.

[Moz00] Mozilla: *Mozilla Public Licence.* http://www.mozilla.org/MPL/MPL-1.1.html, 2000.

[NA05] Nicoara, Angela und Gustavo Alonso: *Dynamic AOP with PROSE.* http://www.mics.org/Loewenberg05/22Sep/T43_DynamicAOP.pdf, 2005.

[Obj] ObjectTeams: *ObjectTeams/Java Language Definition.* http://www.objectteams.org/def/0.8/index.html.

[OT00] Ossher, Harold und Peri Tarr: *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* Technischer Bericht, IBM T.J. Watson Research Center, 2000.

[Pfe03] Pfeiffer, Carsten: *Anpassbarkeit komplexer Software-Module durch ObjectTeams/C++ am Beispiel von graphischen Benutzungsschnittstellen.* Diplomarbeit, Technische Universität Berlin, 2003.

[RAJ02] Roman, Ed, Scott Ambler und Tyler Jewell: *Mastering Enterprise JavaBeans.* John Wiley & Sons, 2002.

[RB04] Roger, Alexander T. und James M. Bieman: *Towards the Systematic Testing of Aspect-Oriented Programs.* Technischer Bericht, Colorado State University, 2004.

[Rup05] Rupp, Heiko W.: *JBoss - Server-Handbuch für J2EE-Entwickler und Administratoren.* Dpunkt Verlag, 2005.

[SGSP02] Spinczyk, Olaf, Andreas Gal und Wolfgang Schröder-Preikschat: *Aspect C++ : An Aspect-Oriented Extension to C++.* Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), 2002.

[SRJ99] Scholz-Reiter, Bernd und Jens Jakobza: *Supply Chain Management - Überblick und Konzeption.* HMD - Praxis der Wirtschaftsinformatik, 207:7–15, 1999.

[Sum05] Sumner, Mary: *Enterprise Resource Planning.* Pearson Prentice Hall, 2005.

[SV00] Saaty, Thomas L. und Luis G. Vargas: *Models, Methods, Concepts & Applications of the Analytic Hierarchy Process.* Springer US, 2000.

[tusT] TSE 'technologieberatung und systementwicklung': *SAP - Marktanteile Deutschland 1999.* http://www.tse-hamburg.de/Papers/SAP/SAPMarktanteile.html.

[Vei02] Veit, Matthias: *Evaluierung modularer Softwareentwicklung mit Öbject Teamsäm Beispiel eines Projektmanagementsystems.* Diplomarbeit, Technische Universität Berlin, 2002.

[Wil06] Wildemann, Horst: *Supply Chain Management - Leitfaden für unternehmensübergreifendes Wertschöpfungsmanagement.* TCW, 2006.

[WK01] Wallace, Thomas F. und Michael H. Kremzar: *ERP: Making It Happen - The Implementers' Guide to Success with Enterprise Resource Planning.* John Wiley & Sons, 2001.

[ZRZ04] Zhou, Yuewei, Debra Richardson und Hadar Ziv: *Towards a Practical Approach to Test Aspect-Oriented Software.* Technischer Bericht, University of California, 2004.