Objektorientierung und Persistenz:

Konzepte und Realisierung eines Materialraums unter Anbindung paradigmatisch unterschiedlicher Datenbanken

Ulrich Brammer und Markus Göhmann

Diplomarbeit
Universität Hamburg
Fachbereich Informatik

Betreuung:
Dr. Ingrid Wetzel
Dr. Daniel Moldt

August 1997

Diese Diplomarbeit wurde dem Fachbereich Informatik der Universität Hamburg zur teilweisen Erfüllung der Anforderungen zur Erlangung des Titels Diplom-Informatiker eingereicht.

Abgrenzung der Beiträge

Bei der Ausarbeitung dieser Diplomarbeit ist darauf hinzuweisen, daß bei der Erstellung des Inhalts im wesentlichen von einer gemeinschaftlichen Arbeitsleistung auszugehen ist.

In einzelnen Kapiteln bzw. Abschnitten liegt eine deutlich überwiegende Arbeitsleistung eines einzelnen Autoren vor, was aus der folgenden Aufstellung ersichtlich ist. In diesem Zusammenhang muß jedoch berücksichtigt werden, daß auch hier eine nicht unerhebliche Mitarbeit des anderen Autoren vorliegt.

Abschnitt	*Autor*
1.1 bis 1.4	beide Autoren
2.1 bis 2.4	Ulrich Brammer
3.1 bis 3.5	Markus Göhmann
4.1 bis 4.3	beide Autoren Niederschrift: Markus Göhmann
5.1 bis 5.3	beide Autoren
5.4 bis 5.6	beide Autoren Niederschrift: Ulrich Brammer
6.1 bis 6.6	beide Autoren
7.1 bis 7.2	beide Autoren

Erklärung

Wir versichern hiermit, diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den 12. August 1997

Ulrich Brammer
Schusterkoppel 4
22399 Hamburg
Matr.-Nr. 4526100

Markus Göhmann
Siebentunnelweg 71
25469 Halstenbek
Matr.-Nr. 4526280

Betreuung

Dr. Ingrid Wetzel (Erstbetreuerin)
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Vogt-Kölln-Straße 30
22527 Hamburg

Dr. Daniel Moldt (Zweitbetreuer)
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Theoretische Grundlagen der Informatik
Vogt-Kölln-Straße 30
22527 Hamburg

Für unsere Eltern

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Hintergrund

Der Arbeitsbereich Softwaretechnik an der Universität Hamburg, Fachbereich Informatik befaßt sich mit dem objektorientierten Entwurf und der objektorientierten Konstruktion rechnergestützter Werkzeuge, die zu einem Informationssystem integriert werden. Der evolutionäre und partizipative Softwareentwicklungsprozeß orientiert sich dabei am Arbeitsplatz für qualifizierte menschliche Tätigkeit und nutzt im Rahmen dieses Leitbilds die Entwurfsmetapher ,,Umgang mit Werkzeug und Material" (Methode WAM). Die Intention dieser Metapher ist die Vereinigung der fachlichen und der technischen Betrachtung eines Informationssystems in einem einheitlichen objektorientierten Modell. Dadurch wird es möglich, die Begriffe und Konzepte der fachlichen Welt ohne semantischen Bruch in ein Programmiermodell und damit in den technischen Kontext zu überführen.

Informationssysteme kommen nicht nur innerhalb des klassischen Informationsmanagements (Kreditanstalten, Versicherungen usw.), sondern zunehmend auch in anderen Branchen wie zum Beispiel der industriellen Fertigung (CIM) oder dem Gesundheitswesen zum Einsatz. Da im Zuge einer innerbetrieblichen und zwischenbetrieblichen Integration von Informationssystemen neben dem Datenaustausch zwischen den Systemen (Datenmobilität) insbesondere die Möglichkeit der dauerhaften Ablage der Informationen (Datenpersistenz) erforderlich ist, bedarf es zusätzlicher Konzepte zur Unterstützung persistenter Materialobjekte innerhalb verteilter Arbeitsumgebungen.

Die Arbeiten [Wulf, Weske 94] und [Gryczan 95] haben sich als erste dieser Thematik aus der Sicht der Methode WAM angenommen und ein Konzept (Materialverwaltung) zur Verwaltung und Koordination persistenter Materialobjekte erarbeitet. Die Materialverwaltung enthält Mechanismen, deren Konformität zur Methode WAM fraglich ist. Zum einen verlieren Materialobjekte während der per-

sistenten Ablage zeitweise ihren Materialcharakter und fungieren als Werkzeuge. Zum anderen stellen Materialobjekte Wissen über ihre technische Repräsentation zur Verfügung, was dem Grundsatz der fachlichen Interpretierbarkeit ihrer Schnittstelle widerspricht.

Darüber hinaus besitzen die dargestellten Ansätze zwei wesentliche Defizite, die ihre allgemeine Einsatzfähigkeit stark einschränken. Erstens berücksichtigen sie nicht die Anbindung unterschiedlicher Datenbanksysteme, auch nicht die eines Paradigmas. Zweitens ermöglicht keiner der Ansätze die partielle Handhabung vernetzter Objektstrukturen. So muß beim Zugriff auf ein einziges Materialobjekt auch auf alle referenzierten Materialobjekte zugegriffen werden. Dies bedeutet im Extremum, daß alle Materialobjekte des Informationssystems in den transienten Speicher geholt werden müssen und somit für andere rechnergestützte Werkzeuge nicht mehr zugreifbar sind.

1.2 Aufgabenstellung

Die Aufgabe dieser Diplomarbeit besteht darin, einen Beitrag zur rein fachlich getriebenen Entwicklung von Informationssystemen zu leisten. Zu diesem Zweck soll ein Framework konzipiert und prototypisch realisiert werden, das von den technischen Konzepten abstrahiert und eine Schnittstelle bereitstellt, die in einem beliebigen fachlichen Kontext fachlich interpretierbar ist. Dazu ist eine Abstraktionsebene zu finden und zu beschreiben, die die technisch transparente Behandlung der technischen Komponenten durch die fachliche Welt ermöglicht und aus der fachlichen Perspektive betrachtet intuitiv als unterer Abschluß der Anwendungswelt verstanden werden kann. Im Vordergrund steht dabei die persistente Ablage von Materialobjekten in paradigmatisch unterschiedlichen persistenten Speichern, wobei insbesondere die partielle Handhabung vernetzter Material- und Objektstrukturen zu unterstützen ist.

1.3 Übersicht

In Kapitel 2 wird der für diese Arbeit grundlegende Begriff des Informationssystems gefaßt und anhand eines Beispiels die fachliche und die technische Sicht eines Informationssystems beschrieben. Nachdem in diesem Kapitel die Rolle des Objekts als Informationseinheit beleuchtet worden ist, werden die Entwicklungsziele benannt und ein grundlegender Architekturvorschlag zur Erreichung dieser Ziele dargestellt. Darauf aufbauend kann abschließend die gewählte Zielsetzung dieser Arbeit konkretisiert werden.

Kapitel 3 gibt zunächst einen Überblick über die objektorientierte Anwendungsentwicklung nach der Methode WAM. Darauf aufbauend erfolgt die Darstellung des Begriffs der Arbeitsumgebung und des für diese Arbeit relevanten Konzepts der Materialverwaltung. Abschließend werden die vorgestellten Begriffe und Konzepte auf ihre fachlich generelle Verwendbarkeit untersucht.

In Kapitel 4 wird das fachliche Basiskonzept des Materialraums erarbeitet. Dazu wird zunächst der dem Konzept des Materialraums zugrundeliegende Materialbegriff hinsichtlich bislang nicht betrachteter Konzepte (zum Beispiel Identität, Komposition, Lebensdauer von Materialien) erweitert. Anschließend wird das Konzept der fachlichen Umgangsarten zur Synchronisation nebenläufiger Handlungen entwickelt. Darauf aufbauend erfolgt abschließend die Darstellung des Materialraums sowie dessen Einordnung bzgl. fachlich spezifischer Konzepte (zum Beispiel Arbeitsumgebung).

Im Mittelpunkt des Kapitels 5 steht die technisch transparente Anbindung paradigmatisch unterschiedlicher Datenbanksysteme an die objektorientierte Programmiersprache C++. Dazu wird eine gemeinsame Schnittstelle für relationale und objektorientierte Datenbanksysteme festgelegt und das Konzept der persistenten Programmiersprache vorgestellt. Darauf aufbauend entwickeln wir das technische Konzept des (objektorientierten) Persistent Distributed Shared Memory (PDSM). Der PDSM abstrahiert von der horizontalen und vertikalen Speichertrennung und stellt damit sowohl die natürliche Erweiterung des Konzepts des virtuellen Speichers persistenter Programmiersprachen als auch ein adäquates Konzept zur (programmier-)technischen Modellierung des fachlich generellen Materialraums dar.

Gegenstand des Kapitels 6 ist das Framework *dinx*, das die in Kapitel 4 beschriebenen fachlich generellen Konzepte und damit insbesondere den Materialraum basierend auf dem in Kapitel 5 beschriebenen Konzept des Persistent Distributed Shared Memory realisiert. Dazu wird neben der Anbindung eines persistenten Speichers die (programmier-)technische Umsetzung fachlich genereller Konzepte (zum Beispiel Materialidentität, Materialverknüpfung und Materialkomposition) beschrieben. Das zentrale Modellierungsmittel stellt dabei die Materialraumreferenz dar, die den bestehenden C++-Referenzmechanismus substituiert und einen technisch transparenten Umgang mit paradigmatisch unterschiedlichen Datenbanken ermöglicht.

In Kapitel 7 werden abschließend die Beiträge der vorliegenden Diplomarbeit zusammengefaßt und kritisch gewürdigt. Darauf aufbauend geben wir einen Ausblick auf zukünftige Entwicklungen und weiterführende Arbeiten.

In Anhang A wird eine umfassende Übersicht über die Eigenschaften von Objekten sowie über die unterschiedlichen Beziehungen zwischen Objekten, Klassen und Typen dargestellt.

Der Anhang B enthält den Quellcode (Borland C++ 4.5) der prototypischen Realisierung des Frameworks *dinx*. Tiefergehende Implementationsdetails können diesem Anhang entnommen werden.

1.4 Schreibweisen

Grafische Konventionen

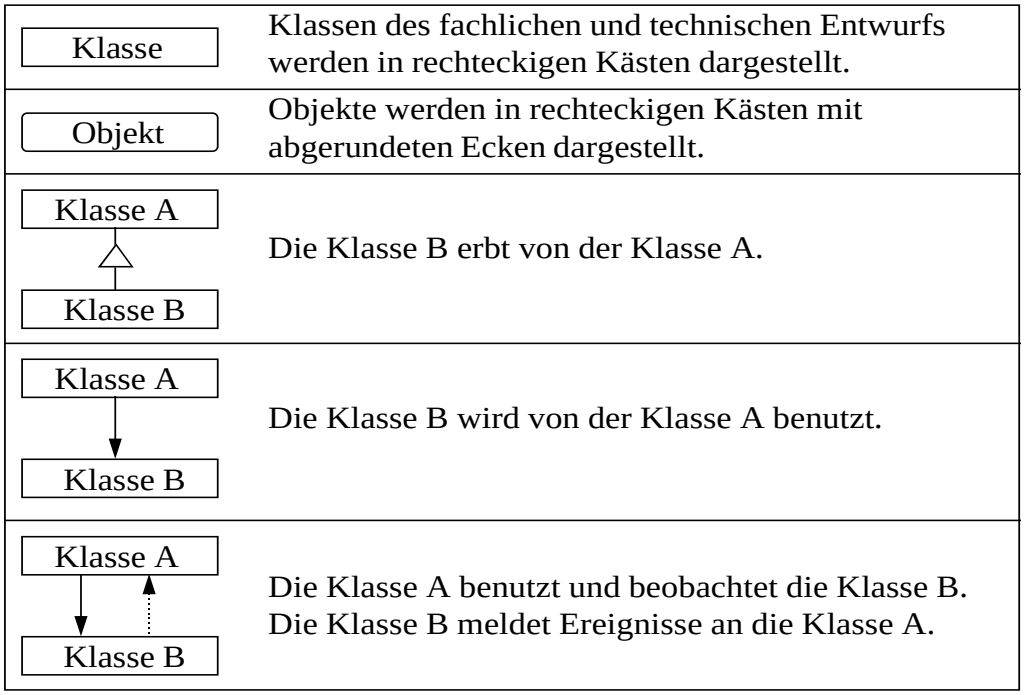


Abbildung 1.1: Konventionen für Klassendiagramme

Textuelle Konventionen

- *Geschlechtsspezifische Schreibweisen*:
  In dieser Diplomarbeit wollen wir auf eine strikte geschlechtslose Schreibweise verzichten, um dadurch eine bessere Lesbarkeit des Textes zu erreichen. Wir benutzen zum Beispiel die Bezeichnung ,,der Programmierer" als Synonym sowohl für die weibliche Programmiererin als auch den männlichen Programmierer. Die Leser(innen) dieser Arbeit mögen uns diese Entscheidung nachsehen und selbst entscheiden, welche Schreibweise sie bevorzugen.

- *Fremdsprachliche Ausdrücke*:
  Wir haben uns bemüht, fremdsprachliche Ausdrücke ins Deutsche zu übertragen. Teilweise mußten wir allerdings darauf verzichten, weil es sich um Namen bestehender Systeme oder Komponenten, etablierte Fachbegriffe oder programmiersprachliche Ausdrücke handelt.

- *Textart und Textstil*:
  In der Regel werden wichtige Begriffe bis zu ihrer Einführung *kursiv* geschrieben und erscheinen anschließend in normaler Schrift. Allerdings stellen wir auch einige fremdsprachliche Schlüsselbegriffe wegen der besseren Lesbarkeit dauerhaft *kursiv* dar.

Kapitel 2

Das Informationssystem

Im Fokus dieses Kapitels steht das Informationssystem. Anhand eines CIM-Systems werden die unterschiedlichen fachlichen und technischen Sichten eines Informationssystems erörtert. Nach der Betrachtung des Objekts als Informationseinheit werden die aufeinander aufbauenden Komponenten eines Informationssystems und ihre Aufgaben dargestellt. Anschließend werden die Entwicklungsziele benannt und ein grundlegender Architekturvorschlag zur Erreichung dieser Ziele dargestellt. Darauf aufbauend wird abschließend die Aufgabenstellung dieser Arbeit konkretisiert.

2.1 Einordnung des Begriffs

Im Zuge einer zunehmenden elektronischen Verarbeitung von Informationen und der dabei fortschreitenden Integration von Anwendungssystemen in Unternehmen, öffentlichen Einrichtungen etc. zeichnet sich ein deutlicher Trend zu Informationssystemen ab, in denen Objekte nicht nur dauerhaft abgelegt, sondern sowohl innerhalb von Anwendungssystemen als auch über deren Grenzen hinaus ausgetauscht werden ([Mathiske 96b]). Dies gilt insbesondere für *Nicht-Standard-Anwendungen*, wie zum Beispiel[1]:

- *Computer Integrated Manufacturing (CIM) Systeme*:
  In einem CIM-System werden Produkte auf der einen Seite entworfen, entwickelt, produziert und qualitätsgeprüft und auf der anderen Seite angeboten, bestellt, in die Produktion eingeplant und schließlich versandt. Im Vordergrund steht das Produkt, das sich jeder Anwendung des CIM-Systems in einer unterschiedlichen Weise präsentiert ([Mertens et al. 92]).

- *Bürokommunikationssysteme*:
  Bürokommunikationssysteme dienen zur Unterstützung der Kooperation

[1] Weitere Beispiele sind u.a.: Computer-Aided Software Engineering, Expertensysteme, wissenschaftliche und medizinische Anwendungen

qualifizierter Anwender auf der Basis von Geschäftsprozessen (*workflows*). Diese Geschäftsvorgänge können sich über unterschiedliche Abteilungen erstrecken, wobei die in diesen Systemen modellierten Objekte zwischen Rechnern ausgetauscht werden ([Kilberth et al. 93]).

- *Computer Aided Publishing Systeme*:
  Computer Aided Publishing Systeme dienen u.a. der Verwaltung und Verarbeitung multimedialer Daten (Text, Bild, Sprache, Musik, Video und Daten). Diese Daten müssen aus Massenarchiven gewonnen (*information retrieval and discovery*), zusammengestellt und urheberrechtlich geschützt werden. Die Heterogenität der genutzten Datentypen zeichnet diese Systeme besonders aus ([Silberschatz et al. 96]).

Während autonome Anwendungen durch den lokalen Umgang mit einfach strukturierten und kurzlebigen Daten geprägt sind, zeichnen sich heutige Informationssysteme besonders durch einen hohen Bedarf an *Persistenz* (Lebensdauer) und *Mobilität* (Beweglichkeit, Migrationsfähigkeit) komplexer und semantisch reichhaltig strukturierter Objekte aus ([Mathiske 96b]). Die Betrachtung einzelner, isolierter Anwendungen ist damit vor allem im Hinblick auf zukünfige Entwicklungen[2] nicht mehr ausreichend. Objekte sind zur Wahrung eines konsistenten Zustands des Gesamtsystems ohne Redundanzen zu halten und müssen beliebig zwischen Anwendungen eines Informationssystems, möglichst auch zwischen Informationssystemen, ausgetauscht werden können. Ein Ziel bei der Entwicklung von Informationssystemen muß es deshalb sein, dem bestehenden Bedarf an Lebensdauer und Beweglichkeit von Objekten in ausreichendem Maße gerecht zu werden.

2.1.1 Motivierendes Beispiel

In Anlehnung an [Mertens et al. 92] soll am Beispiel eines CIM-Systems die typische Struktur und das Zusammenspiel einzelner Anwendungen in einem Informationssystem aufgezeigt werden. Computer Integrated Manufacturing dient in Unternehmen zur Integration der betriebswirtschaftlichen und der technischen Informationsverarbeitung sowie der physischen Produktionsvorgänge und unterstützt damit den qualifizierten Anwender bei der Auftragsabwicklung (*Produktionsplanung und Steuerung*, *PPS*) und der Produktausreifung (*C-Anwendungen*). Das CIM-System setzt sich u.a. aus den nachfolgend genannten Anwendungen zusammen, die gemeinsam auf Objekte zugreifen oder untereinander austauschen und damit im direkten oder indirekten Informationsaustausch zueinander stehen (vgl. auch Abbildung 2.1).

[2] vgl. [Silberschatz et al. 96]

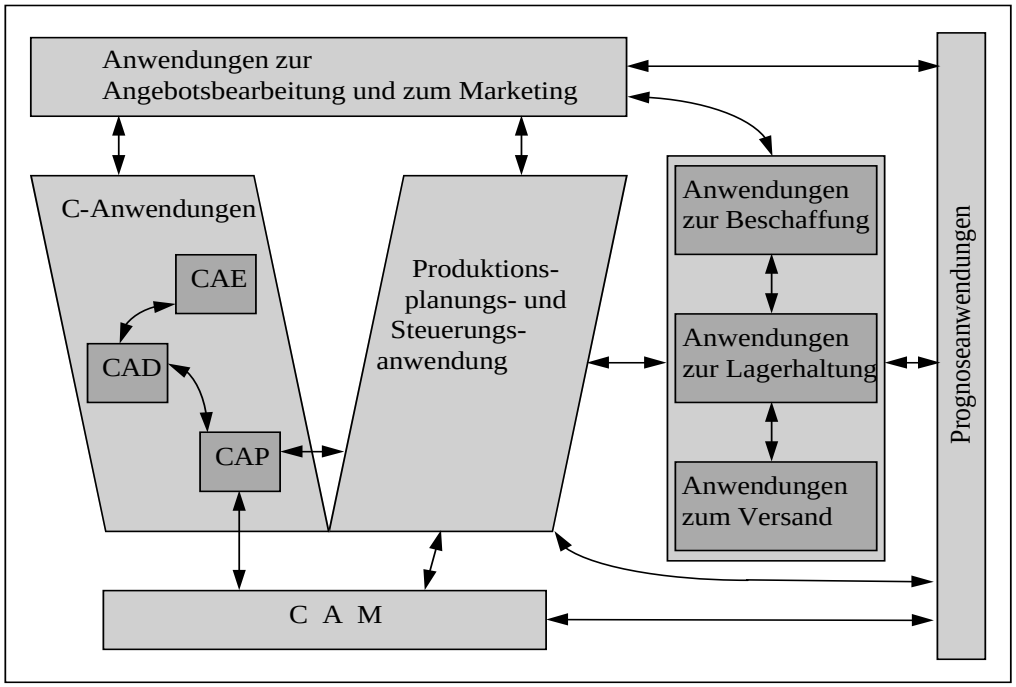


Abbildung 2.1: Ausschnitt des Zusammenspiels der CIM-Anwendungen

1. *Anwendungen zur Angebotsbearbeitung und zum Marketing*
   Anwendungen zur Unterstützung der Angebotsbearbeitung und des Marketings werden unter den Begriffen *Computer Assisted Sales* und *Computer Assisted Selling* (*CAS*) zusammengefaßt. CAS-Anwendungen dienen zur Unterstützung der Kundenanfrage- und Angebotsbearbeitung sowie zur Erfassung und Prüfung von Aufträgen und der Verwaltung von Kundendaten. So können Produkt- und Angebotsinformationen bereitgestellt, Kundenanfragen zielsicher beantwortet und Angebotsaktionen (*Mailings*) durchgeführt werden.

2. *Produktionsplanungs- und Steuerungsanwendung*
   Die PPS-Anwendung bezieht von den CAS-Anwendungen Informationen über die zu produzierenden Stückzahlen der einzelnen Produkte und den kundenspezifischen Produktvarianten. Aus diesen Informationen können Start- und Endtermine von Arbeitsvorgängen festgelegt, Arbeitsgänge den Maschinen zugeordnet und ein Kapazitätsausgleich zwischen Maschinen durchgeführt werden, die Verfügbarkeit der Roh-, Hilfs- und Betriebsstoffe geprüft und Aufträge in Produktion genommen werden. Schließlich wird eine Bearbeitungsreihenfolge der Aufträge an einem Arbeitsplatz bzgl. minimaler Gesamtdurchlaufzeit, minimaler Kapitalbindung, maximaler Kapitalauslastung und minimaler Umrüstkosten festgelegt.

3. *C-Anwendungen*
   Die primären Aufgabengebiete der C-Anwendungen sind der Produktentwurf, die Arbeitsplanung und die Ausführung der Produktion, bei denen der Entwickler durch die Anwendungen *Computer Aided Design* (*CAD*), *Computer Aided Engineering* (*CAE*), *Computer Aided Planning* (*CAP*) und *Computer Aided Manufacturing* (*CAM*) unterstützt wird. Die mit Hilfe von CAD-Anwendungen erstellten Konstruktionszeichnungen werden von CAE-Anwendungen in simulationsfähige Modelle abgebildet und simuliert. Ferner geben CAD-Anwendungen Auskunft über die der CAM-Anwendung bereitzustellenden Roh-, Hilfs- und Betriebsstoffe. CAP-Anwendungen dienen der teilautomatischen Gewinnung von Arbeitsplänen (Fertigungsvorschriften), die aus den Geometrie- und Stücklistendaten der CAD-Systeme und bereits bestehenden Arbeitsplänen abgeleitet werden. Die CAM-Anwendung, die die computergestützte Produktion auf Basis der Fertigungsvorschriften koordiniert, steuert Maschinen, flexible Fertigungssysteme, Roboter etc., verwaltet sowohl den Roh- und Hilfsstoffluß von und zu den Lagern als auch durch den Maschinenpark und unterstützt die Automatisierung der Vorgänge Transportieren, Lagern, Prüfen und Verpacken.

4. *Prognoseanwendungen*
   Durch Prognoseanwendungen, die den Absatz von Enderzeugnissen prognostizieren, wird auf der Basis von historischen Daten der Produktion der Primärbedarf an Fertigungskapazitäten bestimmt. Prognoseanwendungen unterstützen somit das Management bei der Planung und Erfüllung der Unternehmensziele.

5. *Anwendungen zur Beschaffung, Lagerhaltung und zum Versand*
   Auf der Grundlage von aktuellen Lagerbeständen, der aktuellen und der prognostizierten Auftragslage werden der Roh-, Hilfs- und Betriebsstoffbedarf ermittelt und gegebenenfalls eine automatische Bestelldisposition und -überwachung im Zusammenhang mit einer Wareneingangsprüfung durchgeführt oder alternativ eine Eigenproduktion veranlaßt. Der Lagerbestand, der nach Menge und Wert u.a. für Kostenrechnungen, Inventurerstellung und Buchführung fortgeschrieben wird, setzt sich aus Eigen- und Fremdprodukten zusammen. Fertigprodukte werden mit Hilfe von CAS-Anwendungen zur Lieferung an Kunden zusammengestellt und versandt. Die Abläufe im Lager können automatisiert werden, so daß zum Beispiel fahrerlose Transportsysteme (FTS) in Hochregallagern Rohstoffe und Produkte sortieren, beschaffen, ablegen oder direkt der Produktion übergeben können.

2.1.2 Unterschiedliche Sichten und Definition

Das CIM-Beispiel verdeutlicht die typische Hardware- und Softwarestruktur eines Informationssystems, die in der Literatur[3] als *verteiltes System* definiert ist. Ein verteiltes System besteht aus einer Menge lokaler, getrennter und homogener oder heterogener Rechnerknoten[4], die über Netzwerke[5] miteinander in Verbindung stehen, und aus Software, die die Kontrolle der verteilten Komponenten integriert und vereinheitlicht. Ein verteiltes System wird nach außen als eine Einheit verstanden und ist charakterisiert durch die gemeinsame Nutzung von Ressourcen, Offenheit, Nebenläufigkeit, Skalierbarkeit, Fehlertoleranz und Transparenz.

Die Sicht eines Informationssystems als verteiltes System ist rein technisch geprägt. Die Verrichtung der Aufgaben der Anwender sowie die benötigte Koordination und Kommunikation zwischen diesen bleibt außer Betracht. Fachlich gesehen dient ein Informationssystem zur Unterstützung kooperativer qualifizierter menschlicher Tätigkeit, die in einzelnen oder mehreren *Arbeitsumgebungen* erbracht wird. In einer *Arbeitsumgebung* werden *Arbeitsmittel* (*Werkzeuge* und *Automaten*) und *Arbeitsgegenstände* (*Materialien*) anwendungsfachlich motiviert zusammengestellt ([Gryczan 95]). „Unter kooperativer Arbeit sollen Arbeitssituationen verstanden werden, in denen mehrere Personen zusammenarbeiten zwecks Erreichung eines Ergebnisses, ..." ([Oberquelle 91])

Betrachten wir den Produktentwickler im CIM-Beispiel, so besteht seine Arbeitsumgebung aus Reißbrett, Stiften und Papier bzw. aus einer CAD-Anwendung, die ihm *Werkzeuge* und *Materialien* zum Design und zur Konstruktion von Produkten bereitstellt. Zur kundenspezifischen Produktentwicklung muß er sich mit seinen Kollegen der Auftragsabwicklung und der Marketing-Abteilung abstimmen. Vor der Produktion hat sich der Entwickler mit den Mitarbeitern des Lagers zur Bereitstellung der erforderlichen Roh-, Hilfs- und Betriebsstoffe und mit den Kollegen der Produktion abzustimmen. Zwischen allen beteiligten Personen findet eine ständige Rückkopplung statt. Die Unterstützung der Anwender und die Aufteilung ihrer Arbeit muß also bei einer fachlichen Modellierung eines Informationssystems im Vordergrund stehen. Arbeitsteilung hat nach [Fäustle 92] zwei Gesichtspunkte:

1. *Dekomposition*: Zerlegung einer (gemeinsamen) Aufgabe in einzelne Tätigkeiten, die sowohl zeitlich als auch räumlich getrennt voneinander durchgeführt werden können.

[3] u.a. [Kleinöder 92], [Kowalski 93], [Coulouris et al. 94]

[4] Rechnerknoten sind zum Beispiel Hosts, Terminals, PCs, Workstations und Netzwerkrechner.

[5] Netzwerke können zum Beispiel *Local Area Networks* (*LAN*), *Metropolitan Area Networks* (*MAN*) oder *Wide Area Networks* (*WAN*) sein.

2. *Distribution*: Initiale und dynamische Zuordnung der Tätigkeiten zu Organisationseinheiten oder Anwendern.

Damit Informationssysteme nahtlos in die fachliche Arbeitswelt integriert werden können, ist es erforderlich, neben den technischen Gegebenheiten auch die fachlichen Konzepte der Arbeitsdomäne adäquat in den Kontext eines Informationssystems zu überführen. Zu diesem Zweck orientiert sich die Methode WAM[6] am Arbeitsplatz für qualifizierte menschliche Tätigkeit und nutzt im Rahmen dieses Leitbilds die *Entwurfsmetapher* „Umgang mit Werkzeug und Material", die dem Anspruch gerecht zu werden versucht, fachlich analysierte und entworfene verteilte Anwendungen nahtlos in technische Anwendungen zu überführen.

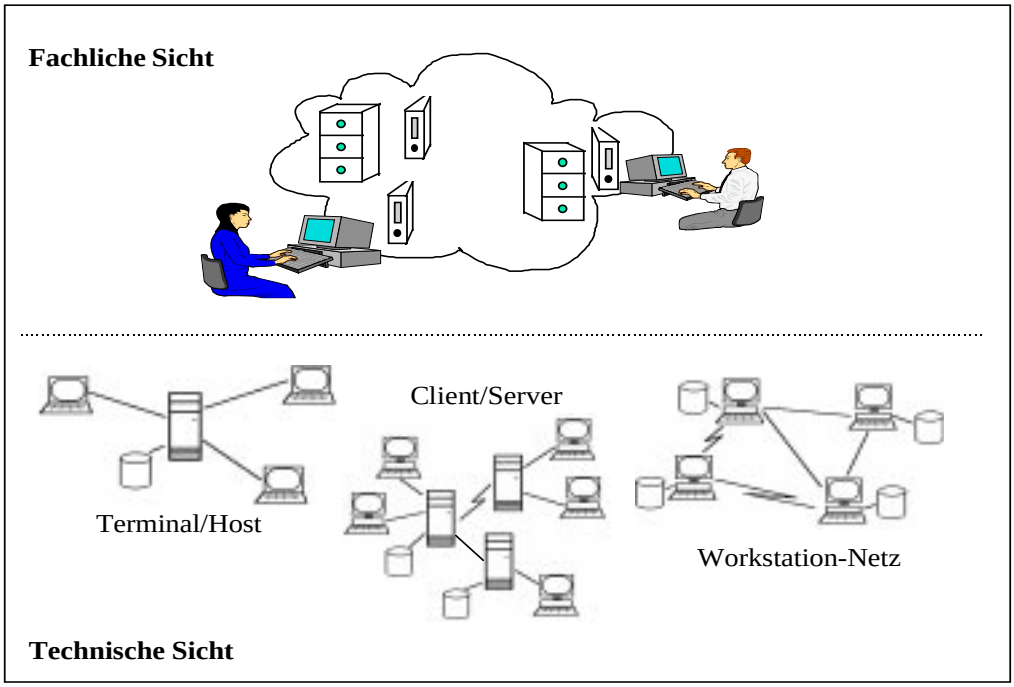


Abbildung 2.2: Technische und fachliche Sicht auf ein Informationssystem

Neben der Charakterisierung des Informationssystems als verteiltes System tragen die oben aufgeführten Beispiele die Merkmale datenintensiver Anwendungen. Diese sind gekennzeichnet durch[7]:

- Langlebigkeit großer Mengen von Datenobjekten
- Wahrung der Konsistenz der verarbeiteten Daten
- Unterschiedliche Sichten auf ein und denselben Datenbestand

[6] Eine eingehende Beschreibung der Methode WAM kann dem Kapitel 3 entnommen werden.
[7] vgl. [Wetzel 94]

- Mehrbenutzerfähigkeit
- Unterschiedliche Zugriffsberechtigungen (Zugriffsschutz, Security)
- Fehlererholung (Recovery)
- Effizienz
- Wiederkehrende Muster von Objektbeziehungen

Ferner zeichnen sich datenintensive Anwendungen durch eine ad-hoc Anfragemöglichkeit aus. Trotz ihrer Mächtigkeit gewinnt der navigierende Zugriff auf Objekte an Bedeutung.

Bevor in den nächsten Abschnitten die Rolle der unterschiedlichen Komponenten eines Informationssystems, die Entwicklungsziele und das Objekt als Informationsträger näher betrachtet werden, soll abschließend der Begriff des Informationssystems, der in der Literatur uneinheitlich und undifferenziert verwendet wird ([Schneider 91]), fachlich und technisch gefaßt werden. Die Abbildung 2.2 verdeutlicht diese zwei Sichten auf ein Informationssystem.

> **Begriff 1 :** *Informationssystem*
> Ein Informationssystem ist ein langlebiges und umfangreiches System, das einer ständigen Weiterentwicklung und zunehmender Komplexität unterworfen ist.
>
> fachliche Interpretation:
> Ein Informationssystem präsentiert sich dem qualifizierten Anwender in Form von Arbeitsumgebungen, die die Kooperation, Koordination und Kommunikation zwischen Anwendern und den Umgang mit gemeinsam genutzten Arbeitsgegenständen und Arbeitsmitteln unterstützen. Dabei können Anwender unterschiedliche Sichten auf gemeinsam genutzte Arbeitsgegenstände haben.
>
> technische Interpretation:
> Ein Informationssystem ist ein verteiltes System, dessen Aufgabe es ist, große Mengen langlebiger Daten sicher, flexibel und problemadäquat zu verwalten, zu manipulieren und darzustellen.

2.2 Das Objekt als Informationseinheit

Der Begriff des Objekts ist fundamental für die objektorientierte Softwareentwicklung. Aus diesem Grund wollen wir in diesem Abschnitt den Objektbegriff näher betrachten. Das im folgenden verwendete technische Begriffsverständnis beruht im wesentlichen auf [Booch 91] und [Geppert 97].

> **Begriff 2 :** *Objekt*
> Ein Objekt besitzt einen Zustand, ein Verhalten und eine Identität. Der Zustand eines Objekts ist immer auf einen Zeitpunkt bezogen und wird durch die aktuelle Ausprägung seiner Merkmale bestimmt. Diese Merkmale können konstant, d.h. in der Zeit unveränderlich, oder im Gegensatz dazu variabel sein. Das Verhalten eines Objekts wird durch seine Operationen und Reaktionen auf empfangene Nachrichten festgelegt.

Im Gegensatz zum Objekt, das als eine konkrete Einheit in Zeit und Raum existiert, beschreibt eine Klasse die gemeinsame Struktur und das gemeinsame Verhalten ihrer Exemplare.

> **Begriff 3 :** *Klasse*
> Die Struktur und das Verhalten gleichartiger Objekte sind in ihrer gemeinsamen Klasse festgelegt. Die Struktur einer Klasse setzt sich aus ihren Merkmalen (*properties*) zusammen. Merkmale sind Eigenschaften (*attributes*) und Beziehungen (*relationships*). Methoden, die die Schnittstelle nach außen definieren, legen das Verhalten fest.

Klassen dienen damit als „Erzeugungsmuster" von Objekten. Aus diesem Grund werden Objekte auch als Exemplare[8] einer Klasse bezeichnet.

2.2.1 Verknüpfung und Komposition von Objekten

Bei der technischen Modellierung von Informationssystemen sind viele Klassen über die Definition von Beziehungen miteinander verbunden. Diese typische statische Struktur verdeutlicht, daß die Betrachtung einer isolierten Klasse wenig Auskunft über die zu verrichtende Aufgabe eines Informationssystems gibt. Übertragen wir diese Erkenntnis auf ein Exemplar einer Klasse, bedeutet dieses, daß ein Objekt zwar eine Funktion innerhalb eines Informationssystems übernimmt, jedoch das Verhalten eines Informationssystems erst durch das Zusammenspiel vieler Objekte, die untereinander in Beziehung stehen, charakterisiert wird. Die in Beziehung stehenden Objekte bilden eine *Objektstruktur*, die wir im folgenden als *vernetzt* bezeichnen wollen.

Der zentrale Gegenstand des Informationssystems eines Verlages ist das Buch, das stark vereinfacht aus einem Titel und einer Menge von Seiten besteht, von einem Autor geschrieben wurde und einem Verlag zugeordnet ist. Der Klassenentwurf bestehe neben der Klasse *Buch* aus den Klassen *Titel*, *Seite*, *Autor* und *Verlag*, mit denen die Klasse *Buch* in Beziehung steht. Die Beziehungen zwischen

[8] In der Literatur wird zudem der Begriff der Instanz verwendet, der auf einer fehlerhaften deutschen Übersetzung des englischen Wortes *instance* beruht.

dem Exemplar eines Buchs und seinen zugeordneten Exemplaren von Titel, Autor, Verlag und Seiten bilden somit eine vernetzte Objektstruktur.

Betrachten wir exemplarisch die Umgangsform „Vervielfältigen" eines Buchs, so fällt auf, daß beim Kopieren eines Objekts Buch zwar das Buch selbst und die in Beziehung stehenden Objekte Titel und Seiten nicht jedoch die Objekte Autor und Verlag kopiert werden dürfen. Dies ist damit zu begründen, daß der Titel und die Seiten direkt zum Buch gehören, während der Autor und der Verlag nur als *Verweise* und nicht als *Bestandteil* eines Buchs verstanden werden können.

Aus dem Beispiel läßt sich unmittelbar ableiten, daß die *uni-direktionalen Beziehungen* zwischen den Objekten einer vernetzten Objektstruktur unterschiedliche Qualität haben[9]. Aus diesem Grund sei die *Kompositions-* von der *Verknüpfungsbeziehung* unterschieden. Eine Verknüpfungsbeziehung ist als eine gerichtete Beziehung zwischen Objekten zu verstehen, die die Sichtbarkeit eines Objekts von einem anderen ausdrückt. Die Komponentenbeziehung ist eine stärkere Ausprägung der Verknüpfungsbeziehung, bei der durch die Beziehung zusätzlich die Abhängigkeit eines Objekts von einem anderen ausgedrückt wird. Während zwischen einem Buch und einer Buchseite eine Kompositionsbeziehung besteht, existiert zwischen dem Buch und einem Autoren eine Verknüpfungsbeziehung.

Ein *komplexes Objekt* (zum Beispiel das Buch) besteht aus einer Menge von Objekten, die in Kompositionsbeziehung stehen. Die Objekte eines komplexen Objekts bilden zusammen eine konzeptionelle bzw. physikalische Einheit, deren Bearbeitung als Ganzes möglich ist. Nach [Hohenstein et al. 96] werden komplexe Objekte derzeit von den meisten Programmiersprachen und Datenbanksystemen höchstens sehr eingeschränkt unterstützt[10].

In Kapitel 4 und Kapitel 6 greifen wir die Begriffe der Verknüpfung und Komposition erneut auf und stellen sie im Kontext von Materialien bzw. Materialobjekten ausführlich dar.

2.2.2 Lebensdauer und Beweglichkeit

Ein Informationssystem erzeugt und manipuliert während seines Einsatzzeitraums eine große Anzahl von Objekten. Jedes dieser Objekte besitzt eine andere Lebensdauer und kann prinzipiell zwischen den Rechnerknoten oder den einzelnen Anwendungen migrieren. Während in der Literatur meist von Persistenz und Mobilität der Objekte gesprochen wird, soll in dieser Arbeit von der Lebensdauer

[9] Der Anhang A gibt eine Übersicht über die Eigenschaften von Beziehungen zwischen Objekten.

[10] Jedoch gerade bei objektorientierten Datenbanksystemen könnten komplexe Objekt als ein Mittel zur physikalischen Clusterung sehr dienlich sein.

bzw. der Beweglichkeit der Objekte gesprochen werden.

Lebensdauer

Ein Informationssystem erzeugt und manipuliert während seiner Laufzeit eine große Menge von Objekten. Diese können nach [Atkinson et al. 83] unterschiedliche Lebensdauern haben:

- *Transienter Gültigkeitsbereich*
  - Objekt als Ergebnis einer Auswertung eines Ausdrucks
  - Objekt innerhalb eines lokalen Sichtbarkeitsbereichs (Prozedur, Funktion, Operation)
  - Global bekanntes oder dynamisch erzeugtes Objekt
- *Persistenter Gültigkeitsbereich*
  - Objekt, das das Ausführungsende einer Anwendung[11] überlebt
  - Informationssystemversion überlebendes Objekt
  - Informationssystemende überlebendes Objekt

Nach [Heuer 92] kann die Lebensdauer ,,als Abstraktion des Speichers angesehen werden, indem wir die krassen Unterschiede zwischen Hauptspeicher und externem Speicher aufheben und nur definieren, wie lange wir ein Objekt behalten wollen." Typischerweise unterstützen jedoch weder Programmiersprachen, die lediglich transiente Objekte verwalten, noch Datenbanksysteme, die ausschließlich persistente Objekte verwalten, die Aufhebung dieser Speichertrennung. Seit Ende der achtziger Jahre sind allerdings vielversprechende Bemühungen zu verzeichnen, die vertikale Speichertrennung zwischen Hauptspeicher und Sekundärspeicher zu überbrücken und Programmiersprachen und Datenbanksysteme zusammenzuführen. In diesem Zusammenhang sind die Begriffe *persistente Programmiersprache* und *Datenbankprogrammiersprache* entstanden, die in Kapitel 5 diskutiert werden.

Da noch näher auf die Abstraktion von der vertikalen Speichertrennung eingegangen wird, soll an dieser Stelle auf eine weitere Diskussion verzichtet werden. Der sich aufdrängenden Frage nach der Art der Fortpflanzung der Lebensdauer in vernetzten Objektstrukturen wird in Unterabschnitt 5.5.2 nachgegangen.

[11] Im Unterschied zum Informationssystemende, bei dem das Informationssystem außer Betrieb genommen wird, meint Ausführungsende das (kurzfristige) Beenden einer Anwendung eines Informationssystems, die erneut gestartet werden kann.

Beweglichkeit

Die Beweglichkeit von Objekten läßt sich nach [Mathiske 96b] wie folgt klassifizieren:

- *Immobiles Objekt*
  Ein immobiles Objekt kann seinen erzeugenden Prozeß nicht verlassen.
- *Mobiles Objekt*
  Ein mobiles Objekt ist zwischen Prozessen eines Rechnerknotens, zwischen Rechnerknoten eines verteilten Systems oder zwischen verteilten Systemen verschiebbar.
- *Ubiquitäres Objekt*
  In allen Prozessen eines verteilten Systems gibt es gleichzeitig Objekte mit einer bestimmten, gleichen Funktionalität, auf die in derselben Weise zugegriffen werden kann[12].

Die Problematik unterschiedlicher Beweglichkeit von Objekten einer vernetzten Objektstruktur und die Frage nach der Fortpflanzung der Eigenschaft der Beweglichkeit von Objekten in vernetzten Objektstrukturen sind nicht Schwerpunkt dieser Diplomarbeit. Deshalb sei auf nachfolgende Arbeiten verwiesen.

2.3 Komponenten und Entwicklungsziele

Die ausreichende Unterstützung von beliebiger Lebensdauer und Beweglichkeit von Objekten ist, wie bereits erwähnt, eine zentrale Frage bei der Entwicklung von Informationssystemen. Die naheliegende Idee, die Bereitstellung beliebiger Lebensdauer und Beweglichkeit in einer Komponente zusammenzuführen, ist für unterschiedliche Komponenten (Programmiersprache, Datenbanksystem und Betriebssystem) berücksichtigt und/oder umgesetzt worden:

- *Programmiersprache*
  Beispiele: PANDA, DPS-Algol
- *Datenbanksystem*
  Beispiele: Oracle, Poet, ObjectStore
- *Betriebssystem*
  Beispiel: Mach

Während im Bereich der Programmiersprachen und der Betriebssysteme lediglich wissenschaftliche Realisierungen existieren, werden im Bereich der Datenbanksysteme auch kommerzielle Produkte angeboten. Aus diesem Grund ist die Entwicklung von Informationssystemen über mehrere Jahrzehnte von der Entwicklung der

[12] zum Beispiel Objekte einer GUI-Bibliothek

Datenbanksysteme geprägt worden. Dies entspricht aus heutiger Sicht nur noch eingeschränkt einer adäquaten Umsetzung eines Informationssystems:

- Nicht objektorientierte Datenbanksysteme bieten, wenn überhaupt, nur eine unzureichende (Datenbank-)Programmiersprache an, um (verteilte) Anwendungen, die über die reine Verarbeitung der in der Datenbank gehaltenen Datenbestände hinausgehen, zu realisieren.

- Die in Datenbanksystemen entwickelten Anwendungen, die auf einem Server ausgeführt werden, sind in der Regel monolithisch und komplex.

- Datenbanksysteme präsentieren sich in Form von Datenbankmasken (Frontend einer Datenbank) auf den Rechnerknoten eines verteilten Systems, die zu „dummen Terminals" degradiert werden.

- Die Entwicklung autonomer Anwendungen, die dezentral in einem verteilten System ausgeführt werden, auf gemeinsame Objekte zugreifen und nicht ausschließlich über Datenbankmechanismen miteinander kommunizieren, wird durch Datenbanksysteme nicht unterstützt.

- Anwender können über Datenbankanfragen Daten einpflegen, manipulieren, löschen oder abfragen. Eine qualifizierte menschliche Arbeit wird dabei nur eingeschränkt unterstützt, da zum einen der Umgang mit den Daten(sätzen) nur teilweise dem Umgang mit den Arbeitsgegenständen der Anwendungswelt entspricht und zum anderen die Arbeitsvorgänge in der Regel durch eine festgelegte Maskenabfolge vorgegeben sind.

- Nicht objektorientierte Datenbankmodelle bieten kein adäquates Mittel zur Analyse und zum Entwurf der Datenschemata eines Unternehmens. Darüber hinaus sind Unternehmensdatenschemata aufgrund der ständigen organisatorischen Anpassung einer kontinuierlichen Fortentwicklung unterworfen, der die Datenbanksysteme nur starr und unflexibel begegnen.

Aufgrund dieser Konsequenzen kommt der Zusammenführung von Datenbanksystemen, Betriebssystemen und Programmiersprachen, bei der die jeweiligen Stärken dieser Komponenten berücksichtigt werden, eine besondere Bedeutung zu. [Mathiske 96b] stellt jedoch fest, daß eine *paarweise Kluft* zwischen Programmiersprachen, Datenbanksystemen und Systemen für Verteilung (Betriebssystemen) besteht und legt ferner dar, daß es auch eine Kluft zwischen persistenten und verteilten Programmiersystemen gibt. Diese Kluft hat zwei Dimensionen:

- Komponenten, die beliebige Lebensdauer unterstützen, stellen zu wenig Mobilität und Komponenten, die beliebige Beweglichkeit unterstützen, zu wenig Persistenz bereit. Des weiteren stellen die Komponenten Dienste bereit, die sich jeweils paarweise überlappen und nur bedingt aufeinander aufbauen.

- Die Komponenten entspringen unterschiedlicher Paradigmen und Sichten, so daß ein Strukturbruch (*impedance mismatch*) zwischen diesen besteht, der lediglich durch die Abbildung der unterschiedlichen Konzepte überbrückt werden kann.

Deshalb werden Informationssysteme gegenwärtig durch aufwendige Kombinationen von Programmiersprachen sowie Diensten von Betriebs- und Datenbanksystemen erstellt ([Mathiske 96b]).

Der Überbrückung des *impedance mismatch* wird in Kapitel 5 intensiv nachgegangen. Als Grundlage dafür soll zunächst auf die Rollen von Datenbanksystemen, Betriebssystemen und Programmiersprachen eingegangen werden, die Entwicklungsziele eines Informationssystems dargelegt und ein Architekturvorschlag vorgestellt werden.

2.3.1 Rolle des Datenbanksystems

Ein Datenbanksystem besteht aus einer *Datenbank* und der zugehörigen Datenbanksoftware, dem *Datenbankverwaltungssystem*. Es bildet eine Abstraktionsebene über der zugrundeliegenden Hardware und dem Betriebssystem zur Bereitstellung von Persistenz. Das Datenbankverwaltungssystem, auch als Datenbankmanagementsystem (DBMS) bezeichnet, ist ein Programmsystem, das Datenbanken (Sekundärspeicher) verwaltet. Ein DBMS realisiert die sichere, zuverlässige und langfristige Verwaltung von großen, integrierten, von vielen Benutzern verwendeten Datenmengen und stellt eine Datendefinitionssprache (DDL), eine Datenmanipulationssprache (DML) sowie eine Anfragesprache für den komfortablen Zugriff darauf bereit ([Geppert 97]). Eine Datenbank ist eine Sammlung von inhaltlich zusammenhängenden Daten, die in Dateien mit kontrollierter Redundanz abgespeichert werden, um für mehrere Anwendungsprogramme und Benutzer in bestmöglicher Weise verwendbar zu sein.

Einer Datenbank liegt ein konkretes *Datenbankschema* (Datenbankbeschreibung) zugrunde, das nach einem *Datenbankmodell* erstellt werden kann[13]. Ein Datenbankmodell stellt nach [Heuer, Saake 95] datenbanksystem-spezifische Konzepte zur Beschreibung von Datenbanken zur Verfügung. Es legt somit die Syntax und die Semantik zur Beschreibung der Datenbankschemata fest.

Die ersten Datenbankmodelle, die in den sechziger Jahren entwickelt wurden, sind das hierarchische und das Netzwerk-Modell. Sie lehnen sich an die Datenstrukturen (Zeigerstrukturen zwischen Datensätzen) der Programmiersprachen PL/I,

[13]Datenbankschemata werden irrtümlicherweise auch als Datenmodelle (zum Beispiel Unternehmensdatenmodelle) bezeichnet.

Cobol und Fortran an. Während das Datenbankprodukt IMS auf dem hierarchischen Modell basiert, liegt dem Datenbankprodukt UDS das Netzwerk-Modell zugrunde. Ersteres wird zwar in der freien Wirtschaft bis heute eingesetzt, hat aber unserer Erfahrung nach auf die Neuentwicklung datenintensiver Anwendungen keinen Einfluß mehr.

In den siebziger und achtziger Jahren wurden die relationalen Datenbanksysteme (RDBS) entwickelt, die auf dem Relationenkalkül nach Codd beruhen. Ihre besondere Leistung besteht in der deskriptiven Beschreibung von Anfragen und dem abstrakten Konzept der Relation, das die Grundlage zur Bearbeitung von Massendaten bereitstellt. Die Tabellenstruktur der Datenbanken paßt eingeschränkt zu den Datenstrukturen imperativer Programmiersprachen (Cobol, Pascal, C) und nicht zu denen objektorientierter Programmiersprachen (Eiffel, Smalltalk, C++). Die standardisierte deklarative Anfragesprache SQL (Structured Query Language) bildet eine sprachliche Schnittstelle eines Datenbanksystems. Da SQL nicht berechnungsvollständig (*computational completeness*) ist, bedarf es neben der Nutzung von SQL stets einer Programmiersprache zur Formulierung von Algorithmen. Die relationalen Datenbanksysteme (zum Beispiel DB2, Oracle) haben sich am Markt durchgesetzt und sind heute marktbeherrschend ([Silberschatz et al. 96]).

Bereits Anfang der achtziger Jahre wurde die Unzulänglichkeit des relationalen Datenmodells für Nicht-Standard-Anwendungen erkannt. Die in diesen Anwendungen verwendeten Objekte und Objektstrukturen müssen bei der Abbildung auf Relationen segmentiert und in der Umkehr aufwendig desegmentiert werden, was sowohl zu Ineffizienz während der Anwendungsentwicklung als auch zur -laufzeit führt. Ende der achtziger und verstärkt Anfang der neunziger Jahre wurden deshalb die ersten objektorientierten Datenbanksysteme (OODBS) entwicklelt (zum Beispiel Poet, ObjectStore). Die von diesen Systemen bereitgestellten Konzepte zur Objektstrukturierung, die der objektorientierten Sichtweise entspringen, können aus heutiger Sicht für die adäquate und natürliche Strukturierung eingesetzt werden und sind somit prädestiniert für die Entwicklung von Informationssystemen.

Für die Entwicklung objektorientierter Datenbanksysteme gibt es drei unterschiedliche Herangehensweisen[14]:

1. *Evolutionärer Ansatz*
   Erweiterung des relationalen Datenmodells um objektorientierte Konzepte (SyBase, Oracle)

[14] Abbildung 2.3 stellt eine Klassifizierung von Datenbanksystemen dar.

2. *Revolutionärer Ansatz*
   Entwicklung objektorientierter Datenbankmodelle (ODMG-93/95)[15]

3. *Programmiersprachlicher Ansatz*
   Es sind bei dieser Herangehensweise zwei Entwicklungslinien zu unterscheiden:

   (a) Entwicklung eines Datenbanksystems durch Übertragung und Erweiterung des Objektmodells einer bestimmten Programmiersprache. Diese Sprache bildet gleichzeitig die Programmierschnittstelle des Datenbanksystems (Poet und C++, GemStone und Smalltalk)

   (b) Erweiterung einer Programmiersprache um Datenbankfunktionalität und einen persistenten Speicher (E, O++, Texas)[16]

Die Datenbanksysteme des evolutionären Ansatzes werden häufig als Objektdatenbanksysteme, die des revolutionären und programmiersprachlichen Ansatzes als objektorientierte Datenbanksysteme bezeichnet.

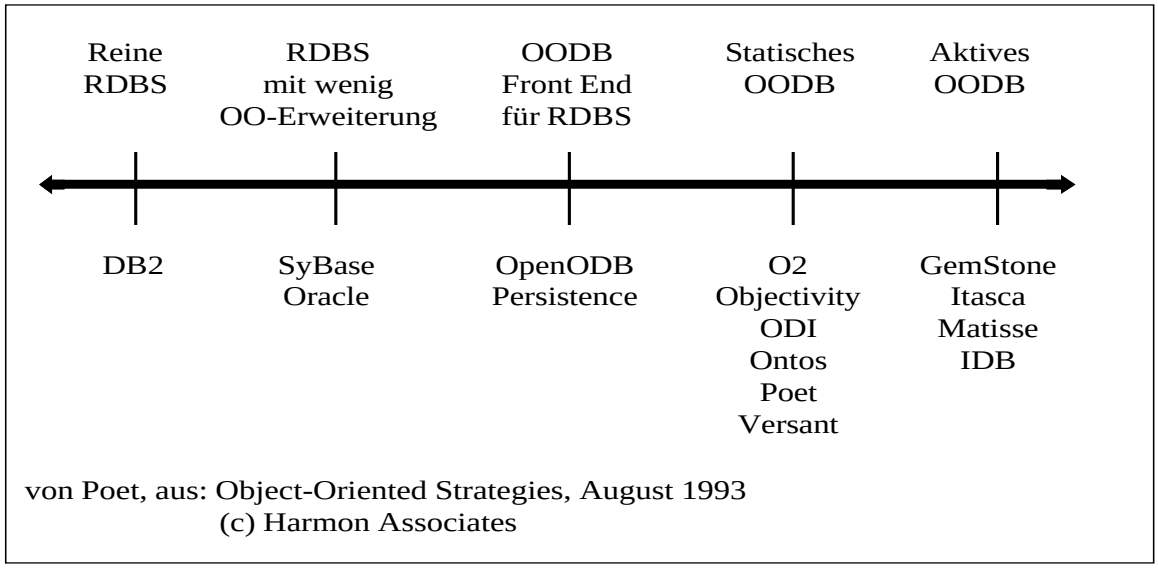


Abbildung 2.3: Klassifikation von Datenbanksystemen

[15]Die Object Database Management Group beschrieb 1993 eine erste objektorientierte Datenbanknorm (Version 1.1), die 1995 nur geringfügig erweitert wurde (Version 1.2). Im Laufe des Jahres 1997 soll die Norm in der Version 2.0 erscheinen.

[16]Eine ausführliche Beschreibung dieser persistenten Programmiersprachen und Literaturverweise finden sich in [Brammer, Göhmann 96].

Die OODBS, die die Trennung von Daten und zugehörigen Funktionen (Code) sowie die Koexistenz von Programmier- und Datenbanksprache aufheben, setzen sich unserer Erfahrung nach am Markt bisher nur schwer durch. Die Gründe hierfür sind vielfältig. Im Gegensatz zu der geringen Erfahrung mit OODBS und der allgemein geringen Akzeptanz des objektorientierten Ansatzes (Denkweise) in der freien Wirtschaft, ist die Technologie der RDBS ausgereift[17], zuverlässig und performant. Ferner besteht ein großes Potential an Wissen über die standardisierten RDBS. Da durch den weiteren Einsatz von RDBS die Infrastruktur bereits bestehender Systeme erhalten bleiben kann, bildet der Einsatz von RDBS einen Investitionsschutz für Unternehmen. Nicht zuletzt bedeutet eine Umstellung auf OODBS eine langwierige und schwierige Migration der sensiblen Datenbestände[18].

2.3.2 Rolle des Betriebssystems

Das Fundament von Informationssystemen bilden Hardware-Komponenten, die sich historisch von isolierten Rechenanlagen über Hostsysteme mit Terminalanschluß sowie Client/Server-Systeme hin zu offenen und heterogenen Workstationnetzen entwickelt haben. Betriebssysteme abstrahieren von der zugrundeliegenden Hardware-Architektur und Netzwerktopologie[19] und erlauben damit den technisch transparenten physikalischen Austausch sowie die permanente Ablage von Daten. Ein Betriebssystem dient damit nicht nur der Integration und Vereinheitlichung der Kontrolle unterschiedlicher Hardware, sondern bildet daneben eine erste Abstraktionsstufe zur Unterstützung von beliebiger Lebensdauer und Beweglichkeit von Objekten.

Ein Betriebssystem stellt eine (hierarchische) Dateiverwaltung zur Ablage von Byte-Strömen bereit, auf die von nur einem Prozeß (*singletasking*) oder mehreren (*multitasking*) und einem Benutzer (*singleusing*) oder mehreren (*multiusing*) zugegriffen werden kann. Ein gemeinsamer Schreibzugriff, der gleichzeitig auf genau einen Byte-Strom (Datei) erfolgt, wird ausgeschlossen. Ferner kann der Zugriff auf Daten durch Zugriffsberechtigungen beschränkt und ein Datenschutz mit Hilfe von User-ID und Paßwort erreicht werden. Ein Betriebssystem stellt einen direkt-adressierbaren Hauptspeicher (zum Beispiel 1MB) und einen *virtuellen Speicher* (zum Beispiel 64GB) zur temporären Ablage von Daten (Byte-Blöcke) zur Verfügung.

[17] OODBS stellen neben navigierenden Anfragemöglichkeiten nur Ansätze von deklarativen Anfragesprachen bereit. Ferner sind die Konzepte des Datenschutzes und der Zugriffskontrolle besonders im Hinblick auf vernetzte Objektstrukturen noch nicht ausgereift.

[18] Namhafte Konzerne arbeiten noch heute mit dem IMS System. Zum Beispiel hat die Deutsche Bank AG vor zwei Jahren beschlossen, IMS mindestens bis zum Jahr 2000 einzusetzen.

[19] Eine Netzwerktopologie (zum Beispiel Ringnetz, Busnetz, Sternnetz oder Vermaschung) beschreibt die strukturelle Verknüpfung von Rechnern.

Ein Betriebssystem stellt nach [Atkinson, Morrison 89] eine abstrakte (high level) Maschine dar,

- die von der zugrundeliegenden Hardware abstrahiert.
- die die (nebenläufige) Ausführung von Prozessen organisiert.
- die die (Primär-, Sekundär-, Tertiär-)Speicher verwaltet und organisiert (virtuelles Speichermanagement, Datenschutz, Datensicherheit).
- die die Datenübertragung organisiert.
- die eine Kontrollsprache bereitstellt, mit deren Hilfe ein Benutzer die Funktionalität des Betriebssystems nutzen kann.

Ein Betriebssystem ist auf der einen Seite zwar sehr performant und einfach im Umgang, stellt aber ausschließlich primitive Mechanismen zur Unterstützung von Persistenz (Dateisystem) und Mobilität (zum Beispiel Socket-Kommunikation) von Byte-Strömen zur Verfügung. Weitere Probleme bei der Bereitstellung von Persistenz und Mobilität sind[20]:

- Ausschließlich primitive Manipulations-Operatoren über Byte-Strömen
- Keine Unterstützung von Programmiersprachen-Strukturen
- Keine Konsistenzmechanismen
- Schlechte Skalierbarkeit bei grober Speichergranularität, weil große Speichersegmente versandt werden

Ein Betriebssystem stellt jedoch mächtige Mechanismen zur Kommunikation und zum Datentransfer zwischen Rechnerknoten[21] bzw. Primär-, Sekundär- und Tertiärspeicher bereit. Darüber hinaus steuert und überwacht es die Gesamtheit der Programme eines Rechnersystems.

Einen viel versprechenden Ansatz, beliebige Beweglichkeit von Objekten bereits in systemnaher Software zu realisieren, zeigt das objektorientierte Betriebssystem Mach[22]. Mach ist in der zweiten Hälfte der achtziger Jahre an der Carnegie Mellon University, Pittsburgh unter der Zielsetzung entwickelt worden,

- große virtuelle Adreßräume zu verwirklichen,

[20] vgl. [Haines 95],[Simmel, Godard 92],[Mathiske 96b]

[21] zum Beispiel TCP/IP

[22] Eine kurze Einführung der Leistungsmerkmale von Mach liefern [Coulouris et al. 94] und [Mairandres 96].

- Interprozeßkommunikation mit Schutzmechanismen bereitzustellen und
- die Unterstützung von Multiprozessorsystemen zu gewährleisten.

Besonders interessant für diese Arbeit ist das Konzept des virtuellen Adreßraums, der Mechanismen zur einfachen Implementierung eines virtuell gemeinsamen Speichers zur Verfügung stellt[23]. Das Konzept des virtuell gemeinsamen Speichers[24], das in der Literatur als *Distributed Shared Memory* (*DSM*) bezeichnet wird, bildet auch die Grundlage für einige verteilte Programmiersprachen[25].

2.3.3 Rolle der Programmiersprache

Da bereits eine ausführliche Betrachtung von *persistenten Programmiersprachen* in [Brammer, Göhmann 96] und von mobilität-unterstützenden Programmiersprachen in [Mathiske 96b] vorgenommen worden ist[26], soll an dieser Stelle nur kurz und allgemein auf Programmiersprachen und ihre Konzepte zur Unterstützung von beliebiger Lebensdauer und Beweglichkeit eingegangen werden.

Eine Programmiersprache ist ein formales und auf einem Computersystem realisiertes Notationssystem zur Formulierung von Algorithmen und dient somit zur Entwicklung ablauffähiger Software. Eine Programmiersprache unterliegt einem bestimmten Paradigma[27] und stellt einen Satz von Daten- und Kontrollstrukturen bereit, die zur Abstraktion von der zugrundeliegenden Hardware dienen. Während Maschinensprachen Binärcode, der genau auf einer Hardware lauffähig ist, erzeugen, stellen höhere Programmiersprachen Konstrukte bereit, die durch Compiler oder Interpreter auf unterschiedlicher Hardware ablauffähig sind.

Da weder Betriebssysteme noch Datenbanksysteme (abgesehen von OODBS) Konstrukte zur Umsetzung von Algorithmen zur Verfügung stellen, ist die Entwicklung von Informationssystemen sowie die Realisierung einzelner Anwendungen eines Informationssystems ohne die Nutzung von Programmiersprachen nicht

[23] „Während früher in Betriebssystemen die Sichtweise vertreten war, Anwendungen im Hauptspeicher zu verwalten und im Fall von Engpässen auf Hintergrundspeicher auszulagern ..., hat sich in den letzten Jahren die umgekehrte Betrachtungsweise mehr und mehr durchgesetzt. Auf dem Hintergrundspeicher werden die Daten der Anwendung organisiert und gespeichert, der Hauptspeicher wird lediglich als Cache für die zur Ausführung benötigten Teile genutzt ...." ([Kleinöder 92])

[24] Weitere Betriebssysteme mit einem virtuell gemeinsamen Speicher sind u.a. Chorus, Clouds, SOS (vgl. [Borghoff, Nast-Kolb 89]).

[25] u.a. DPS-Algol, PANDA

[26] Ferner gibt [Borghoff, Nast-Kolb 89] einen nicht nur auf Programmiersprachen beschränkten Überblick über Systeme, die beliebige Beweglichkeit unterstützen.

[27] imperativ (Modula), funktional (Lisp), objektorientiert (Eiffel), deklarativ (SQL) etc.

denkbar. Obwohl Programmiersprachen einer ständigen Weiterentwicklung unterworfen sind und die Forderungen nach der Bereitstellung von Persistenz und Mobilität auch an Programmiersprachen gestellt werden, ist diesem Wunsch bislang nicht entsprochen worden. Weder die frühen Programmiersprachen, wie zum Beispiel Fortran oder Cobol, noch die objektorientierten, wie zum Beispiel Eiffel oder Smalltalk, die auch in der kommerziellen Programmierung eingesetzt werden, unterstützen die persistente Ablage und die Mobilität von Daten.

Einen viel versprechenden Ansatz, neben der beliebigen Beweglichkeit auch die beliebige Lebensdauer von Objekten in einer Programmiersprache zu unterstützen, bietet die Programmiersprache PANDA[28], die eine Betriebssystemerweiterung und eine Klassenbibliothek für C++ bereitstellt. Die Programmiersprache PANDA erweitert den *virtuellen Speicher* zu einem *Distributed Shared Memory*, der den gemeinsamen Zugriff auf Objekte auch über das Ausführungsende eines Programms hinaus erlaubt. Dabei werden jedoch die ausgereiften Konzepte der Datenbanksysteme weder genutzt noch dem Programmierer bereitgestellt. Zum vertiefenden Verständnis der Konzepte von PANDA sei der Leser auf die Arbeiten [Assenmacher et al. 93] und [Assenmacher et al. 94] verwiesen.

2.3.4 Entwicklungsziele

Im ersten Abschnitt dieses Kapitels sind wir bereits auf die fachliche und technische Sicht eines Informationssystems eingegangen. An die fachlichen und technischen Anteile eines Informationssystems werden u.a. die nachfolgenden Anforderungen gestellt.

Unterstützung des qualifizierten Anwenders

Das *Leitbild des qualifizierten Mitarbeiters*[29] beruht auf der Annahme, daß der Anwender eines Informationssystems für die Ausübung seines Berufs hinreichend qualifiziert ist. D.h., daß der Anwender befähigt ist, seinen beruflichen Aufgaben selbständig und ohne Unterstützung anderer nachzukommen. Somit muß ein computergestützter Arbeitsplatz den Anwender unterstützen und ihn von lästigen Routineaufgaben befreien, ohne ihn damit zu einem reagierenden Wesen zu degradieren. Der Anwender soll bei der Gestaltung seiner Arbeit nicht behindert oder eingeschränkt, sondern ausreichend unterstützt werden. Dazu gehört auch, daß er seine computergestützten Werkzeuge zielgerichtet einrichten und einsetzen kann. Ferner muß der Anwender im Rahmen einer kooperativen menschlichen

[28]PANDA ist an der Universität Kaiserslautern, Fachbereich Informatik, mit dem Ziel entwickelt worden, basierend auf der Programmiersprache C++ eine technisch transparente, parallele Programmierung zu ermöglichen.

[29]Sichtweise des am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg vertretenen Ansatzes zur Entwicklung von Software, siehe u.a. [Gryczan 95].

Tätigkeit, die durch ein hohes Maß an Koordination und Kommunikation mit anderen geprägt ist, technisch ausreichend unterstützt werden.

Bei der Entwicklung eines Informationssystems „ist es sicher lohnend, sich über die Art der Kooperationsunterstützung klar zu werden. Entwicklungsziel sind Systeme, die unterstützen, nicht automatisieren; ergänzen, nicht ersetzen; neutral verbinden, integrieren, nicht Zusatzaufgaben schaffen; gruppenbezogene Autonomie forcieren und Verhandlungen unterstützen (Oberquelle zitiert von Piepenburg)." ([Kirsche 94])

Bereitstellung fachlicher Transparenz

Beim Einsatz computergestützter Werkzeuge ist der Umgang des qualifizierten Anwenders mit den Arbeitsgegenständen der Anwendungsdomäne (zum Beispiel mit Notizen, Verträgen und Ordnern) nachzuempfinden. Dazu muß die fachliche Anwendungssituation, zum Beispiel "Der Kunden-Ordner liegt auf dem Schreibtisch von Herrn Maier und wird von ihm bearbeitet" explizit gemacht werden. D.h., daß sowohl der Ort von Arbeitsgegenständen als auch die Anwesenheit anderer Anwender des Systems erfahrbar sein müssen. Darüber hinaus ist insbesondere der simultane Zugriff auf Arbeitsgegenstände zu verdeutlichen.

[Gryczan 95] spricht in diesem Zusammenhang von der fachlichen Transparenz, die er wie folgt definiert:

- *Lokalität (Ortstransparenz/Verteilungstransparenz)*
  „Benutzer sind über den Ort eines Materials informiert. Ein Material befindet sich immer an einem bestimmten Ort. Dieser Ort ist die Umgebung oder ein Archiv, aus dem ein Material beschafft werden kann."

- *Zugriff (Zugriffstransparenz)*
  „Die Orte, an denen auf Materialien zugegriffen werden kann, sind über [ein Medium] ... bekannt. Der Zugriff findet nach einem einheitlichen Schema statt."

- *Migration (Migrationstransparenz)*
  „Die Bewegung eines Materials an einen anderen Ort kann nur durch eine Aktivität eines Benutzers erfolgen."

- *Simultaner Zugriff (Nebenläufigkeitstransparenz)*
  „Simultaner schreibender Zugriff aus verschiedenen Umgebungen auf ein Material ist nicht möglich. Er wird durch das Konzept von Kopie und Original verhindert."

- *Kopienverwaltung (Replikationstransparenz)*
  „Liegt in der Verantwortung der Benutzer."

- *Ausfall (Fehlertransparenz)*
  „Der systembedingte Ausfall von Orten, an denen Material für Umgebungen bereitgestellt wird, wird nicht verborgen."

Dieser Begriff der Ortstransparenz besagt, daß die räumliche Lage von Arbeitsgegenständen und die Ausführung von Tätigkeiten bekannt ist. Dabei wird keine Aussage über die Abstraktion der Entfernung zwischen Orten gemacht.

Bereitstellung technischer Transparenz

Von dem fachlichen Umgang mit computergestützten Werkzeugen, dem Wissen über den Aufenthaltsort von Arbeitsgegenständen, der Bearbeitung dieser Gegenstände an einem Ort und der fachlichen Zusammenstellung von Arbeitsmitteln und -gegenständen ist die tatsächliche Ausführung von Operationen und Algorithmen von Anwendungen sowie die physikalische Lage von Objekten auf den Rechnerknoten eines verteilten Systems zu unterscheiden. Somit ist der fachliche Ort (zum Beispiel *Arbeitsumgebung*, Archiv), an dem sich Arbeitsgegenstände befinden und bearbeitet werden, von der tatsächlichen Lage und Ausführung der Bearbeitung auf den einzelnen Rechnerknoten zu trennen. In diesem Zusammenhang wird von der technischen Transparenz[30] gesprochen:

- *Ortstransparenz/Verteilungstransparenz* ([Borghoff, Nast-Kolb 89])
  Ressourcen werden angefordert ohne Wissen, wo sie liegen.

- *Zugriffstransparenz* ([Borghoff, Nast-Kolb 89])
  Gleiche Zugriffsart auf lokale und entfernte Daten.

- *Migrationstransparenz* ([Parrington 92])
  Die Benutzer sehen keine Objektverschiebung zwischen Rechnerknoten.

- *Nebenläufigkeitstransparenz* ([Borghoff, Nast-Kolb 89])
  Die Benutzer sehen weder nebenläufige Handlungen noch deren Synchronisation.

- *Replikationstransparenz* ([Borghoff, Nast-Kolb 89])
  Ein Benutzer sieht genau ein Objekt, unabhängig davon, ob und wieviele Replikate existieren.

- *Fehlertransparenz* ([Borghoff, Nast-Kolb 89])
  Die Benutzer bemerken keine Netzwerkfehler. Nicht abgeschlossene Modifikationen sind nach außen nicht sichtbar.

[30] Der Begriff der technischen Transparenz ist durch die im Englischen übliche Bedeutung des Wortes Transparenz geprägt (Durchsichtigkeit). Im Deutschen wird in der Regel das Gegenteil (Einsichtigkeit) darunter verstanden (vgl. den Begriff der fachlichen Transparenz).

- *Transparenz der Leistungserbringung* ([Klein 91])
  Die Dauer der Leistungserbringung in einem verteilten System schwankt ähnlich der in einem lokalen System.

Dem Anwender eines Informationssystems bleibt die technische Realisierung des Informationssystems verborgen, ohne auf die Leistungsmerkmale von verteilten Systemen verzichten zu müssen:

- *Datenverbund*: Dezentrale Datenhaltung
- *Lastverbund*: Hohe Verfügbarkeit und Lastverteilung durch dezentrale Funktionsausführung
- *Leistungsverbund*: Effizienz durch Parallelisierung komplexer Anwendungen
- *Programm-/Geräteverbund*: Gemeinsame Nutzung von Ressourcen

Weitere Anforderungen

Neben der technischen Transparenz und den in den letzten Unterabschnitten bereits erwähnten Entwicklungszielen der Bereitstellung der Persistenz und der Mobilität von Objekten werden weitere technische Anforderungen formuliert:

- *Softwareintegration*
  Die Entwicklung von Informationssystemen ist meist nur dann wirtschaftlich, wenn bereits getätigte Investitionen in bestehende Softwarekomponenten (zum Beispiel Datenbanksysteme) weiterhin genutzt werden können. Dazu bedarf es der Entwicklung von Softwarekomponenten, die eine Integration unterschiedlichster Software unterstützen und eine abstrakte und allgemeingültige Schnittstelle bereitstellen ohne die Leistungsmerkmale der speziellen Softwarekomponenten zu ignorieren.

- *Plattformunabhängigkeit*
  Analog zu der Softwareintegration muß sich ein (teilweise) neuentwickeltes Informationssystem der bereitgestellten Hardware und dem zugrundeliegenden Betriebssystem anpassen. Weil darüber hinaus die Heterogenität der Hardwarekomponenten eines verteilten Systems steigt, ist die Portabilität von Informationssystemkomponenten von wachsender praktischer Bedeutung.

- *Flexibilität*
  Bei der Entwicklung von Informationssystemen wird das Ziel verfolgt, Anwender bei der Verrichtung ihrer Aufgaben zu unterstützen. Informationen müssen dabei flexibel und problemadäquat verwaltet, manipuliert und dargestellt werden können ([Mathiske 96b]). Da zudem die Organisationsstruktur eines Unternehmens einer ständigen Weiterentwicklung und damit der

Veränderung der Aufgabengebiete von Organisationseinheiten und Anwendern unterworfen ist, ist es erforderlich, daß computergestützte Werkzeuge, Anwendungen und Anwendungssysteme flexibel zusammengestellt werden können. Dabei ist zu berücksichtigen, daß der sensible Datenbestand eines Unternehmens erhalten bleibt.

2.3.5 Architektur

Eine Architektur spezifiziert nach [Booch 94] die Klassen- und Objektstruktur eines Softwaresystems. Läßt sich die Klassen- und Objektstruktur in Schichten untergliedern, wird auch von einer Schichtenarchitektur gesprochen. Unter einer Schicht wird die Zusammenarbeit von Klassen und Objekten verstanden, die einer (hierarchisch) höheren Schicht bestimmte Dienste zur Verfügung stellen. Durch die Benennung von Schichten wird es möglich, Details von höheren Schichten gegen die Details von niedrigeren Schichten abzugrenzen ([Booch 94]).

Bei der architektonischen Konzeption von Informationssystemen stellt sich die Frage, auf welche Weise die fachliche und die technische Sicht zusammenführbar sind bzw. wie die fachlichen und technischen Komponenten gekoppelt werden können. Gehen wir in einer ersten Näherung davon aus, daß die fachliche Sicht durch fachliche Komponenten einer Anwendungsschicht und die technische Sicht durch rein technische Komponenten einer Abstraktionsschicht umgesetzt werden. Gehen wir ferner davon aus, daß die Anwendungsschicht direkt Dienste der Abstraktionsschicht nutzt, so führt das zu einer Abhängigkeit zwischen diesen beiden Schichten (2-Schicht-Architektur). Weil die Anwendungsschicht sich direkt der Dienste der Abstraktionschicht bedient, schlagen sich technische Details auf die Anwendungsschicht und/oder umgekehrt fachliche Anteile auf die Abstraktionsschicht durch. Dies impliziert eine Vermischung von fachlichen und technischen Anteilen sowohl in der Anwendungs- als auch der Abstraktionsschicht und widerspricht somit den rein fachlich und technisch motivierten Sichten eines Informationssystems.

Deshalb wollen wir uns bei der Entwicklung von Informationssystemen an einer Ausprägung der sogenannten 3-Schicht-Architektur, die Gerald Schröder[31] vorschlägt, orientieren. Diese besteht aus drei Abstraktionsebenen ([Schröder 96]):

- **Anwendungsschicht** (Fachliche Anwendung, Frontend)
  anwendungsabhängig, ressourcenunabhängig
  - Benutzerabhängige Komponenten mit Programmlogik und Benutzeroberfläche
  - Abwicklung der Benutzerinteraktion und lokaler Operationen

[31] Wissenschaftlicher Mitarbeiter am Arbeitsbereich TIS, Technische Universität Harburg

- **Adaptionsschicht** (Anwendungsserver, Middleware)
  anwendungsabhängig, ressourcenabhängig
  - Bereitstellung der gemeinsam genutzten Komponenten
  - Koordination von Transaktionen, Nebenläufigkeit, Datenschutz und Datensicherheit
  - Bereitstellung technischer Transparenz (zum Beispiel Abstraktion von der vertikalen und horizontalen Speichertrennung)
- **Abstraktionsschicht** (Speicher, Datenbankserver, Betriebssystem)
  anwendungsunabhängig, ressourcenabhängig
  - Bereitstellung der Persistenz und Mobilität
  - Ausführung der Datenbank-Transaktionen
  - Abbildung zwischen (Objekt-)Repräsentationen

Die Abbildung 2.4 veranschaulicht die 3-Schicht-Architektur und stellt sie der 2-Schicht-Architektur gegenüber.

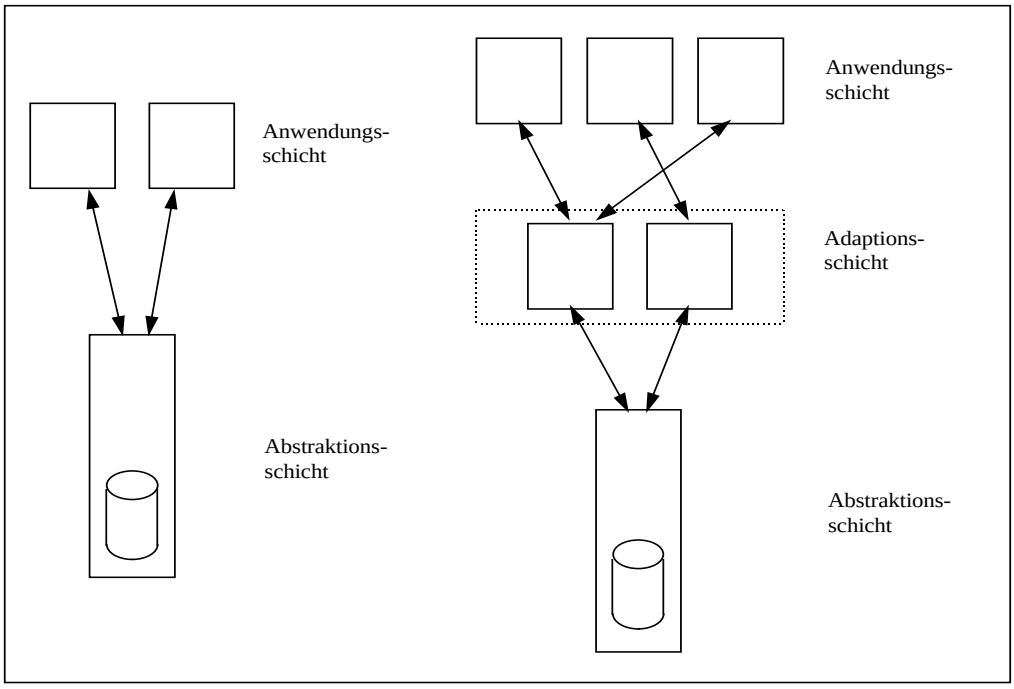


Abbildung 2.4: 2-Schicht- und 3-Schicht-Architektur

Mit Hilfe dieses Architekturvorschlags lassen sich die gestellten Entwicklungsziele an Informationssysteme in besonderer Weise umsetzen. Während die Ab-

straktionsschicht eine rein technische Schicht[32] ist, die die Softwareintegration unterstützt und eine plattform-unabhängige Schnittstelle präsentiert, dient die Adaptionsschicht der Abstraktion von der vertikalen Speichertrennung[33] und der horizontalen Speichertrennung[34] (technische Transparenz). Somit können Anwendungen auf der Anwendungsschicht losgelöst von technischen Details auf einfache Weise nach dem Leitbild des qualifizierten Anwenders (fachliche Transparenz) konzipiert und implementiert werden. Ferner hat dieser Architekturvorschlag auf die Entwicklung von Informationssystemen folgende Auswirkungen:

- Vorteile:
  - Keine komplexe Client-Schicht, in der die Anwendungs- und Adaptionsschicht realisiert werden muß
  - Reduzierung der Abhängigkeiten zwischen fachlichen und technischen Komponenten durch die Einführung einer mittleren Schicht
  - Flexibilität und Skalierbarkeit eines Informationssystems, weil Komponenten der Abstraktionsschicht beliebig austauschbar sind
  - Effiziente Benachrichtigungsmechanismen zum Beispiel bei Datenveränderung
  - Reduzierung des Netzwerkverkehrs (Erhöhung der Performanz)
  - Erfüllung der Forderung nach der Modularisierung, der Austauschbarkeit etc.
- Nachteile:
  - Keine direkte Nutzung von Leistungen der Komponenten der Abstraktionsschicht (zum Beispiel Nebenläufigkeitskontrolle, Transaktionen und Datenwiedergewinnung)
    => Konsistenzkontrolle muß erneut in der Adaptionsschicht implementiert werden
  - Größere Systemkomplexität
    => mehr Verantwortung beim Entwickler
  - Datenbank evtl. nur ,,dummer" Speicher

Die klare Festlegung der Verantwortlichkeiten und damit verbunden die klare Trennung der Aufgaben bei einer Schichtenarchitektur führen auf natürliche

[32]Sie besteht mindestens aus dem Betriebssystem, das von der Hardware-Architektur und der Netzwerktopologie unter Bereitstellung von Primitiven zur Verteilung abstrahiert, und aus einem Datenbanksystem, das vom Dateisystem abstrahiert und einen sicheren und konsistenten Datenspeicher bereitstellt.

[33]Trennung zwischen dem Primär-, Sekundär- und Tertiärspeicher

[34]Trennung zwischen den Speichern unterschiedlicher Rechnerknoten

Weise zu Client/Server-Architekturen ([Mathiske 96b]) oder Workstation-Server-Architekturen. Dabei ist zu überlegen, welche Abstraktionsebenen von der Client-Anwendung und welche von der Server-Anwendung bereitgestellt werden.

2.4 Gewählte Aufgabenstellung

Zu Beginn dieses Abschnitts seien zwei provokante Äußerungen erlaubt:

- Obwohl der *virtuelle Speicher* auf der einen Seite als Konzept zur Überbrückung der vertikalen Speichertrennung und damit zur persistenten Ablage von Objekten und auf der anderen Seite zur Überbrückung der horizontalen Speichertrennung erkannt wurde, sind nur wenige Bemühungen (zum Beispiel PANDA) unternommen worden, das Konzept des *virtuellen Speichers* zur Aufhebung aller Speichertrennungen einzusetzen.

- Die Entwickler von Betriebssystemen, Datenbanksystemen und Programmiersprachen entwickeln über ihre jeweils spezifischen Problemstellungen hinaus Lösungen, die in die Bereiche der jeweils anderen Entwickler fallen. So kommt es, das viele Probleme doppelt gelöst werden (vgl. [Rahm 93]), die Funktionalität von Betriebssystemen, Datenbanksystemen und Programmiersprachen sich überschneiden und diese Komponenten nur eingeschränkt aufeinander aufbauen und damit kombinierbar sind.

Das Ziel der im Rahmen dieser Arbeit entworfenen und spezifizierten Erweiterung der Programmiersprache C++ ist die Bereitstellung eines einheitlichen programmiersprachlichen und architektonischen Frameworks[35], das technisch transparente Persistenz und Mobilität von beliebigen Objekten bereitstellt. Durch die Unterstützung direkter, unkomplizierter Abbildungen der fachlichen Anwendungsstrukturen in korrespondierende objektorientierte Programmiersprachenkonstrukte und in durch das Framework bereitgestellte erweiterte Konstrukte sowie die technisch transparente Nutzung der zugrundeliegenden technischen Komponenten gelingt eine Erhöhung der Flexibilität und Qualität bei gleichzeitiger Minimierung des Aufwands der Entwicklung von Informationssystemen. Unter einem Framework verstehen wir nach [Flor 96] eine generische Lösung, die aus kooperierenden Klassen besteht und für verwandte Probleme unter der Vorgabe von Abläufen konzipiert ist. Abbildung 2.5 setzt den Begriff des Frameworks mit den Begriffen des Entwurfsmusters, der Komponente und der Architektur in Beziehung.

Um zukunftsweisende Fortschritte in der Entwicklung von fachlich motivierten Informationssystemen zu erarbeiten, konzentriert sich die vorliegende Diplomarbeit auf folgende Teilaufgaben:

[35] Synonym: Rahmenwerk

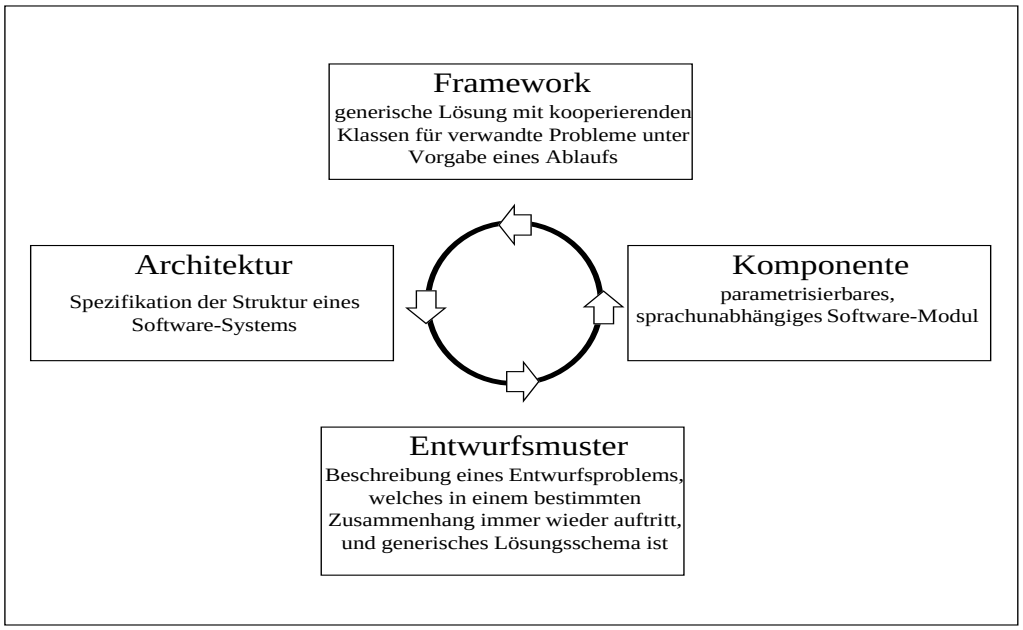


Abbildung 2.5: Zusammenhang der Begriffe Entwurfsmuster, Komponente, Framework und Architektur nach [Flor 96]

1. Entwicklung eines fachlichen Basiskonzepts auf der Grundlage einer Überarbeitung der Materialverwaltung mit dem Schwerpunkt, fachliche Konzepte von technischen zu trennen.

2. Bereitstellung der Transparenz der Persistenz und Mobilität von Objekten.

3. Entwurf eines Anschlusses zur Anbindung paradigmatisch unterschiedlicher Datenbanksysteme.

Bei allen drei Teilaufgaben besteht ein vielversprechender Ansatz darin, die positiven Eigenschaften der Programmiersprache C++ dazu zu nutzen, die Sprache durch das Konzept des *Distributed Shared Memory* zu einer *mobilität-unterstützenden persistenten Programmiersprache* zu erweitern.

Die konkret verfolgten Problemstellungen und Lösungsansätze lassen sich wie folgt näher charakterisieren:

1. *Entwicklung eines fachlichen Basiskonzepts auf der Grundlage einer Überarbeitung der Materialverwaltung mit dem Schwerpunkt, fachliche Konzepte von technischen zu trennen* (Kapitel 3 und 4).
   Für die Verwaltung von *Materialien* und für den Zugriff auf diese steht zur Zeit das Konzept der Materialverwaltung ([Wulf, Weske 94], [Gryczan 95]) zur Verfügung, das aus der fachlichen Sicht nicht motivierbar ist. Um zudem eine klare Trennung der fachlichen und technischen Konzepte und Komponenten zu erreichen, ist eine fachlich generelle Abstraktionsebene zu finden

und zu beschreiben, die die abstrakte Behandlung der technischen Komponenten durch die fachliche Welt ermöglicht und aus der fachlichen Perspektive betrachtet intuitiv als unterer Abschluß der Anwendungswelt verstanden werden kann. Eine solche fachlich generelle Abstraktion bietet das Konzept des *Materialraums*, der alle existierenden *Materialien* verwahrt und diese von außen zugreifbar macht. In diesem Zusammenhang ist der für das Konzept des *Materialraums* fundamentale Begriff des *Materials* in seiner fachlichen und technischen Ausprägung näher zu betrachten.

2. *Bereitstellung der Transparenz der Persistenz und Mobilität von Objekten* (Kapitel 5 und 6).
   Zwischen Programmiersprachen und Datenbanksystemen aber auch zwischen unterschiedlichen Datenbanksystemen besteht eine Kluft, die nur durch aufwendige Kombination von Programmiersprache und Datenbanksystem überbrückt werden kann. Dadurch liegt der Gesamtaufwand der Programmierung des Codes zur Einbettung von Datenbanksystemen bei über 30 Prozent des gesamten Entwicklungsaufwands eines Informationssystems[36]. Eine *persistente Programmiersprache*, die darüber hinaus auch die beliebige Beweglichkeit von Objekten unterstützt, kann den Entwickler erheblich entlasten. Zur Erweiterung der Programmiersprache C++ zu einer mobilität-unterstützenden persistenten Programmiersprache bedienen wir uns des Konzepts des *Distributed Shared Memory*, der den technisch transparenten Umgang mit transienten und persistenten aber auch lokalen und entfernten Objekten erlaubt. Besonders interessant sind dabei folgende Fragestellungen:

   (a) Sind die Schnittstellen von C++ und die der Datenbanksysteme aufeinander abgestimmt?

   (b) Wer ist für die Konsistenzerhaltung der Datenbestände zuständig?

   (c) Wer übernimmt die Synchronisation und Ablaufsteuerung nebenläufiger Prozesse?

   (d) Inwieweit können die Dienste der Datenbanksysteme zur Realisierung von Persistenz, Nebenläufigkeitskontrolle und Konsistenzerhaltung der Datenbestände genutzt werden?

   (e) Welche Funktionalität muß doppelt und welche zusätzlich realisiert werden?

   (f) Wer stellt Meta-Objekt-Wissen bereit?

   (g) Wie können getrennte Speicher- und Adreßräume zusammengeführt und technisch transparent präsentiert werden?

[36] vgl. [Hohenstein et al. 96]

3. *Entwurf eines Anschlusses zur Anbindung paradigmatisch unterschiedlicher Datenbanksysteme* (Kapitel 5).
   Für diese Arbeit bilden die relationalen und objektorientierten Datenbanksysteme die Grundlage für die Bereitstellung von Persistenz. Bedingt durch den *impedance mismatch* zwischen ihren Paradigmen erweist sich die Festlegung einer gemeinsamen Schnittstelle als problematisch. Ferner bestehen Unterschiede zwischen den Objektmodellen der Programmiersprache C++ und denen objektorientierter Datenbanksysteme. Aus diesem Grund ist eine gemeinsame Schnittstelle für paradigmatisch unterschiedliche Datenbanksysteme zu suchen und zu beschreiben, die auf der einen Seite bereits heute von möglichst vielen Datenbanksystemen direkt unterstützt wird und an die auf der anderen Seite andere Datenbanksysteme angepaßt werden können. Eine solche Schnittstelle bietet die C++-binding des ODMG-93-Standards.

Kapitel 3

Modellierung eines Arbeitsplatzes für kooperative qualifizierte menschliche Tätigkeit

Zunächst wird in diesem Kapitel ein Überblick über die objektorientierte Anwendungsentwicklung nach der Methode WAM gegeben. Dazu werden die wesentlichen Begriffe *Werkzeug*, *Automat*, *Material* und *Aspekt* kurz eingeführt und in Beziehung zueinander gesetzt. Darauf aufbauend erfolgt die Darstellung des Begriffs der *Umgebung* und des für diese Arbeit relevanten Konzepts der Materialverwaltung. Abschließend wird dieses Kapitel durch eine kritische Würdigung der vorgestellten Begriffe und Konzepte abgerundet.

3.1 Einleitung

Damit Informationssysteme nahtlos in die Arbeitswelt integriert werden können, ist neben ihrer Kompatibilität zu bestehenden technischen Komponenten (Datenbanksysteme, Betriebssysteme, Netzwerke etc.) auch die Kompatibilität zur fachlichen Arbeitsdomäne erforderlich. Zu diesem Zweck muß neben dem softwaretechnischen auch das anwendungsspezifische Fachwissen in den Softwareentwicklungsprozeß einfließen.

Um bei den Entwicklern das nötige Verständnis von der Arbeitsdomäne zu schaffen, muß der Kommunikationsprozeß zwischen Softwareentwicklern auf der einen und den Anwendern auf der anderen Seite eine zentrale Schlüsselrolle einnehmen[1].

[1] „Einige Manager befürworten vielleicht die Idee der direkten Kommunikation zwischen Entwickler und Endanwender (für andere ist es geradezu erschreckend, sich vorzustellen, daß ein Anwender einen Entwickler überhaupt sieht!)" ([Booch 94])

Im traditionellen Software Engineering wird allerdings eine Minimierung der Kommunikationsbeziehungen zwischen den am Entwicklungsprozeß beteiligten Gruppen angestrebt ([Kilberth et al. 93]), so daß Vorgehensmodelle wie zum Beispiel das Wasserfallmodell nicht die nötige fachliche Kompatibilität eines Softwaresystems garantieren können. Vielmehr ist ein am Anwender ausgerichtetes Vorgehen erforderlich, das eine mit der fachlichen Welt rückgekoppelte und damit evolutionäre Softwareentwicklung ermöglicht.

Ein solches Vorgehensmodell bietet der Methodenrahmen STEPS[2], der die Softwareentwicklung als ein zyklisches Vorgehen, bei dem der Kommunikations- und Lernprozeß zwischen Anwendern und Entwicklern im Mittelpunkt steht, begreift (vgl. dazu [Floyd 91]). Da STEPS keinen umfassenden Satz von Werkzeugen und Darstellungsmitteln zur Entwicklung von Software bereitstellt, muß der Methodenrahmen durch eine zur Sichtweise von STEPS passende Methode gefüllt werden. Eine solche ist die im folgenden vorgestellte Methode WAM.

3.2 Die Methode WAM

Die Methode WAM[3] orientiert sich am Arbeitsplatz für qualifizierte menschliche Tätigkeit und nutzt im Rahmen dieses Leitbilds die *Entwurfsmetapher* ,,Umgang mit Werkzeug und Material". Damit eignet sich die Methode ,,besonders für die Entwicklung interaktiver Anwendungssysteme auf graphischen Arbeitsplatzrechnern" ([Kilberth et al. 93]).

Die Methode WAM dient der Zusammenführung der fachlichen und der technischen Modellierung eines Informationssystems in einem einheitlichen objektorientierten Modell. Auf diese Weise soll die Trennung zwischen der Analyse- und der Entwurfsphase aufgehoben werden ([Kilberth et al. 93]), so daß die Begriffe und Konzepte der fachlichen Welt ohne semantischen Bruch in ein Programmiermodell und damit in den technischen Kontext überführt werden können.

Im folgenden soll das Leitbild des Arbeitsplatzes für kooperative qualifizierte menschliche Tätigkeit kurz dargestellt werden, um anschließend die Begriffe von *Werkzeug*, *Automat*, *Material* und *Aspekt* einführen zu können.

---

[2]STEPS (**S**oftware**t**echnik für **E**volutionäre **P**artizipative **S**ystementwicklung) ist ein an der Technischen Universität Berlin entwickelter theoretischer und methodischer Ansatz der Softwaretechnik, siehe [Floyd 91].

[3]WAM (**W**erkzeug **A**utomat **M**aterial bzw. **W**erkzeug **A**spekt **M**aterial) ist eine an der Universität Hamburg entwickelte objektorientierte Methode, siehe [Kilberth et al. 93].

3.2.1 Das Leitbild des Arbeitsplatzes für kooperative qualifizierte menschliche Tätigkeit

Das Leitbild des Arbeitsplatzes für qualifizierte menschliche Tätigkeit basiert auf einer Reihe von Annahmen, die [Riehle 95] wie folgt beschreibt:

- „Die Benutzer besitzen die notwendige Kompetenz und Fähigkeit, ihre Arbeit selbständig zu erledigen.
- Es gibt keinen Grund, einen vordefinierten Arbeitsablauf einzuführen, da die Steuerung der Vorgänge durch die Benutzenden selbst flexibel gestaltet werden kann und soll.
- Die zu erledigenden anspruchsvollen Aufgaben verlangen individuell einstellbare und organisierbare (Software-)Umgebungen." ([Riehle 95])

Damit basiert die Methode WAM auf der Grundannahme, daß die Anwender „in ihrer Tätigkeit ausreichend qualifiziert sind, um die angebotenen Freiheitsgrade sinnvoll und verantwortlich nutzen zu können ... ." ([Riehle 95])

Das beschriebene Leitbild steht damit im krassen Gegensatz zu Softwaresystemen, die auf einer tayloristischen Arbeitsteilung beruhen. Systeme dieser Art degradieren Anwender zu „Bedienern", die vom Softwaresystem je nach Bedarf „getriggert" werden. Dabei werden die Arbeitsabläufe fest vorgegeben und fachlich inkonsistente Systemzustände nicht mehr zugelassen ([Riehle 95]).

Wie wir bereits in Kapitel 2 gezeigt haben, wird qualifizierte menschliche Tätigkeit nur selten isoliert ausgeführt. Viel öfter findet sie im Kontext mit anderen handelnden Personen statt. Aus diesem Grund wollen wir dieser Arbeit das Leitbild des Arbeitsplatzes für *kooperative* qualifizierte menschliche Tätigkeit zugrundelegen.

Begriff 4 : *Kooperation*
„Als Kooperation wird das Zusammenwirken (mindestens) zweier Personen bezeichnet, die ihre Handlungen auf Grundlage wechselseitig abgestimmter Ziele und Pläne ausführen." ([Gryczan 95])

Begriff 5 : *Kooperative Arbeit*
„Kooperative Arbeit ist Arbeit, bei der verschiedene Personen mit zwischen ihnen ausgehandelten Konventionen zur Koordination ein Ergebnis anstreben, das nur gemeinsam erreicht werden kann." ([Gryczan 95])

Begriff 6 : *Handlung*
„Eine Handlung ist die menschliche Aktivität, die untrennbar mit einem vom Menschen benennbaren Ziel verbunden ist." ([Gryczan 95])

Piepenburg stellt fünf Bedingungen für Kooperation ([Kirsche 94]):

1. *Zielidentität:*
   Die Ziele bzw. Teilziele der handelnden Personen müssen übereinstimmen.

2. *Veränderbarkeit der Pläne:*
   Die Handlungspläne der beteiligten Personen müssen potentiell durch die Interaktionen zwischen den Personen veränderbar sein (vgl. Piepenburgs Begriff der Plan-Kompatibilität).

3. *Ressourcenaustausch:*
   Neben den Ergebnissen kooperativer Arbeit müssen auch andere Ressourcen, wie zum Beispiel genutzte Arbeitsmittel und Arbeitsgegenstände, aber auch Zeit und Raum, zwischen den Personen prinzipiell ausgetauscht werden können.

4. *Regelbarkeit:*
   Es muß möglich sein, Handlungspläne von Personen von außen zu beeinflussen und Ausnahmesituationen zu behandeln.

5. *Kontrolle:*
   Die Handlungspläne der kooperierenden Personen müssen erkennbar sein, was zum Wunsch nach maximaler Durchschaubarkeit der Pläne führt.

Neben den Bedingungen für kooperative Arbeit können verschiedene Dimensionen für kooperative Tätigkeit unterschieden werden. Im folgenden soll exemplarisch eine für diese Arbeit grundlegende Auswahl benannt werden (vgl. [Kirsche 94]):

- *Ort:*
  Kooperative Tätigkeit kann an einem Ort (*nicht verteilt*) oder an unterschiedlichen Orten (*verteilt*) durchgeführt werden.

- *Zeit:*
  Kooperative Tätigkeit kann gleichzeitig (*synchron*) oder zu unterschiedlichen Zeiten (*asynchron*) stattfinden.

- *Anzahl der beteiligten Personen:*
  Die Anzahl der an der kooperativen Arbeit beteiligten Personen ist maßgeblich, weil sie den erforderlichen Kommunikations- und Koordinationsbedarf stark beeinflußt. So erschweren große Gruppen (> 10 Personen) die Kooperation ([Kirsche 94]).

- *Art des Vollzugs:*
  Es muß unterschieden werden, ob eine kooperative Arbeit von den beteiligten Personen *konjunktiv* oder *disjunktiv* vollzogen wird. Konjunktiver Vollzug heißt, daß die Aufgabe erst dann bearbeitet ist, wenn alle beteiligten Personen ihre Arbeit bis zum Ende durchgeführt haben. Daneben genügt es beim disjunktiven Vollzug, daß eine Person ihre Arbeit erledigt hat.

- *Art der gemeinsamen Nutzung von Arbeitsgegenständen:*
  Es gibt drei verschiedene Arten wie eine gemeinsame Nutzung von Arbeitsgegenständen erfolgen kann:

  1. *Sequentielle gemeinsame Nutzung:*
     Genau eine Person nutzt exklusiv einen Arbeitsgegenstand. Dieser Gegenstand kann nach vollendeter Handlung von einer anderen an der kooperativen Arbeit beteiligten Person übernommen und wieder exklusiv bearbeitet werden.
  2. *Parallele/Gleichzeitige gemeinsame Nutzung:*
     Es erfolgt eine gleichzeitige gemeinsame Nutzung von Arbeitsgegenständen ohne Synchronisation der Handlungen. Arbeitsgegenstände, die auf diese Weise gemeinsam genutzt werden, werden *gleichzeitig genutzt* genannt.
  3. *Simultane/Quasi-gleichzeitige gemeinsame Nutzung:*
     Es erfolgt prinzipiell eine gleichzeitige gemeinsame Nutzung von Arbeitsgegenständen ohne Synchronisation der Handlungen. Eine zeitliche Synchronisation der Zugriffe findet nur dann statt, wenn die beteiligten Personen auf ein und denselben Bestandteil eines Arbeitsgegenstands zugreifen wollen. Damit nutzt im Gegensatz zur sequentiellen gemeinsamen Nutzung keine Person den Arbeitsgegenstand exklusiv.

Neben den genannten Dimensionen kooperativer Tätigkeit gibt es noch eine Reihe weiterer Dimensionen wie zum Beispiel den Grad der Problemkomplexität, die Größe des Koordinations- bzw. des Unterstützungsbedarfs, aber auch soziale Merkmale wie zum Beispiel die Sprachform[4] oder den Sprechakttyp[5].

Jede Art von kooperativer Arbeit ist ohne eine Koordination, die zur wechselseitigen Abstimmung der handelnden Personen dient, nicht denkbar. Eine Koordination muß aber keineswegs immer explizit, zum Beispiel durch Diskussion zwischen den kooperierenden Personen, sondern kann auch implizit über gemeinsam genutzte Arbeitsgegenstände[6], erfolgen: „There are many aspects of cooperative

[4] Ausprägungen: assertiv (Feststellung), direktiv (Aufforderung), kommissiv (Versprechen), expressiv (Ausdruck von Meinungen) und deklarativ (Erklärung)

[5] Ausprägungen: Anfrage/Versprechen, Angebot/Annahme und Bericht/Quittung

[6] insbesondere gleichzeitig gemeinsam genutzte Arbeitsgegenstände

work. One of these is the use of shared material. Much cooperation is based on silent coordination mediated by the shared material used in the work process." ([Sörgaard 88])

3.2.2 Werkzeug, Automat und Material

Grundlage der Methode WAM ist die Identifizierung der in der fachlichen Welt relevanten Gegenstände. Diese können in Arbeitsmittel (*Werkzeuge* und *Automaten*) und Arbeitsgegenstände (*Materialien*) unterschieden werden. Die Begriffe lassen sich nach [Kilberth et al. 93], [Gryczan 95] und [Brammer, Göhmann 96] wie folgt fachlich und technisch definieren:

Begriff 7 : *Werkzeug*
fachliche Interpretation:
Ein Werkzeug ist im Rahmen einer Aufgabenerledigung ein Arbeitsmittel, mit dem Arbeitsgegenstände (Materialien) bearbeitet werden. Das Werkzeug kann Materialien verändern und deren Zustand sondieren.

technische Interpretation:
Ein Werkzeug spaltet sich in eine Funktionskomponente, in der die Funktionalität modelliert ist, und in eine oder mehrere Interaktionskomponenten, die das Werkzeug an der Benutzeroberfläche präsentieren, auf. Die Klassen der Komponenten stellen Operationen zur Verfügung, die den Umgangsformen des Benutzers mit dem Werkzeug entsprechen.

Begriff 8 : *Automat*
fachliche Interpretation:
Ein Automat ist im Rahmen einer Aufgabenerledigung ein Arbeitsmittel, das Arbeitsgegenstände (Materialien) nach einem fest vorgegebenen Algorithmus, der über einen längeren Zeitraum ablaufen kann, bearbeitet. Dabei verändert er wie ein Werkzeug den Zustand der Materialien.

technische Interpretation:
Ein Automat spaltet sich wie ein Werkzeug in eine Funktionskomponente und eine oder mehrere Interaktionskomponenten, die dem Anwender Parametereinstellungen ermöglichen, auf. Die Klassen der Komponenten stellen Operationen zur Verfügung, die den Umgangsformen des Benutzers mit dem Automaten entsprechen.

Begriff 9 : *Material*
fachliche Interpretation:
Ein Material ist im Rahmen einer Handlung ein Arbeitsgegenstand, der durch Werkzeuge und Automaten bearbeitet, d.h. verändert und sondiert, werden kann.

technische Interpretation:
Eine Materialklasse stellt Methoden zum Verändern und Sondieren des Zustands von Exemplaren der Materialklasse bereit.

Nach [Kilberth et al. 93] besitzt jedes Werkzeug einen internen Zustand, der als *Werkzeuggedächtnis* bezeichnet wird. Auch Materialien besitzen einen internen Zustand, der prinzipiell durch die Nutzung von Umgangsformen sondiert bzw. modifiziert werden kann[7]. Der Zustand eines Materials stellt damit die kumulierten Ereignisse seines Verhaltens dar (nach [Booch 94]).

Da die Begriffe Werkzeug und Automat auf der einen und Material auf der anderen Seite zyklisch definiert sind, ist die Festlegung, ob ein fachlicher Gegenstand Arbeitsmittel oder Arbeitsgegenstand ist, kontextabhängig und dynamisch ([Kilberth et al. 93]). Damit kann ein fachlicher Gegenstand während seiner Existenz ein Arbeitsmittel sein, obwohl er vorher Arbeitsgegenstand gewesen ist und umgekehrt: Ein Hammer verliert zum Beispiel seinen Werkzeugcharakter, wenn der Hammerstiel erneuert wird. Während dieser Zeit wird der Hammer zu einem Material.

Die Eigenschaft, ob ein fachlicher Gegenstand Material- oder Werkzeugcharakter hat, ist somit nicht objektiv festlegbar. Eine Festlegung kann ausschließlich dynamisch abhängig vom jeweiligen Verwendungszusammenhang erfolgen.

3.2.3 Die Aspektkopplung

Ein *Aspekt* beschreibt die von einem Werkzeug bzw. einem Automaten sichtbare Perspektive auf ein Material und legt die durch das Werkzeug bzw. durch den Automaten nutzbare Schnittstelle des Materials fest. *Aspekte* dienen damit zur Kopplung zwischen Werkzeugen und Automaten sowie den zu ihnen passenden Materialien.

[7] Die aktuelle Farbe eines Autos ist zum Beispiel Bestandteil seines Zustands. Durch Anwendung der Umgangsform „Umspritzen" kann dieses Merkmal in seiner Ausprägung verändert werden.

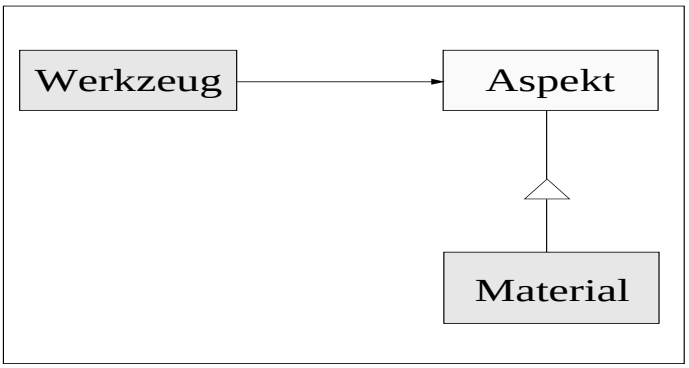


Abbildung 3.1: Die Aspektkopplung

Begriff 10 : *Aspekt*
fachliche Interpretation:
Ein Aspekt gibt an, ob ein Material für ein bestimmtes Arbeitsmittel (Werkzeug, Automat) geeignet ist und wie es von dem Arbeitsmittel verändert und sondiert werden kann. Ein Arbeitsmittel bearbeitet Materialien ausschließlich unter einem bestimmten Aspekt, d.h. sie benutzen nur ausgewählte Umgangsformen der Materialien.

technische Interpretation:
Eine Aspektklasse spezifiziert das Protokoll, das eine Materialklasse konkretisieren muß, um unter dem Aspekt bearbeitet werden zu können. Eine Klasse ist in einem Verwendungszusammenhang eine Werkzeugklasse/Automatenklasse, wenn sie eine Aspektklasse benutzt; eine Klasse ist eine Materialklasse, wenn sie Unterklasse einer Aspektklasse ist und in einem Verwendungszusammenhang von einer Werkzeugklasse/Automatenklasse benutzt wird.

3.3 Die Arbeitsumgebung

Um eine Organisation von Werkzeugen, Automaten und Materialien zu ermöglichen, wird ein Konzept benötigt, das wie ein Schreibtisch in der realen Welt dazu dient, unterschiedliche Arbeitsmittel und Arbeitsgegenstände aufzunehmen. Zu diesem Zweck wurde der Begriff der *Umgebung* eingeführt, für den es bislang keine einheitliche Definition gibt. Aus diesem Grund wollen wir im folgenden die zwei alternativen Sichtweisen von [Riehle 95] und [Gryczan 95] vorstellen.

3.3.1 Das Interpretations- und Gestaltungsmuster Umgebung

[Riehle 95] identifiziert das Konzept der *Umgebung* als ein Interpretations- und Gestaltungsmuster[8], das er mit weiteren aus dem Kontext der Methode WAM motivierten Mustern in Beziehung setzt:

> **Begriff 11 :** *Umgebung*
> „Werkzeuge [,Automaten] und Materialien befinden sich immer in einer Umgebung. Die Umgebung definiert die zur Organisation von Werkzeugen [,Automaten] und Materialien verwendbaren räumlichen und logischen Dimensionen sowie zwischen ihnen geltende Konsistenzbedingungen. Sie stellt ... eine äußere Hülle dar und schirmt die verwendeten Werkzeuge [,Automaten] und Materialien nach außen hin ab."([Riehle 95])

Diesem Umgebungsbegriff liegt die *Schreibtischmetapher* zugrunde. Der Schreibtisch in der realen Welt besitzt einen Begriff von Raum, der es Menschen ermöglicht, die Gegenstände darauf zu strukturieren und auf diese Weise die Arbeitsmittel und -gegenstände zu organisieren. Diese Tätigkeiten sind ein wesentlicher Bestandteil qualifizierter Arbeit ([Riehle 95]).

Der Umgebung kommen damit drei Grundaufgaben zu:

1. Durch die Etablierung von Umgebungen wird die fachlich spezifische Sicht auf einen Arbeitsplatz in der realen Welt simuliert. Damit kann durch die Umgebung das räumliche Enthaltensein von Arbeitsmitteln und -gegenständen rudimentär ausgedrückt werden. Wird der Umgebung zusätzlich ein Besitzer zugeordnet, so unterstützt die Umgebung ein triviales Konzept von Besitz.

2. Die Umgebung dient dazu, die Konsistenz der fachlichen Abhängigkeiten zwischen Materialien sowie zwischen Werkzeugen und Materialien zu überwachen. So wird ein Anwender beispielsweise von der Umgebung über Terminüberschneidungen informiert. „Auf dem physischen Schreibtisch gibt es üblicherweise kein Pendant zu diesen Konsistenzbedingungen." ([Riehle 95])

3. Die Umgebung ermöglicht die Überprüfung von Verwendungszusammenhängen (Aspekten) zwischen Arbeitsmitteln und -gegenständen bevor es zur Anwendung von Werkzeugen auf Materialien kommt.

[8] „Ein Interpretations- und Gestaltungsmuster ist ein Muster, welches zur Interpretation und Gestaltung von tatsächlichen oder antizipierten Anwendungssituationen und Softwaresystemen verwendet werden kann." ([Riehle 95])

Umgesetzt wird das Interpretations- und Gestaltungsmuster Umgebung durch das Entwurfsmuster[9] [10] Umgebungsobjekt, das die Aufgaben der Umgebung an ein Schreibtischobjekt, an ein oder mehrere Werkzeugkoordinatoren bzw. an eine Materialverwaltung delegiert:

- Aufgabe des *Schreibtischobjekts* ist die Visualisierung der Umgebung, die durch einen virtuellen Schreibtisch mit den darauf befindlichen Arbeitsmitteln und -gegenständen in Form von Bildschirm-Icons dargestellt wird.

- Die *Werkzeugkoordinatoren* dienen zur Benachrichtigung von Werkzeugen über Zustandsänderungen der von ihnen bearbeiteten Materialien. „Diese [Zustandsänderungen] können ohne Wissen des Werkzeugs auftreten, weil entweder mehrere Arbeitsmittel dasselbe Material bearbeiten oder die Konsistenzbedingungen zwischen Materialien ... zur Änderung der Materialien führen." ([Riehle 95])

- Die *Materialverwaltung* stellt die Schnittstelle der Umgebung zu externen Diensten dar und dient somit zum Beispiel zur Kapselung angeschlossener Datenbanksysteme.

Eine detaillierte Betrachtung der Materialverwaltung nach [Riehle 95] erfolgt in den Unterabschnitten 3.4.2 und 3.4.4.

3.3.2 Die Entwurfsmetapher Arbeitsumgebung

Zeitgleich mit der Arbeit von [Riehle 95] wurde in [Gryczan 95] der Begriff der *Arbeitsumgebung* geprägt, die die konzeptionelle und räumliche Begrenzung eines Arbeitsplatzes modelliert und damit den fachlichen Rahmen für Arbeitsmittel und Arbeitsgegenstände darstellt ([Gryczan 95]). Das Konzept der *Arbeitsumgebung* erlaubt die Unterscheidung von verschiedenen Arbeitsplätzen und bietet damit ein fachliches Basiskonzept zur Entwicklung von Informationssystemen zur Unterstützung kooperativer und qualifizierter Arbeit.

[9] „Ein Entwurfsmuster ist ein Muster, das als Vorlage für die Konstruktion eines softwaretechnischen Entwurfs dient. Ein Entwurfsmuster besteht aus dem Zusammenspiel technischer Elemente wie Klassen, Objekte und Operationen. Die Struktur und Dynamik eines Entwurfsmusters klärt die beteiligten Komponenten, ihre Zusammenarbeit und die Verteilung der Verantwortlichkeiten." ([Riehle 95])

[10] Ein Entwurfsmuster dient als generisches Lösungsschema für ein bestimmtes Entwurfsproblem, welches in einem bestimmten Zusammenhang immer wieder auftritt ([Flor 96]).

Begriff 12 : *Arbeitsumgebung*
„Eine Arbeitsumgebung zur Unterstützung qualifizierter menschlicher Tätigkeiten ist der Ort für eine anwendungsfachlich motivierte Zusammenstellung von Werkzeugen, Automaten und Materialien. Durch die Arbeitsumgebung werden keine Reihenfolgebedingungen für die Verwendung von ... [Arbeitsmitteln und Arbeitsgegenständen] festgelegt." ([Gryczan 95])

Um den konkurrierenden Zugriff auf Materialien kontrollieren zu können, nutzt [Gryczan 95] das fachlich motivierte Konzept von Original und Kopie. Während auf ein Original sowohl sondierend als auch modifizierend zugegriffen werden kann, sind bei einer Kopie ausschließlich sondierende Zugriffe erlaubt. Zu einem bestimmten Zeitpunkt kann maximal eine Arbeitsumgebung im Besitz des Originals eines Materials sein und Änderungen an diesem vornehmen (sequentielle gemeinsame Nutzung). Anderen Arbeitsumgebungen ist es nur möglich, Kopien anzufordern und auf diese ausschließlich sondierend zuzugreifen. Auf diese Weise wird gewährleistet, daß ein Material(original) in einem Informationssystem nur einmal vorhanden und damit die Lokalität des Materials im System eindeutig bestimmbar ist (vgl. Unterabschnitt 2.3.4).

Durch den Einsatz des Konzepts von Original und Kopie kann die modifizierende Bearbeitung von Materialien ausschließlich in einer Arbeitsumgebung durch einen Benutzer stattfinden, so daß eine gleichzeitige modifizierende Bearbeitung eines Materials durch mehrere Benutzer a priori ausgeschlossen wird. Sollte ein Material nicht als Original verfügbar sein, so muß der anfordernde Anwender an den Originalbesitzer über ein geeignetes Kommunikationsmedium (zum Beispiel e-mail) herantreten, um ihn von seinem Wunsch zu unterrichten.

Innerhalb einer Arbeitsumgebung wird der simultane Zugriff auf ein Material durch einen Benachrichtigungsmechanismus geregelt. Auf diese Weise soll nach [Gryczan 95] gewährleistet werden, daß Werkzeuge und Automaten stets eine aktuelle Sicht auf das Material präsentieren. Auf den Einsatz des Konzepts von Original und Kopie wird auf Umgebungsebene bewußt verzichtet, weil der Anwender für seine Tätigkeiten ausreichend qualifiziert ist, um die angebotenen Freiheitsgrade sinnvoll und verantwortlich nutzen zu können. Damit liegt die Verantwortung für mögliche Inkonsistenzen bei gemeinsamer Nutzung von Materialien durch mehrere Werkzeuge bzw. Automaten einer Arbeitsumgebung allein beim qualifizierten Anwender.

Damit kommen der Arbeitsumgebung zwei Grundaufgaben zu:

1. *Versorgung der Werkzeuge und Automaten der Umgebung mit den in der Arbeitsumgebung verwahrten Materialien:* Zu diesem Zweck benutzen die

Werkzeuge und Automaten auf technischer Ebene explizit eine sogenannte Materialverwaltung, die die Schnittstelle zu externen technischen Diensten bereitstellt und für die Beschaffung und Ablage von Materialien sorgt.

2. *Koordinierung des konkurrierenden Zugriffs auf ein Material sowohl aus einer Umgebung als auch aus unterschiedlichen Umgebungen:* Auf technischer Ebene wird zur Koordinierung auf Umgebungsebene ein Ereignisverwalter eingesetzt, der den Benachrichtigungsmechanismus modelliert. Umgebungsübergreifend kommt ein Materialkoordinator zum Einsatz, der für die Realisierung des Konzepts von Original und Kopie sorgt. Dieser Materialkoordinator ist Bestandteil der Materialverwaltung.

Eine detaillierte Betrachtung der Materialverwaltung nach [Gryczan 95] wird in den Unterabschnitten 3.4.2 und 3.4.3 vorgenommen.

3.4 Das Konzept der Materialverwaltung

Zur Zeit gibt es für die Softwareentwicklung nach der Methode WAM kein fachlich motiviertes Konzept zur Versorgung von Werkzeugen und Automaten mit Materialien. Materialien stehen innerhalb einer Arbeitsumgebung entweder nur transient zur Laufzeit zur Verfügung oder müssen von sogenannten Datenbankautomaten bzw. Materialversorgern explizit im persistenten Speicher abgelegt und nach Bedarf in der Anwendung rekonstruiert werden. Diese technisch motivierten Automaten sind in praxisrelevanten Systemen bislang unabdingbar, weil sie die einzige Möglichkeit bieten, den Anschluß an einen persistenten Speicher zu ermöglichen.

Die bestehenden Ansätze zur Verwaltung von Materialien ([Kilberth et al. 93], [Wulf, Weske 94], [Riehle 95] und [Gryczan 95]) und deren Ablage in Datenbanksystemen werden im folgenden vorgestellt und gegeneinander abgegrenzt.

3.4.1 Konzeptionelle Anbindung einer relationalen Datenbank

Um ein konkretes Datenbanksystem vor einem Anwendungsprogramm zu verbergen, wird in [Kilberth et al. 93] ein *Datenbankautomat* beschrieben, der dazu dient, die ihm übergebenen Materialien in einer Datenbank abzulegen, abgelegte Materialien im transienten Speicher zu rekonstruieren und Materialien im persistenten Speicher zu löschen.

Der Datenbankautomat[11] benutzt dabei den Aspekt *speicherbar*, der das Protokoll für die Ablage und Rekonstruktion persistenter Materialien festlegt. Jedes

[11] vgl. Abbildung 3.2

in der Datenbank ablegbare Material erbt von der Aspektklasse *speicherbar* eine abstrakte Schnittstelle. Diese muß konkretisiert werden, indem in jedem Material implementiert wird, wie sich das Material in der Datenbank ablegt (*ErzeugeDatenbankRepraesentation()*).

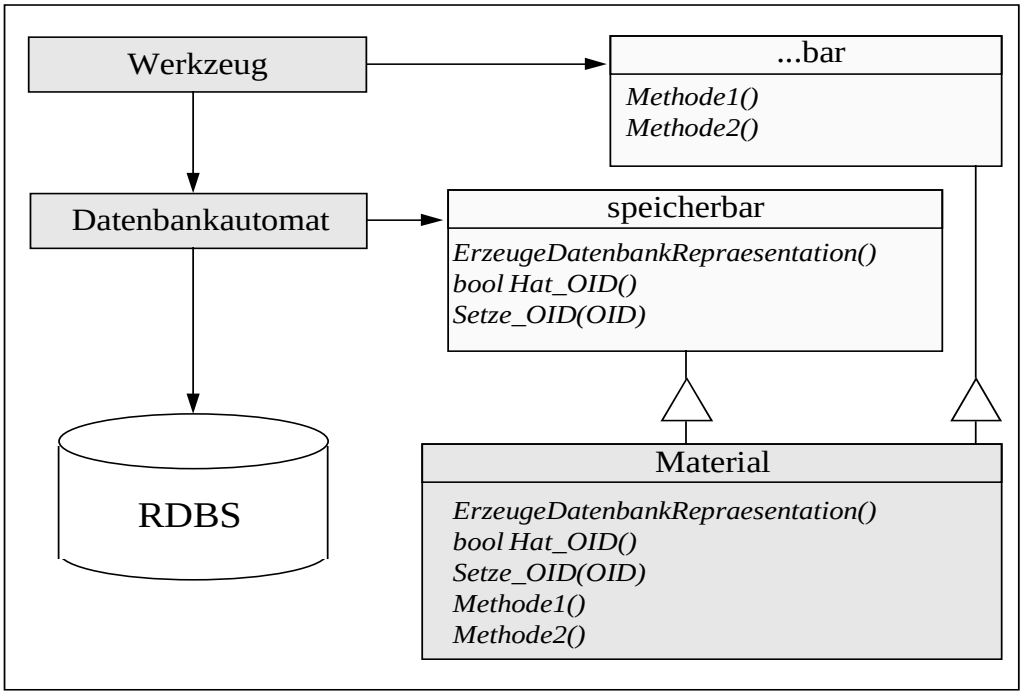


Abbildung 3.2: Einbindung eines Datenbankautomaten

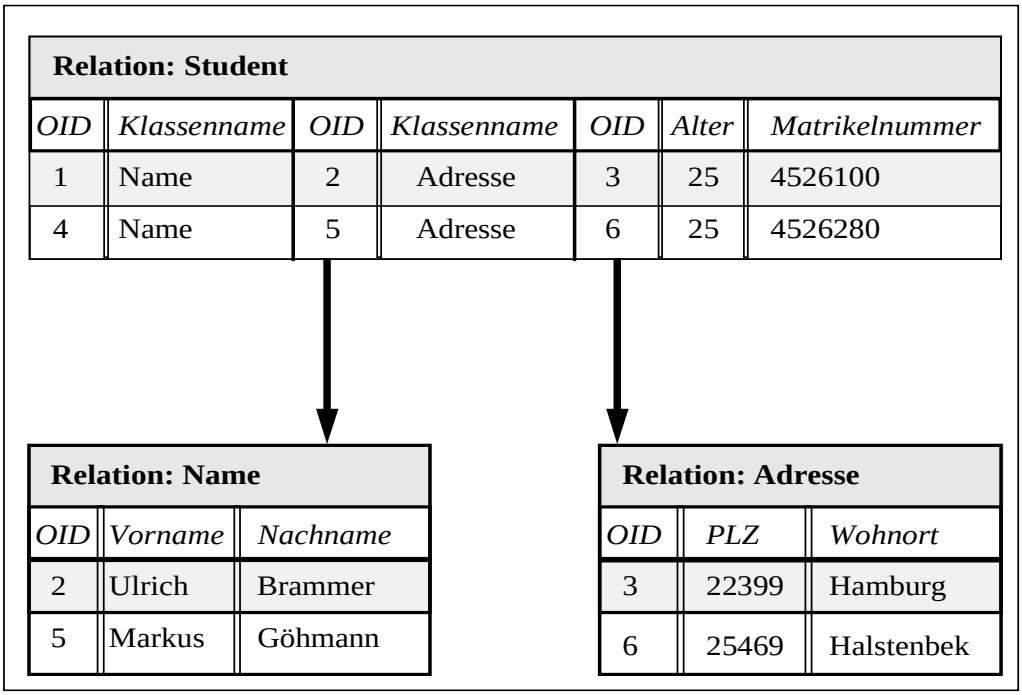
Als Beispiel dient in [Kilberth et al. 93] der Anschluß eines relationalen Datenbanksystems. Für jede speicherbare Materialklasse existiert eine Relation, die Exemplare der Klasse aufnehmen kann.

Bei der Ablage eines Materials[12] in die Datenbank wird ein neues Tupel in die zugehörige Relation eingefügt, das als Primärschlüssel den eindeutigen Materialidentifikator[13] des zu speichernden Materials erhält. Zusätzlich wird das Tupel mit den Attributen des Materials gefüllt. Dabei werden Exemplare von Basistypen (zum Beispiel *int* oder *float*) direkt und Exemplare von Materialklassen als Referenzen, bestehend aus dem Namen der Materialklasse und dem zum Exemplar gehörenden Materialidentifikator, in das Tupel aufgenommen[14]. Bei der Speicherung einer solchen Referenz wird gleichzeitig die Ablage des zugehörigen

[12] vgl. Beispiel in Abbildung 3.3

[13] in Form einer OID, die vom Datenbankautomaten vergeben wird.

[14] Strategien zur Abbildung zwischen Relationen und Objekten werden in Kapitel 5 vorgestellt.



Relation: Student

OID	*Klassenname*	*OID*	*Klassenname*	*OID*	*Alter*	*Matrikelnummer*
1	Name	2	Adresse	3	25	4526100
4	Name	5	Adresse	6	25	4526280

Relation: Name

OID	*Vorname*	*Nachname*
2	Ulrich	Brammer
5	Markus	Göhmann

Relation: Adresse

OID	*PLZ*	*Wohnort*
3	22399	Hamburg
6	25469	Halstenbek

Abbildung 3.3: Relationen zur Speicherung von Studenten

Materialexemplars in die Datenbank angestoßen. Die Ablage von Materialien erfolgt somit durch rekursiven Abstieg, der bis zu den Basistypen durchgeführt wird.

Die Rekonstruktion der Materialien aus der Datenbank erfolgt durch die Umkehrung des beschriebenen Ablaufs. Das Material wird attributweise rekursiv wieder ausgelesen. Dazu ist es erforderlich, daß der Datenbankautomat alle Materialklassen mit den zugehörigen Klassennamen kennt, um Materialien erst erzeugen und anschließend mit Werten belegen zu können (vgl. Erzeugungsmuster in [Gamma et al. 95]). Vernetzte Objektstrukturen sind nach dem Konzept von [Kilberth et al. 93] nicht partiell handhabbar, d.h. es werden stets auch alle referenzierten Objekte in den transienten Speicher transferiert.

3.4.2 Das Materialversorgungskonzept

In [Wulf, Weske 94] wird das Problem der persistenten Haltung von Materialien erneut aufgegriffen und die Anbindung des objektorientierten Datenbanksystems ONTOS als Grundlage für die Beschreibung des Konzepts der *Materialversorgung*[15] genutzt, in welches der Datenbankautomat aus [Kilberth et al. 93] modifiziert als *Materialversorger* integriert wurde.

[15] Die Begriffe Materialversorgung und Materialverwaltung bezeichnen ähnliche Konzepte. Die Begriffsbildung hat sich historisch entwickelt.

Zusätzlich werden Fragen der *Materialverwaltung* innerhalb einer Arbeitsumgebung[16] und die *Materialkoordination* in einem Mehrbenutzersystem beleuchtet. Zu diesem Zweck wird das bereits beschriebene[17] fachlich motivierte Bild von Original und Kopie für den Umgang mit gemeinsamen Materialien genutzt, bei dem nur ein Arbeitsmittel im Besitz des Originals eines Materials sein kann und damit in der Lage ist, Änderungen an diesem vorzunehmen. Wenn das Original eines Materials ausgecheckt[18] ist, dann ist es anderen Anwendern bis zum Zeitpunkt des Einchecken[19] nur möglich, Kopien anzufordern und auf diesen zu arbeiten.

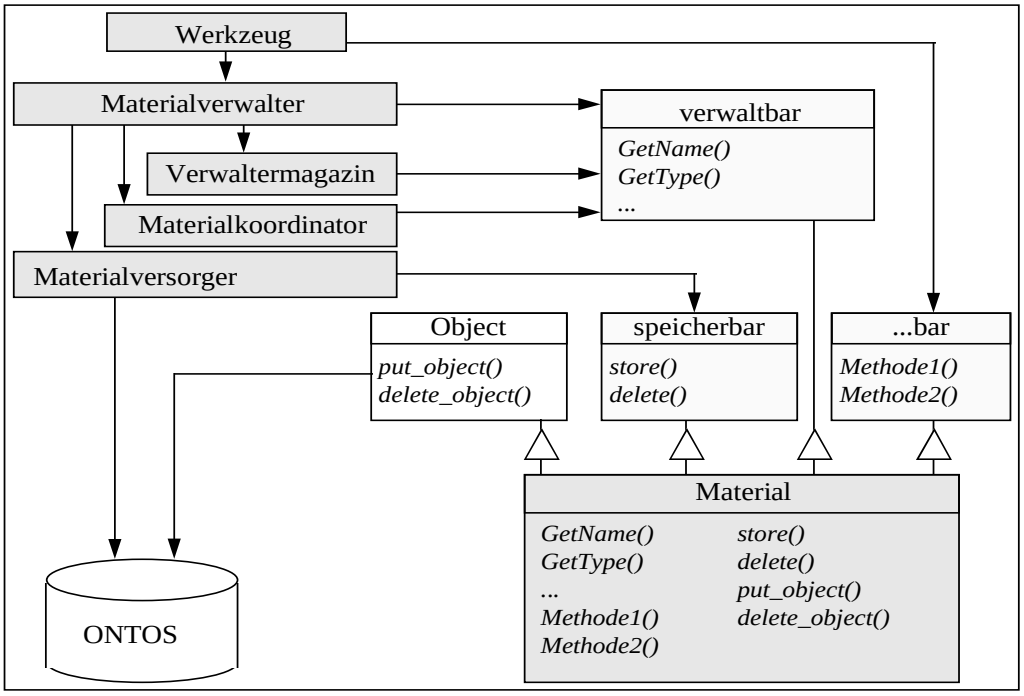


Abbildung 3.4: Die Komponenten des Materialversorgungskonzepts

Das in [Wulf, Weske 94] vorgestellte fachliche Konzept für die Materialversorgung[20] besteht aus den im folgenden beschriebenen Komponenten.

Der Materialverwalter

Der *Materialverwalter* ist ein Automat, der in jeder Arbeitsumgebung genau einmal existiert und zur Beschaffung, Verwahrung und Bereitstellung von Mate-

[16] [Wulf, Weske 94] spricht von Werkzeugumgebung

[17] vgl. Unterabschnitt 3.3.2

[18] Datenbankvokabular: Ein Datum wird einer Datenbank entnommen und als ausgecheckt markiert.

[19] Datenbankvokabular: Ein Datum wird einer Datenbank übergeben. Eine evtl. Markierung wird gelöscht.

[20] vgl. Abbildung 3.4

rialien innerhalb der Arbeitsumgebung dient. Damit der Materialverwalter ein Material verwalten kann, muß es die durch den Aspekt *verwaltbar* spezifizierte Schnittstelle aufweisen.

Der Materialverwalter benutzt ein oder mehrere *Verwaltermagazine*, um die Materialien der lokalen Arbeitsumgebung effizient verwahren zu können. Diese Verwaltermagazine sind Behälter, die Referenzen der in der Arbeitsumgebung verwalteten Materialien enthalten.

Verwaltbare Materialien lassen sich nach ihrem Namen und Typ befragen, so daß durch Angabe dieser Informationen Materialien gesucht und gefunden werden können. Zu diesem Zweck bietet der Materialverwalter für die Formulierung von Anfragen durch die Werkzeuge der lokalen Arbeitsumgebung eine rudimentäre „Spezifikationssprache". Die Anfragen werden gegen die Verwaltermagazine gehalten und ausgewertet. Dabei werden Referenzen auf Kopien der Materialien, die die Anfrage erfüllen, in einem Behälter aufgenommen und dieser dem Werkzeug zur Verfügung gestellt.

Um angeforderte Materialien, die sich nicht in der lokalen Arbeitsumgebung befinden, dem Werkzeug zur Verfügung stellen zu können, bedient sich der Materialverwalter der am Rande der Arbeitsumgebung liegenden Datenbankautomaten, die in [Wulf, Weske 94] als Materialversorger bezeichnet werden.

Da im Gegensatz zu [Gryczan 95] in [Wulf, Weske 94] auch auf Arbeitsumgebungsebene das fachliche Bild von Original und Kopie zum Tragen kommt, kann ein Material nur von einem Arbeitsmittel zur Zeit als Original bearbeitet werden. Zu diesem Zweck wird im Verwaltermagazin auch abgelegt, welches Arbeitsmittel im Besitz des Originals ist.

Der Materialversorger

Der Materialversorger ist ein Automat, der wie der Datenbankautomat das Wissen um die konkrete Schnittstelle des persistenten Speichers kapselt.

Der Materialversorger dient als ein Materiallieferant, der nach außen eine Grundfunktionalität zur Verfügung stellt, mit deren Hilfe Materialien in den persistenten Speicher überführt, aus diesem rekonstruiert und aus dem persistenten Speicher gelöscht werden können. Dazu müssen alle Materialien, mit denen der Materialversorger umgeht, die abstrakte Schnittstelle des rein technischen Aspekts *speicherbar* und eine konkrete Implementation dieser aufweisen (vgl. Unterabschnitt 3.4.1).

Die Anzahl der einer Arbeitsumgebung zur Verfügung stehenden Materialversor-

ger ist prinzipiell beliebig, so daß auch mehrere verschiedene persistente Speicher angebunden werden können. Um diese nutzen zu können, besitzt der Materialverwalter eine Versorgertabelle, mit deren Hilfe er ein Material dem zugehörigen Versorger zuordnen kann. In [Wulf, Weske 94] wird zur Vereinfachung festgelegt, daß ein Materialtyp genau einem Materialversorger zugeordnet wird, der Materialien verschiedener Materialtypen aufnehmen kann.

Angestoßen wird der Materialversorger ausschließlich durch den Materialverwalter, so daß der Anwender nicht mehr direkt mit einem Speicher- bzw. Ladewerkzeug, wie in vielen Anwendungen üblich, umgehen muß[21], sondern nur noch mit dem Materialverwalter kommuniziert, der den im Hintergrund aktiven Materialversorger zur Aktivierung und Passivierung der gewünschten Materialien benutzt.

Der Materialkoordinator

Der Automat *Materialkoordinator* wird nicht wie die bisher vorgestellten Komponenten lokal in einer Arbeitsumgebung, sondern nur einmal in einer sogenannten Administrationsumgebung[22] erzeugt, so daß jeder lokalen Arbeitsumgebung lediglich ein Materialkoordinator-Proxy[23] zur Verfügung steht. Deshalb wird auch von dem *zentralen Materialkoordinator* gesprochen.

Der Materialkoordinator soll nicht wie der Materialversorger den persistenten Speicher verbergen oder wie der Materialverwalter zur Beschaffung, Verwahrung und Bereitstellung dienen, sondern die fachlich motivierte Zugriffskontrolle nach dem Bild von Original und Kopie zwischen mehreren konkurrierenden Arbeitsumgebungen ermöglichen.

3.4.3 Das Konzept der Materialverwaltung

Das in [Gryczan 95] dargestellte Konzept der Materialverwaltung[24] entspricht im wesentlichen dem Materialversorgungskonzept aus [Wulf, Weske 94].

Beim Vergleich des Konzepts der Materialverwaltung ([Gryczan 95]) mit dem Materialversorgungskonzept ([Wulf, Weske 94]) finden sich drei Unterschiede:

1. *Ein Materialmagazin statt vieler Verwaltermagazine:*
   Die Möglichkeit mehrere Verwaltermagazine innerhalb einer Arbeitsumgebung einzusetzen, wird in [Gryczan 95] nicht länger unterstützt. Außerdem

[21] vgl. hierzu den Datenbankautomat nach [Kilberth et al. 93]

[22] Eine Administrationsumgebung ist eine in sich abgeschlossene Umgebung, die sich in mehrere Arbeitsumgebungen auf unterschiedlichen Rechnerknoten eines Netzwerks aufteilen kann.

[23] Ein Proxyobjekt dient als ein lokaler Platzhalter eines Objekts, das sich in einem anderen Adreßraum befindet (vgl. [Gamma et al. 95]).

[24] vgl. Abbildung 3.5

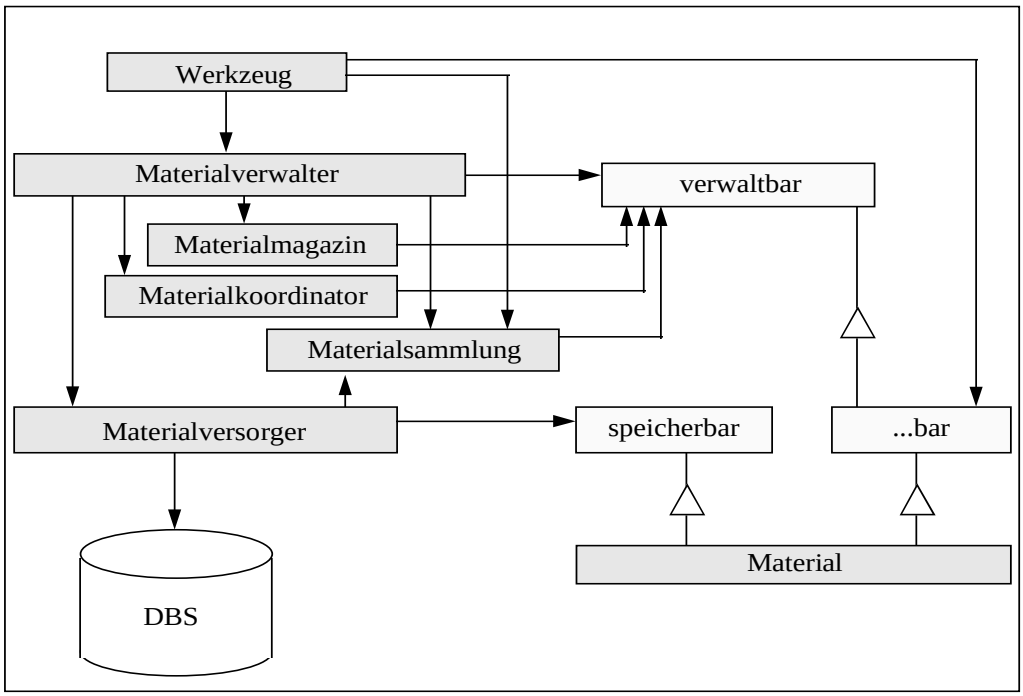


Abbildung 3.5: Die Komponenten der Materialverwaltung

erfolgt eine Umbenennung des Verwaltermagazins in *Materialmagazin*. Diesen Begriff werden wir für weitere Betrachtungen übernehmen.

2. *Keine Zugriffskontrolle innerhalb der lokalen Arbeitsumgebung nach dem Bild von Original und Kopie:*
   Die in [Wulf, Weske 94] geforderte Zugriffskontrolle auf Arbeitsumgebungsebene wird in [Gryczan 95] nicht realisiert (vgl. Unterabschnitt 3.3.2). Durch diesen Verzicht entfällt die Aufgabe des Materialmagazins, Zugriffe von Arbeitsmitteln auf Materialien zu koordinieren. Stattdessen wird der simultane Zugriff von mehreren Werkzeugen einer Arbeitsumgebung auf ein Material durch einen Benachrichtigungsmechanismus geregelt. Dieser ist auf der Basis eines Ereignisverwalters realisiert, bei dem sich die Werkzeuge der Arbeitsumgebung registrieren lassen können, um über alle Änderungen an einem bestimmten Material informiert zu werden.

3. *Materialsammlung als Ergebnis von Anfragen:*
   Das Ergebnis einer Anfrage eines Werkzeugs an den Materialverwalter ist in [Gryczan 95] ein Behälter, der alle Materialien enthält, die der Anfrage genügen. Dieser Behälter wird *Materialsammlung* genannt und dient nach Angaben von Guido Gryczan[25] auch als Ergebnis von Anfragen des Materialverwalters an einen Materialversorger.

[25] Arbeitsbereich SWT, Fachbereich Informatik, Universität Hamburg

3.4.4 Das Entwurfsmuster Materialverwaltung

Auch das in [Riehle 95] beschriebene Entwurfsmuster Materialverwaltung[26] basiert auf dem oben beschriebenen Materialversorgungskonzept ([Wulf, Weske 94]). Es finden sich jedoch Unterschiede und interessante Erweiterungen des Materialversorgungskonzepts.

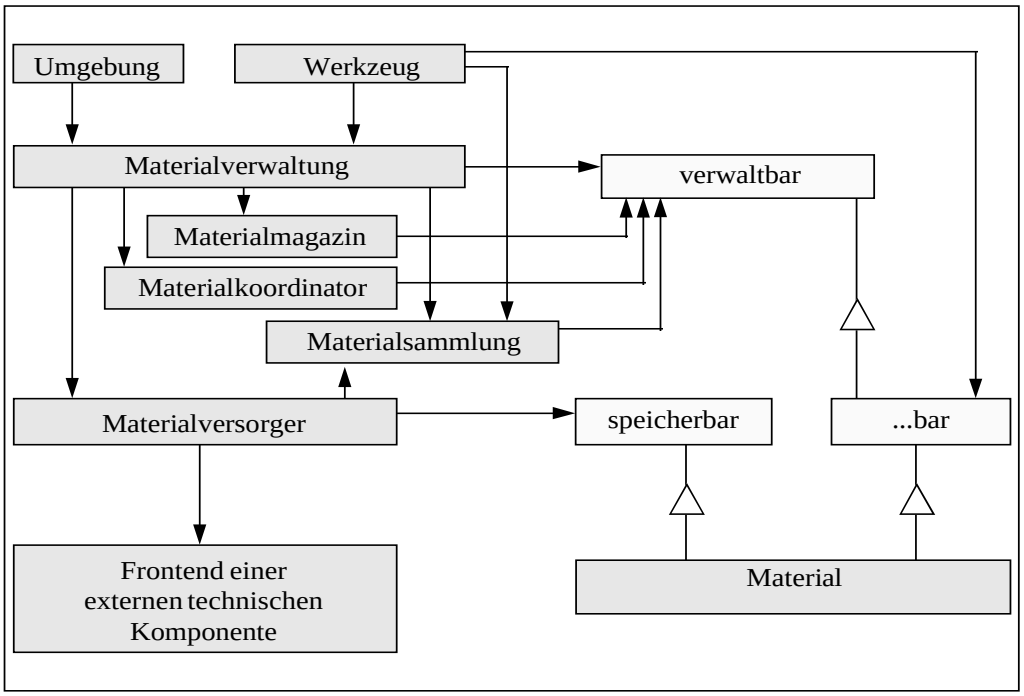


Abbildung 3.6: Die Komponenten des Entwurfsmusters Materialverwaltung

[Riehle 95] motiviert die Materialverwaltung als objektorientierte Anfrageschnittstelle für die Materialversorgung aus externen technischen Komponenten. Diese Schnittstelle soll alle externen technischen Komponenten (nicht nur Datenbanksysteme) gleich behandeln, so daß die Inanspruchnahme einer spezifischen Komponente nach außen gekapselt wird. In diesem Rahmen stellt [Riehle 95] Anforderungen an die einfache Umsetzung eines Datenbankzugriffs im Rahmen der Metapher ,,Umgang mit Werkzeug und Material":

- ,,Materialien müssen ... persistent gehalten und wiedergefunden werden können.
- Materialien müssen über eine ausreichend mächtige Anfrageschnittstelle spezifiziert und aktiviert werden.

[26] vgl. Abbildung 3.6

- Die Anfrageschnittstelle muß möglichst einheitlich sein und, sofern nicht anders gewünscht, die Delegation der Aufgabe an verschiedene .. [technische Komponenten] verstecken.

- Die unterschiedlichen Dienste [der technischen Komponenten] müssen einheitlich genutzt werden können. Relationale, objektorientierte oder hostbasierte hierarchische Datenbanken müssen zugreifbar sein."

Beim Vergleich der Materialverwaltung ([Riehle 95]) mit dem Materialversorgungskonzept ([Wulf, Weske 94] findet sich eine Reihe von Unterschieden:

- Es erfolgt eine Umbenennung des Materialverwalters in Materialverwaltung, die wir für die folgende Diskussion nicht übernehmen wollen. Stattdessen lehnen wir uns an die Begriffsbildung von [Gryczan 95] an.

- Wie [Gryczan 95] erweitert auch [Riehle 95] das Materialversorgungskonzept um das Konzept der Materialsammlung, die alle Materialien enthält, die einer Anfrage an den Materialverwalter genügen. Zudem wird neben der üblichen Nutzung des Materialverwalters durch Werkzeuge auch seine Benutzung durch andere Komponenten erlaubt, wie zum Beispiel der oben beschriebenen Umgebung.

- Im Gegensatz zu [Gryczan 95] hält [Riehle 95] an der Zugriffskontrolle auf Umgebungsebene nach dem Konzept von Original und Kopie fest, wobei [Riehle 95] nicht auf die Prevention umgebungsübergreifender konkurrierender Zugriffe eingeht.

- In [Riehle 95] dienen Materialversorger zur Kapselung verschiedener technischer Dienste und damit nicht nur der Dienste eines Datenbanksystems eines bestimmten Paradigmas. In [Riehle 95] wird somit zum ersten Mal die Forderung nach der generellen Anschließbarkeit paradigmatisch unterschiedlicher Datenbanksysteme erhoben.

Die Nutzung mehrerer Materialversorger für beliebige externe Dienste würde nach dem in [Wulf, Weske 94] beschriebenen Materialversorgungskonzept für jeden einzelnen Materialversorger einen eigenen Aspekt (vgl. *speicherbar*) erfordern. Um dieses Problem zu umgehen, wäre es sinnvoll, ein einheitliches Protokoll zur Fragmentierung und Defragmentierung von Materialien bereitzustellen[27], um auf diese Weise Transformationen zwischen unterschiedlichen Paradigmen[28] bzw. unterschiedlichen Darstellungsformen eines Paradigmas[29] zu ermöglichen. Ein solches

[27] vgl. hierzu die ausführliche Diskussion in [Mathiske 96b] über die Linearisierung von Datenrepräsentationen in verteilten Systemen.

[28] zum Beispiel Abbildung von Objekten auf Relationen und umgekehrt

[29] zum Beispiel Abbildung eines C++-Objekts auf ein Objekt in einer objektorientierten Datenbank und umgekehrt

Protokoll wird in [Riehle et al. 96] beschrieben:

> ,,Application classes should have no knowledge about the external representation format which is used to represent their instances. Otherwise, introducing a new representation format or changing an old one requires to change almost every class in the whole system. These classes should contain no representation specific code for reading or writing their instances."([Riehle et al. 96])

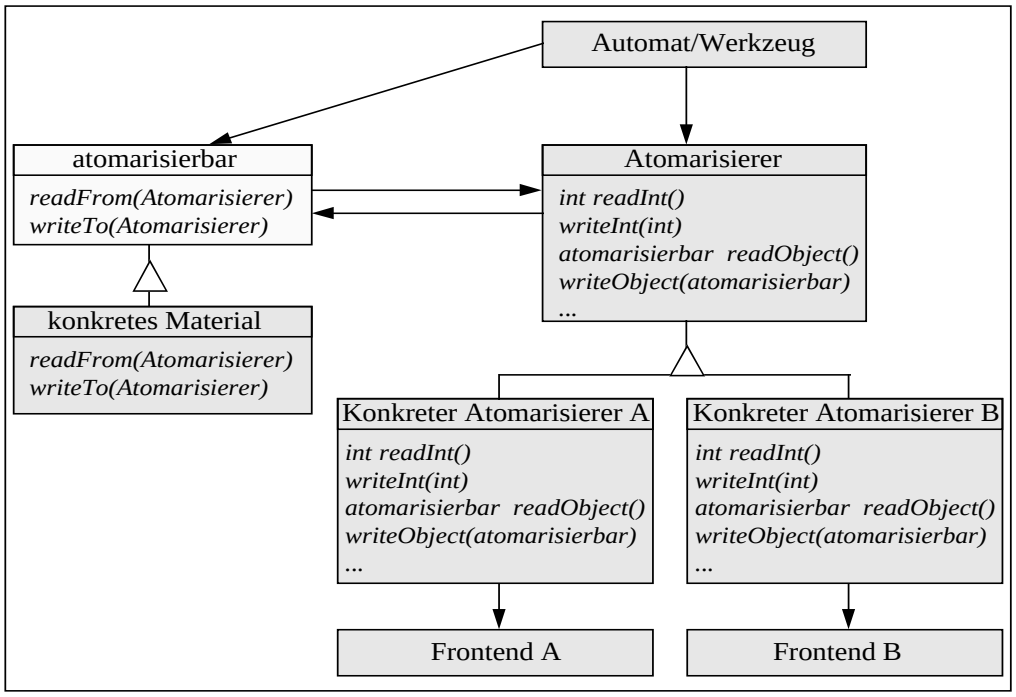


Abbildung 3.7: Die Struktur des Entwurfsmusters Atomarisierer

Das Entwurfsmuster Atomizer[30] [31] ([Riehle et al. 96]) dient zur effizienten Transformation von Objekten in beliebige Datenstrukturen und ermöglicht die Re-Transformation von Objekten aus diesen Datenstrukturen. Das Muster basiert auf dem in [Gamma et al. 95] beschriebenen Besucher-Muster[32], das wie das Atomizer-Muster aus fünf Teilnehmern besteht:

[30] Dieses Muster kommt teilweise modifiziert in vielen gängigen Frameworks (zum Beispiel ET++), in persistenten Programmiersprachen (zum Beispiel Arjuna) und anderen programmiersprachlichen Erweiterungen (zum Beispiel CORBA) zum Einsatz.

[31] Atomizer engl. (Atomarisierer, Zerstäuber)

[32] Zweck des Besuchermusters: ,,Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es .. , eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern."([Gamma et al. 95])

1. Der *Atomarisierer* kapselt bestehende externe technische Dienste und stellt abstrakte Protokolle zum Lesen und Schreiben *atomarisierbarer* Elemente zur Verfügung. Zu diesem Zweck bietet der *Atomarisierer* Lese- und Schreiboperationen für Exemplare von *atomarisierbaren* Materialien (*readObject()* und *writeObject()*) und für Exemplare von Basistypen (zum Beispiel *readInt()* und *writeInt()*).

2. Ein *konkreter Atomarisierer* ist Unterklasse von *Atomarisierer* und konkretisiert die geerbten abstrakten Protokolle zum Lesen und Schreiben *atomarisierbarer* Elemente. Zudem kapselt ein *konkreter Atomarisierer* das Frontend einer spezifischen externen technischen Komponente.

3. Die Klasse *atomarisierbar* ist abstrakt und spezifiziert die Schnittstelle zum Lesen und Schreiben *atomarisierbarer* Materialien (*readFrom()* und *writeTo()*). Daneben bietet sie eine Operation zum Erzeugen von Exemplaren konkreter Materialien (*newByName()*).

4. Ein *konkretes Material* ist Unterklasse von der Klasse *atomarisierbar* und konkretisiert die geerbte Schnittstelle zum Lesen und Schreiben, um zum einen den internen Zustand an den *Atomarisierer* zu übergeben und zum anderen wieder entgegenzunehmen.

5. Das *Frontend* einer spezifischen externen technischen Komponente wird ausschließlich vom *konkreten Atomarisierer* benutzt.

Durch den Einsatz des Entwurfsmusters Atomizer ist es möglich, die Materialklassen frei von Wissen über das Repräsentationsformat der externen technischen Komponente zu halten. Die technischen Details des Lesens und Schreibens werden an externe und austauschbare Klassen (konkrete Atomarisierer) delegiert.

3.5 Kritische Würdigung

Im Rahmen der Einführung einer expliziten fachlichen und technischen Sicht auf Informationssysteme ist es unserer Ansicht nach erforderlich, eine Identifikation und anschließende klare Trennung der fachlichen und technischen Komponenten vorzunehmen. Durch diese Trennung wird es möglich, eine neue Abstraktionsebene einzuführen, die die abstrakte Behandlung der technischen Komponenten durch die fachliche Welt ermöglicht. Auf diese Weise kann die Korrelation der beiden Welten minimiert bzw. ganz aufgehoben und damit der Anwendungsentwickler bei der Gestaltung fachlich spezifischer Anwendungen für kooperierende qualifizierte menschliche Tätigkeit unterstützt werden.

Begriff 13 : *Subsystem*
In einem Subsystem „werden Gruppen von Klassen, aber auch weitere Subsysteme zusammengefaßt mit dem Ziel, eine wohldefinierte Menge von Dienstleistungen anzubieten. Von außen betrachtet, repräsentiert ein Subsystem eine ebensolche konzeptionelle Einheit wie eine Klasse- nur auf einer 'höheren Ebene'." ([Kilberth et al. 93])

In Anlehnung an die in Kapitel 2 vorgestellte 3-Schicht-Architektur halten wir es für sinnvoll, die fachlichen und die technischen Komponenten durch ein *Subsystem*, das die technisch transparente Nutzung der technischen Komponenten ermöglicht, voneinander zu entkoppeln. Ein solches Subsystem kann als Adaptionsschicht (*gateway*) zwischen der fachlichen und der technischen Welt verstanden werden (vgl. Abbildung 3.8).

Die durch das Subsystem bereitgestellte Schnittstelle sollte nicht nur von den technischen Details abstrahieren, sondern gleichzeitig der fachlichen Welt eine wohldefinierte Menge fachlicher Basisdienste bereitstellen, die in jeder fachlichen Anwendungsdomäne und damit in einem beliebigen fachlichen Kontext vorzufinden sind (fachlich generelle Schnittstelle).

Die fachlich generelle Schnittstelle des Subsystems bildet aus dem fachlichen Blickwinkel betrachtet einen unteren Abschluß und könnte aufgrund ihrer fachlich generellen Verwendbarkeit in einer maximalen Anzahl fachlich spezifischer Kontexte eingesetzt werden. Dabei ist allerdings zu beachten, daß die durch das Subsystem bereitgestellte Schnittstelle möglichst intuitiv und nicht zu abstrakt sein sollte.

Auch wenn die Aufteilung von Softwaresystemen in unterschiedliche Schichten nach [Kilberth et al. 93] problematisch ist, weil sie nicht den objektorientierten Architekturprinzipien entspricht, ist [Booch 94] der Auffassung, daß Klassen, Objekte und Module nur unzulängliche Mittel der Abstraktion darstellen. Viele komplexe Systeme sind daher hierarchisch in Schichten zu organisieren, die „unterschiedliche Abstraktionsebenen dar[stellen], die aufeinander aufbauen, und die doch jede für sich betrachtet werden können. Auf jeder Abstraktionsebene finden [sich] .. wichtige Ansammlungen von Objekten, die zusammenarbeiten, um ein höhergeordnetes Verhalten zu realisieren." ([Booch 94])

Die durch das Subsystem bereitgestellte Abstraktion von den technischen Details ist ein wichtiges Mittel zur Komplexitätsreduktion, weil sich die nach wie vor erforderliche (programmier-)technische Realisierung fachlich spezifischer Konzepte wesentlich vereinfacht. Der Grund dafür liegt in der bereitgestellten technischen Transparenz, die eine abstrakte Behandlung der technischen Komponenten

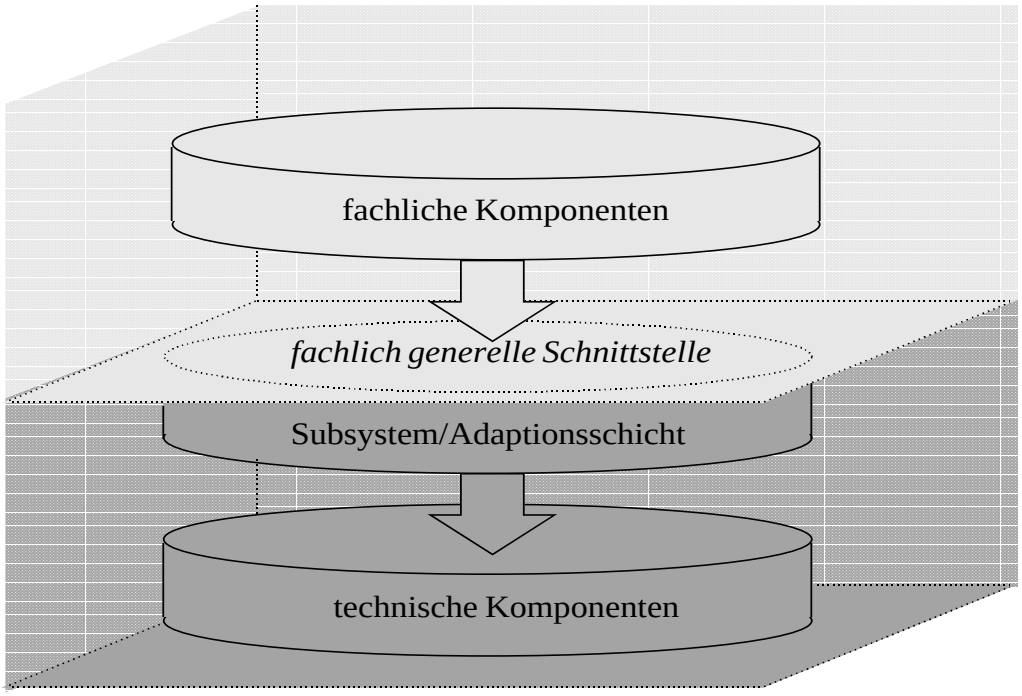


Abbildung 3.8: Die Adaptionsschicht zur Entkopplung fachlicher und technischer Komponenten

durch fachliche Komponenten ermöglicht und damit das Durchschlagen technischer Konzepte auf die fachliche Modellierung verhindert. Stellt das Subsystem zudem eine fachlich generelle Schnittstelle bereit, so können die bereitgestellten Dienstleistungen als fachliche Basisdienste in die Anwendungsentwicklung nach der Methode WAM einfließen. Im folgenden wollen wir deshalb die oben vorgestellten Konzepte auf ihre fachlich generelle Verwendbarkeit untersuchen.

3.5.1 Der Umgebungsbegriff

Im Gegensatz zu [Gryczan 95], der die Arbeitsumgebung auf einer konzeptionellen Ebene betrachtet, stellt [Riehle 95] die musterbasierte technische Umsetzung seines fachlich spezifischen Umgebungsmusters in den Vordergrund. Wir wollen uns daher im folgenden an dem allgemeineren Begriff der Arbeitsumgebung von [Gryczan 95] orientieren.

Es ist festzuhalten, daß die in [Gryczan 95] gegebene Definition der Arbeitsumgebung nicht vollständig ist: So macht sie zum Beispiel keine Aussage über die Anzahl der Anwender einer Arbeitsumgebung, obwohl in den Erläuterungen zu der Definition stets davon ausgegangen wird, daß eine 1:1 Zuordnungsbeziehung zwischen Anwendern und Arbeitsumgebungen besteht.

Eine tiefergehende Betrachtung des Umgebungsbegriffs ist Thema einer zu dieser

Arbeit parallel entstehenden Diplomarbeit[33], die sich mit dem Umgebungsbegriff im Rahmen einer sogenannten ,,Raummetapher" beschäftigt. Aus diesem Grund wollen wir uns bei der kritischen Würdigung ausschließlich auf die Bewertung der Zugriffskontrolle nach Original und Kopie sowie auf eine Analyse der Konzepte zur Materialverwaltung beschränken.

3.5.2 Das Konzept von Original und Kopie

Paal Sörgaard, der sich in [Sörgaard 88] mit der Realisierung von gleichzeitig gemeinsam genutzten Materialien (*shared materials*) beschäftigt, glaubt, daß durch die fehlende Unterstützung einer gleichzeitig gemeinsamen Nutzung von Materialien der Aspekt der Kooperation bei der Arbeit vernachläßigt bzw. unterschätzt wird[34], weil gleichzeitig gemeinsam genutzte Materialien ein wichtiges Medium zur Koordination kooperativer Arbeit darstellen: ,,When a table is carried [by two people], however, the two people can follow each other's actions because the actions get mediated through the shared material. " Diese Betrachtungen führen zur WYSIWIS-Metapher (**W**hat **Y**ou **S**ee **I**s **W**hat **I S**ee), bei der die Anwender den Eindruck erhalten, auf einer gemeinsamen Oberfläche zu arbeiten[35]. Jeder Anwender sieht damit die Cursor-Bewegungen der anderen Anwender, die das Material gleichzeitig sondieren und modifizieren können, so daß Zugriffskollisionen auf dieser Ebene erkannt und potentielle Konflikte stets als natürliche Eigenschaft des Materials[36] betrachtet werden können ([Sörgaard 88]).

Da ein WYSIWIS-System zur Laufzeit jede Interaktion eines Anwenders an alle anderen Anwender des Systems in Echtzeit propagieren muß und die zugrundeliegenden technischen Komponenten selten in der Lage sind, dies zu leisten, relativiert Sörgaard seinen Ansatz, indem er eine Abbildung gleichzeitig gemeinsamer Nutzung in der Arbeitsdomäne auf quasi-gleichzeitige gemeinsame Nutzung im Softwaresystem zuläßt: Alle Anwender haben die Sicht auf das Material, wobei einzelne Teile des Materials exklusiv bearbeitet werden und diese den anderen als blockiert sichtbar gemacht werden. Auf diese Weise erfolgt eine implizite Zugriffssynchronisation durch die Anwender des WYSIWIS-Systems. ,,The grain size of the system, the size of the object allocated to the single user for update, should of course be small enough to allow the needed degree of concurrency. At the same time it should be as large as possible to allow maximal flexibility." ([Sörgaard 88])

[33] Henning Wolf und Stefan Roock, Fachbereich Informatik an der Universität Hamburg, Arbeitsbereich Softwaretechnik

[34] ,,The properties of the shared material are, however, often ignored when work is computerised. ... This has resulted in very inflexible systems."([Sörgaard 88])

[35] ,,impression of working on a shared surface" ([Sörgaard 88])

[36] ,,natural property of the material" ([Sörgaard 88])

Da Sörgaards Ansatz wie die Methode WAM dazu dient, die im Anwendungsbereich vorgefundenen Arbeitsformen möglichst ohne semantischen Strukturbruch in ein ablauffähiges Softwaresystem zu übertragen ([Kilberth et al. 93]), schließen wir uns der Forderung Sörgaards nach einer Unterstützung gleichzeitiger oder zumindest einer quasi-gleichzeitigen gemeinsamen Nutzung von Materialien an. Dazu werden, wie wir im folgenden zeigen wollen, umfassendere Konzepte benötigt, die das Konzept von Original und Kopie nicht zur Verfügung zu stellen vermag.

Eine Zugriffskontrolle nach dem Bild von Original und Kopie erlaubt ausschließlich dem Besitzer eines Materialoriginals, auf alle Bestandteile des Materials modifizierend zuzugreifen. Die Besitzer von Kopien können diese ausschließlich sondieren und sind damit nicht in der Lage, Änderungen an dem Material vorzunehmen. Die Zugriffskontrolle nach dem Bild von Original und Kopie bietet damit kein Konzept für die gleichzeitig gemeinsame Nutzung von Materialien und erlaubt neben der sequentiellen gemeinsamen Nutzung lediglich eine sehr eingeschränkte Variante quasi-gleichzeitiger gemeinsamer Nutzung von Arbeitsgegenständen, weil gleichzeitige modifizierende Zugriffe auf unterschiedliche Teile des Materials nicht möglich sind.

Festzuhalten ist zudem, daß eine Kopie nicht den aktuellen Zustand eines Materials widerspiegelt, sondern es sich bei einer Kopie um einen auf Anforderung bereitgestellten und nicht aktualisierten Snapshot des Materialoriginals handelt. Weil der Zustand der Kopie nicht synchron mit dem des Materialoriginals geändert wird und es für den Kopiebesitzer nicht ersichtlich ist, wann der Zustand der Kopie nicht mehr dem aktuellen Stand eines Materialoriginals entspricht, kann eine Kopie nur sehr eingeschränkt als Stellvertreter für ein Materialoriginal fungieren. Es besteht stets die Gefahr, daß sondierende Zugriffe auf Materialkopien nicht mehr aktuelle Resultate liefern.

Auch die Idee, eine erweiterte Unterstützung der quasi-gleichzeitigen gemeinsamen Nutzung von Arbeitsgegenständen auf der Basis des Konzepts von Original und Kopie zu simulieren, ist problematisch, weil erstens durch Anforderung eines Originals immer das gesamte Material für einen einzigen Anwender exklusiv gesperrt wird und zweitens die Bereitstellung einer stets aktuellen Sicht auf ein Material auf Basis ununterbrochener Anforderung (*polling*) von Kopien schwierig ist. Das hat im wesentlichen zwei Gründe: Erstens ist der Zeitpunkt einer durchgeführten Materialmodifikation für den anfordernden Anwender der Kopien (Sichtinhaber) nicht erkennbar und zweitens muß die Kopie explizit angefordert werden, während eine Sicht implizit ohne Aufforderung aktualisiert wird.

Ein weiteres Problem stellt der Kopiebegriff dar. Es ist festzustellen, daß der in [Wulf, Weske 94] geprägte Kopiebegriff (Original und Kopie) verwirrend gewählt

worden ist, weil er im krassen Widerspruch zu dem in [Wulf, Weske 94] als Motivation herangezogenen Begriff der Kopie aus unserem täglichen Leben (Abschrift, Zweitschrift, Durchschlag, Durchschrift oder Fotokopie) steht. Diese Kopien gleichen zum Entstehungszeitpunkt den zugehörigen Originalen, haben aber eine eigene Identität und sind deshalb auch prinzipiell veränderbar. Somit können sie wieder als eigenständige Originale betrachtet werden. Eine nach [Wulf, Weske 94] erzeugte Kopie hingegen hat weder eine eigene Identität noch ist sie modifizierbar.

Damit können wir festhalten, daß sich das Konzept von Original und Kopie als nicht ausreichend erweist, weil es auf der einen Seite neben der sequentiellen gemeinsamen Nutzung lediglich eine sehr eingeschränkte Variante der quasi-gleichzeitigen gemeinsamen Nutzung von Materialien unterstützt und zum anderen in der Wahl der Begriffe unpräzise gefaßt ist. Vielmehr könnte man meinen, daß es sich bei dem Konzept von Original und Kopie um den Versuch einer fachlichen Interpretation der technischen Schreib- und Lesesperren von Datenbanksystemen handelt[37], weil dieses Konzept zum wechselseitigen Ausschluß beim Zugriff auf Datenbanken[38] verwendet wird.

Obwohl das Konzept von Original und Kopie bereits für verschiedene fachliche Kontexte gedacht ist, muß es konzeptionell erweitert und in der Begriffsbildung präzisiert werden. Andernfalls kann es nicht als fachliches Basiskonzept zum Einsatz kommen.

3.5.3 Das Konzept der Materialverwaltung

In diesem Unterabschnitt wollen wir die Unzulänglichkeiten der Konzepte und Komponenten der Materialverwaltung zusammenfassend darstellen. Dabei gehen wir neben der fehlenden Trennung fachlicher und technischer Anteile insbesondere auf die mangelnde praktische Anwendbarkeit der Materialverwaltung ein.

Die in [Wulf, Weske 94] beschriebene Materialverwaltung realisiert das Konzept der Identität von Materialien auf der Basis fachlicher Materialnamen. Damit haben ausschließlich benannte Materialien eine Identität und die Umbenennung eines Materials impliziert die Änderung seiner Identität. Der Aspekt *verwaltbar* ermöglicht es, daß alle verwaltbaren Materialien nach ihrem Namen befragt werden können[39], so daß die fachlichen Materialnamen zur Lokalisierung von Materialrepräsentationen sowohl im transienten (Materialmagazin) als auch im persistenten Speicher (Materialversorger) eingesetzt werden können. Dabei werden die fachlichen Materialnamen direkt an technische Speicheradressen gebunden,

[37] vgl. Unterabschnitt 4.2.1

[38] vgl. Materialversorgungskonzept in Unterabschnitt 3.4.2

[39] In diesem Zusammenhang ist anzumerken, daß die Aspektklasse *verwaltbar* lediglich eine Schnittstelle festlegt, die in jeder Materialklasse zu konkretisieren ist.

wodurch es zu einer Korrelation fachlicher und technischer Konzepte kommt.

Der Materialverwalter, der als unterer Abschluß der fachlichen Welt dient, stellt der fachlichen Welt einen CheckIn/CheckOut-Mechanismus zur Verfügung, mit dessen Hilfe Materialien aus technischen Komponenten beschafft werden können. Dieser Materialbeschaffung aus technischen Diensten kann kein fachlicher Vorgang zugeordnet werden, so daß der CheckIn/CheckOut-Mechanismus rein technisch geprägt ist und damit im fachlichen Modell nicht sichtbar sein sollte. Zudem ist es unserer Ansicht nach fraglich, ob die Interaktion eines Werkzeugs mit einem Materialverwalter aus einem fachlichen Blickwinkel überhaupt motivierbar ist. So ist es beispielsweise nicht der Hammer, der dem Materialkasten einen Nagel entnimmt, sondern der Anwender, der sich sowohl das Werkzeug Hammer als auch das Material Nagel aus dem Kasten beschafft. In diesem Zusammenhang stellt sich die grundsätzliche Frage, ob zuerst das Werkzeug und dann ein dazu passendes Material oder in der Umkehr erst das Material und dann ein dazu passendes Werkzeug selektiert wird. Beide Sichtweisen sind gleichberechtigt, wie anhand des gebrachten Beispiels leicht zu zeigen wäre.

Der Materialversorger[40] dient lediglich zur Transformation zwischen technischen Repräsentationen von Materialien. Dabei werden transiente in persistente und persistente in transiente Speicherrepräsentationen transformiert. Da sich Materialien einer Arbeitsdomäne in der Regel nicht verändern, wenn sie abgelegt werden, kann dieser Materialtransformation kein fachlich motivierter Vorgang zugeordnet werden. Damit sind der Materialversorger und der zugehörige Aspekt *speicherbar* rein technische Komponenten, die im fachlichen Modell nicht sichtbar sein sollten. Im Gegensatz zum Materialversorger, der durch den Materialverwalter gekapselt wird, schlägt sich der Aspekt *speicherbar* auf alle fachlichen Materialien durch. Da zudem nach [Züllighoven 94] Materialklassen keine eigenständigen Ein- und Ausgabeoperationen besitzen dürfen, sondern nur Zugriffsoperationen bereitstellen, die den Umgangsformen der Materialien entsprechen und somit fachlich interpretierbar sind, sind Operationen wie *load()*, *store()* und *delete()* bzw. *readFrom()* und *writeTo()* per definitionem in Materialklassen nicht zulässig.

Ferner steht der Aspekt *speicherbar*[41] im Widerspruch zu der üblichen Aspektkopplung nach der Methode WAM, weil er so spezifiziert wurde, daß das Datenbank-Frontend direkt vom speicherbaren Material benutzt wird und das Wissen wo, wie und mit welchen referenzierten Materialien sich ein Material ablegt, Bestandteil der Materialklasse ist. Das gleiche gilt auch für den Einsatz des Atomizer-Musters, bei dem das konkrete Material die Operationen *readFrom()* und *writeTo()* bereitstellt, in denen das Material den Atomarisierer verwendet,

[40] auch der Datenbankautomat ([Kilberth et al. 93])
[41] vgl. [Wulf, Weske 94]

um sich zu erzeugen bzw. extern abzulegen. Darüber hinaus ist die Kopplung durch den Aspekt *speicherbar* vor allem deshalb problematisch, weil die fachlichen Materialklassen Wissen über die technischen Details ihrer Objektstruktur bzw. ihrer Repräsentation in der Datenbank besitzen[42]. Damit ist jede fachliche Materialklasse abhängig von mindestens einer technischen Komponente, was im krassen Widerspruch zu der expliziten Trennung der fachlichen und technischen Anteile eines Informationssystems steht. Diese Korrelation zwischen der fachlichen und der technischen Welt kann durch das Entwurfsmuster Atomizer nur teilweise gelöst werden, da auch unter Einsatz dieses Musters das Material Wissen über seine technische Repräsentation hält.

Wie wir gesehen haben, werden die Materialklassen durch die Implementation des Protokolls der Aspektklasse *speicherbar* zwangsläufig abhängig von einem konkreten persistenten Speicher. Aus diesem Grund sind im Falle des Austauschs des persistenten Speichers durch einen anderen alle Materialklassen zu modifizieren, indem entweder der zur Konkretisierung des Protokolls des Aspekts *speicherbar* geschriebene Implementationscode auszutauschen oder durch ein weiteres Protokoll (zum Beispiel ein Aspekt *speicherbar2*), das die Transformation in einen anderen konkreten persistenten Speicher erlaubt, zu ergänzen ist. Dieses Problem könnte durch den Einsatz des Entwurfsmusters Atomizer gelöst werden.

Abgesehen von diesen im Zusammenhang mit der Methode WAM und der Vermischung fachlicher und technischer Konzepte stehenden Problemen bieten die Aspekte *speicherbar*[43] und *atomarisierbar* drei wesentliche technische Unzulänglichkeiten, die die Einsatzfähigkeit der Materialverwaltung fraglich erscheinen läßt:

1. *Redundanz bei der Implementation:*
   Es müssen für jede Materialklasse stets zwei Operationen, eine Lese- und eine Schreiboperation, implementiert werden. Diese Aufgabe ist nicht nur stupide, sondern auch insbesondere im Fall von nachträglichen Änderungen an den Materialklassen fehlerträchtig. Da es sich lediglich um eine bloße Anwendung von Meta-Objekt-Wissen handelt, wäre es sinnvoll, die Operationen nicht nur wie in [Riehle et al. 96] vorgeschlagen zu einer Operation zusammenzufassen, sondern das Meta-Objekt-Wissen vollständig von den Materialklassen zu entkoppeln.

2. *Probleme bei der Mehrfachvererbung:*
   Wenn es - bedingt durch den Einsatz von Mehrfachvererbung - zur Bildung von Diamantstrukturen[44] in der Klassenhierarchie kommt, muß der

[42] vgl. die Operation *ErzeugeDatenbankRepraesentation()* in [Kilberth et al. 93]

[43] vgl. [Wulf, Weske 94]

[44] Beispiel: Eine Klasse A erbt von den Klassen B und C, die eine gemeinsame Oberklasse D besitzen.

Implementator dafür sorgen, daß mehrfach geerbte Attribute nicht doppelt geladen bzw. gespeichert werden.

3. *Probleme bei Referenzen:*
   Die Nutzung der beschriebenen Mechanismen zur Ablage und Rekonstruktion von Materialien impliziert, daß Exemplare von Materialklassen nur komplett mit allen ihren referenzierten Objekten in den transienten Speicher geladen und auch nur komplett wieder aus diesem verdrängt werden können. Zudem gibt es keine Garantie für die Zyklenfreiheit vernetzter Objektstrukturen. Damit kann es dazu kommen, daß ein Lade- bzw. ein Speichervorgang nicht abbricht, weil zyklisch die Ablage eines referenzierten Objekts angestoßen wird. Auch unter Einsatz des Atomizer-Musters sind vernetzte Objektstrukturen nicht partiell handhabbar. Als Abhilfe wird daher in [Riehle et al. 96] vorgeschlagen, referenzierte Objekte ausschließlich explizit nachzufordern oder referenzierte Objekte nur bis zu einer festgelegten Tiefe mitzuladen. Beide Ideen sind problematisch, weil zum einen die Anzahl der expliziten Materialanforderungen stark ansteigt und zum anderen die Bestimmung einer optimalen Tiefe stark kontextabhängig und aufgrund der Asymmetrie von vernetzten Objektstrukturen nicht möglich ist[45].

Zusammenfassend ist damit festzuhalten, daß die Materialverwaltung ein Gemisch fachlicher und technischer Konzepte und Komponenten darstellt. Es liegt damit keine klare Trennung der fachlichen und technischen Anteile vor, so daß der Einsatz der Materialverwaltung als rein fachliche Schnittstelle eines von den technischen Komponenten abstrahierenden Subsystems nicht möglich ist. Darüber hinaus gibt es eine Reihe pragmatischer Einschränkungen (zum Beispiel keine partielle Handhabung vernetzter Objektstrukturen), die die praktische Einsatzfähigkeit der Konzepte der Materialverwaltung fraglich erscheinen lassen. Aufgabe der folgenden Kapitel muß es daher sein, eine Abstraktion zu finden und zu beschreiben, die aus der fachlichen Perspektive betrachtet intuitiv als unterer Abschluß und damit als ein fachliches Basiskonzept verstanden werden kann. Eine solche Abstraktion bietet das im folgenden Kapitel dargestellte Konzept des *Materialraums*.

[45] Bereits das Laden eines Buchs in einer Tiefe von zwei Referenzen kann dazu führen, daß der komplette Verlag mit allen seinen Büchern mitgeladen werden muß, wenn das Buch eine direkte Referenz auf den Verlag hält.

Kapitel 4

Der Materialraum

Im Fokus dieses Kapitels steht das Konzept des *Materialraums*, der alle existierenden Materialien verwahrt und diese von außen zugreifbar macht. Es wird sich zeigen, daß das Konzept des *Materialraums* die in Kapitel 3 genannten Anforderungen an eine fachlich generelle Abstraktion zur Kapselung der technischen Welt erfüllt, so daß der *Materialraum* als ein fachliches Basiskonzept in die Entwicklung beliebiger fachlich spezifischer Anwendungen einfließen kann.

Zunächst wollen wir den für das Konzept des *Materialraums* fundamentalen Materialbegriff präzisieren, indem wir in einem ersten Abschnitt die Komposition, die Identität, die Gleichheit, die Erstellung von Kopien sowie die Lebensdauer von Materialien eingehend untersuchen wollen. Anschließend entwickeln wir ein erweitertes Konzept zur fachlichen Zugriffskontrolle. Darauf aufbauend erfolgt abschließend die Darstellung des *Materialraums* sowie dessen Einordnung bzgl. fachlich höherer Konzepte (zum Beispiel Arbeitsumgebung, Raummetapher).

4.1 Erweiterung des Materialbegriffs

Die in Kapitel 3 gegebene Definition des Materialbegriffs ist für unsere Zwecke nicht ausreichend, weil sie keine Aussagen über die Komposition, die Identität und die Gleichheit sowie die Lebensdauer von Materialien macht. Aus diesem Grund sollen in diesem Abschnitt die bisher vernachläßigten Konzepte fachlich diskutiert werden, um abschließend eine erweiterte Definition des fachlichen Materialbegriffs geben zu können.

4.1.1 Verknüpfung und Komposition

Die Betrachtung isolierter Materialien stellt lediglich eine eingeschränkte Sichtweise dar. Ein Material besitzt nicht nur einen Materialzustand sowie eine Menge von Umgangsformen, sondern kann zudem auch in Beziehung mit anderen Ma-

terialien stehen. Erst durch eine Betrachtung dieser Beziehungen läßt sich der Zusammenhang zwischen Materialien erschließen. Dazu werden wir in diesem Unterabschnitt unterschiedliche Arten von Beziehungen zwischen Materialien identifizieren, die wir im folgenden als eigenständige Konzepte und damit losgelöst vom Materialzustand betrachten wollen.

Begriff 14 : *Umgangsform*[a]
Eine Umgangsform ist eine charakteristische Handlung an oder mit einem Material ([Gryczan 95]).

[a] vgl. hierzu den Begriff der Handlung in Unterabschnitt 3.2.1

Aufgrund von Gemeinsamkeiten der Materialien, die sich an den von ihnen bereitgestellten Umgangsformen manifestieren, lassen sich Materialien hinsichtlich ihrer Umgangsformen ordnen und gruppieren. Die auf diese Weise geordneten Materialien führen zu den relevanten *Begriffen* der Arbeitsdomäne und den dahinterstehenden *Konzepten* ihrer Verwendung (vgl. [Kilberth et al. 93]). Im Gegensatz zu den in [Kilberth et al. 93] verwendeten Begriffen ,,Begriff" bzw. ,,Konzept" wollen wir in Anlehnung an die Konzepte der Objektorientierung im folgenden von dem *Typ eines Materials* sprechen bzw. als Synonym den Begriff des ,,Materialtyps" verwenden, da die Begriffe ,,Begriff" und ,,Konzept" sehr generisch sind und ihre Benutzung aus diesem Grund zu Verwirrungen führen könnte.

Ein Materialtyp beschreibt damit einen spezifischen Satz von Umgangsformen, so daß alle Materialien, die diese Umgangsformen aufweisen, diesem Materialtyp zugeordnet werden können. Die Materialtypen sollten in ein zyklenfreies Begriffsgerüst/Materialtypgerüst (Begriffshierarchie/Materialtyphierarchie) eingeordnet werden. Während jedem Materialtyp beliebig viele Unterbegriffe/Untermaterialtypen zugeordnet werden können, sollte ein Materialtyp maximal einen Oberbegriff/Obermaterialtyp besitzen (vgl. [Kilberth et al. 93]). Diese Art von Beziehung zwischen den Materialtypen wird auch als *Spezialisierung/Generalisierung* bezeichnet. Eine ausführliche Beschreibung findet sich in [Kilberth et al. 93].

Neben den verhaltensorientierten Spezialisierungs- bzw. Generalisierungsbeziehungen können strukturorientierte *Assoziationen* zwischen den Materialtypen bestehen. Eine *Assoziation* beschreibt eine Gruppe von Verknüpfungen mit gemeinsamer Semantik ([Rumbaugh 93]) und kann durch Vergabe von Namen benannt werden. Dabei drückt der Name in der Regel eine bestimmte Richtung aus, obwohl eine *Assoziation* in beide Richtungen betrachtet werden kann. Zum Beispiel sind Bücher *im-Besitz* von Personen, die ihrerseits Bücher *besitzen*[1]. *Assoziationen* sind für Systeme der realen Welt ein wichtiges Modellierungskonstrukt

[1] Es handelt sich bei diesem Beispiel um genau eine *Assoziation* zwischen den Materialtypen Person und Buch mit einer 1:n Kardinalität. Daneben sind die Kardinalitäten 1:1 und m:n gebräuchlich.

([Rumbaugh 93]) und bilden ein adäquates Modellierungsmittel in der Analyse ([Gamma et al. 95]).

Begriff 15 : *Assoziation zwischen Materialtypen*
Eine Assoziation zwischen Materialtypen ist eine Beziehung zwischen Materialtypen. Eine Assoziation legt die in Beziehung zueinander stehenden Materialtypen, ihre Rollen innerhalb der Beziehung sowie die Kardinalität der Beziehung fest. Eine Assoziation ist bi-direktional und kann benannt werden.

Besteht zwischen zwei Materialtypen (zum Beispiel *Buch* und *Autor*) eine Assoziation (zum Beispiel *wurde-geschrieben-von* (Rolle des Buchs)), so können Exemplare dieser Materialtypen (zum Beispiel *Die Deutschstunde* und *Siegfried Lenz*) untereinander[2] *verknüpft* sein. Eine einzelne Verknüpfung zwischen Materialien ist damit ein Exemplar einer Assoziation und modelliert eine Verbindung zwischen Materialien[3] (zum Beispiel *Die Deutschstunde wurde geschrieben von Siegfried Lenz*).

Begriff 16 : *Materialverknüpfung*
Eine Materialverknüpfung ist ein Exemplar einer Assoziation und beschreibt die Verbindung zwischen zwei Materialien. Materialien können untereinander beliebig verknüpft sein, so daß sich eine vernetzte Materialstruktur (Materialgraph) ergibt, deren Ausprägung beliebig sein kann.

Begriff 17 : *Verknüpftes Material*
Ein Material heißt verknüpft, wenn es mit mindestens einem anderen Material durch eine Materialverknüpfung verbunden ist.

Eine Sonderform der Assoziation ist die *Aggregation*, die die Beziehung zwischen einem Ganzen und seinen Teilen beschreibt. Die *Aggregation* ist eine strengere Beziehung als die Assoziation, da sie im Gegensatz zur Assoziation antisymmetrisch[4] und transitiv[5] ist.

Begriff 18 : *Aggregation von Materialtypen*
Die Aggregation ist eine antisymmetrische, transitive Assoziation, die die Beziehung zwischen einem Ganzen (Rolle: Kontext) und seinen Teilen (Rolle: Komponente) ausdrückt, wobei sich das Ganze aus den Teilen zusammensetzt.

---

[2] auch uni-direktional

[3] vgl. [Rumbaugh 93]

[4] Eine *Aggregation* ist antisymmetrisch, d.h., wenn A Teil von B ist, dann ist B nicht Teil von A.

[5] Eine *Aggregation* ist transitiv, d.h., wenn A Teil von B ist und B Teil von C, dann ist A auch Teil von C.

So wie auf der Ebene der fachlichen Materialtypen die Aggregation eine Sonderform der Assoziation darstellt, ist auf der Ebene der Materialexemplare die Materialkomposition eine Sonderform der oben vorgestellten Materialverknüpfung.

Begriff 19 : *Materialkomposition*
Eine Materialkomposition ist ein Exemplar einer Aggregation und beschreibt die antisymmetrische und transitive Beziehung zwischen einem Ganzen (Kontextmaterial) und einem seiner Teile (Komponentenmaterial). Ein Komponentenmaterial kann selbst das Kontextmaterial einer anderen Materialkomposition sein.

Begriff 20 : *Komplexes Material*
Ein komplexes Material ist eine Menge von Materialien, die aus beliebig vielen Komponentenmaterialien und genau einem Kontextmaterial besteht. Das Kontextmaterial steht mit allen Komponentenmaterialien in Materialkompositionsbeziehung. Das Kontextmaterial modelliert die durch das komplexe Material bereitgestellte Abstraktion und ermöglicht somit die Betrachtung und Bearbeitung der Komponentenmaterialien als konzeptionelle Einheit. Komplexe Materialien enthalten sich oder sind disjunkt.

Ein komplexes Material umfaßt neben seinem Kontextmaterial beliebig viele Komponentenmaterialien. Da Materialien untereinander nicht beliebig in Materialkomposition stehen können, ergibt sich im Gegensatz zur Materialverknüpfung ein baumartiger Materialgraph. Es ist zudem festzuhalten, daß das Kontextmaterial als Schnittstelle (Fassade) und damit als logischer Stellvertreter für das komplexe Material dienen kann und deshalb auch denselben Namen wie das zugehörige komplexe Material trägt. Ein nicht komplexes Material ist vergleichbar mit einem Kontextmaterial, dessen bereitgestellte Abstraktion auch ohne Komponentenmaterialien immer gültig ist[6] [7].

Durch die Einführung der Materialkomposition wird es möglich, die im Vergleich zur Materialverknüpfung zwischen einem Buch und seinem Besitzer semantisch strengere Beziehung zwischen dem Buch und seinen Seiten adäquat auszudrücken. Anhand dieses Beispiels läßt sich zeigen, daß Materialverknüpfungen in der Regel sehr viel öfter erstellt und wieder aufgelöst werden als Materialkompositionen (vgl. [Rumbaugh 93]).

[6] vgl. fachlicher Wert

[7] Im folgenden wollen wir komplexe von nicht-komplexen Materialien abgrenzen und verwenden dort, wo es nicht aus dem Kontext ersichtlich ist, den Begriff „Material" ausschließlich für nicht-komplexe Materialien. Ein solches nicht-komplexes Material kann beliebig verknüpft sein und/oder in einer Materialkomposition die Rolle eines Kontext- bzw. Komponentenmaterials innehaben.

Da das Konzept der Materialkomposition bislang im Zusammenhang mit der Methode WAM wenig Beachtung gefunden hat und es sich zudem bei der Materialkomposition um ein für diese Arbeit relevantes Konzept handelt, wollen wir im folgenden ausgehend von der bereits etablierten Werkzeugkomposition[8] die Notwendigkeit der Einführung der Materialkomposition motivieren.

Ein grundlegendes Vorgehen bei der Werkzeugkonstruktion ist die sogenannte Werkzeugkomposition, bei der komplexe Werkzeuge aus Kontext- und Subwerkzeugen zusammengesetzt werden: Zum Beispiel wird die komplexe Aufgabe des Werkzeugs PKW (sicheres Verkehrsmittel) von einem zusammengesetzten Werkzeug (ein PKW besteht aus Blinkern, Airbags usw.) erledigt, wobei Teilaufgaben (Richtungsanzeige, Schutz der Insassen) durch die Subwerkzeuge (Blinker, Airbag) erledigt werden.

Da jedes Werkzeug in einem veränderten Kontext zu einem Material werden kann[9], stellt sich automatisch die Frage, warum im Gegensatz zur Werkzeugkomposition das Prinzip der Materialkomposition bislang nicht behandelt worden ist. Neben dem Arbeitsmittelcharakter, den der PKW während einer Fahrt vom Ort A zum Ort B hat, kann der Wagen zum Beispiel während einer Inspektion in einer Werkstatt oder als Modell innerhalb eines CAD-Systems durchaus Materialcharakter haben, wobei seine Struktur unverändert bleibt. In einem CAD-System fungiert der PKW als Material, wobei in der kostenminimierten industriellen Automobilentwicklung gerade die Interpretation des Automobils als zusammengesetztes Material (komplexes Material) eine wichtige Rolle spielt[10].

Aus dieser Motivation heraus, ist es sinnvoll, neben der hierarchischen Werkzeugkomposition (Kontext- und Subwerkzeuge) auch das Konzept einer Materialkomposition (Kontext- und Komponentenmaterialien) zu etablieren. Dabei geht es uns weniger um die Komplexitätsreduktion durch eine Hierarchie- und die zugehörige Abstraktionsbildung[11]. Vielmehr soll die Materialkomposition als ein weiteres fachliches Basiskonzept für eine adäquate Modellierung fachlich spezifischer Konzepte wie zum Beispiel die Komposition[12] und die Wiederverwendung[13] fachlicher Arbeitsgegenstände dienen.

---

[8] ,,Werkzeuge werden rekursiv aus Subwerkzeugen aufgebaut. Jedem Werkzeugobjekt in der resultierenden Baumstruktur kommt eine Aufgabe zu, welche es durch eigene Leistung sowie Delegation von Teilaufgaben an Subwerkzeuge realisiert." ([Riehle 95])

[9] vgl. Abschnitt 3.2

[10] vgl. hierzu zum Beispiel die Standardisierung von Baugruppen in der Automobilindustrie

[11] vgl. das Prinzip von ,,devide et impera"

[12] Beispiel: Ein Auto besteht aus Rädern, Chassis, Sitzen, Motor usw.

[13] Die Wiederverwendung von Materialien sollte stets fachlich motivierbar sein. Es ist keinesfalls unsere Intention durch ein weiteres fachliches Konzept die Wiederverwendung von Implementationscode auf der technischen Ebene zu rechtfertigen.

Neben physischen Gegebenheiten (zum Beispiel physikalisches Enthaltensein) können durch den Einsatz der Materialkomposition auch andere bestehende logische Abhängigkeiten (zum Beispiel konzeptionelle Zusammengehörigkeit) zwischen Materialien ausgedrückt werden[14].

Da das Kontextmaterial als logischer Stellvertreter für das komplexe Material dient, stellt es eine Abstraktion bezüglich der Komponentenmaterialien dar. Während der Ingenieur ein Auto lediglich als eine Ansammlung von Komponentenmaterialien betrachtet, die zusammengenommen Auto genannt werden, ist es für den Fahrzeugspediteur unerheblich, aus welchen Komponenten sich ein zu transportierendes Auto zusammensetzt. Auch der Bibliothekar verwaltet Bücher stets als Ganzes, während der Autor, der Lektor und der Leser das Buch als gebundene Menge von Seiten betrachten.

Materialkomposition impliziert damit weder die Kapselung noch den Verlust der Eigenständigkeit von Komponentenmaterialien. Ein komplexes Material kann zwar als Ganzes betrachtet werden, aber nach außen sind sowohl das Kontext- als auch die Komponentenmaterialien sichtbar. Damit ist es einem Materialbenutzer möglich, Komponentenmaterialien aus einem komplexen Material zu entnehmen, um auf diese Weise das komplexe Material aufzuteilen und es unterschiedlichen Werkzeugen zur Bearbeitung übergeben zu können. Das Kontextmaterial und die Komponentenmaterialien eines komplexen Materials sind damit prinzipiell eigenständig.

Ein solches Vorgehen läßt sich anhand des oben gegebenen Beispiels motivieren: Ein schadhafter PKW wird in der Werkstatt teilweise demontiert, um Fehler erkennen und ggf. beseitigen zu können. Auch ein Buch kann in seine Komponentenmaterialien (zum Beispiel Umschlag und Kapitel) aufgeteilt werden. Damit wird es möglich, daß der Layouter das Titelblatt gestalten und der Lektor die ersten Kapitel redigieren kann, während der Autor noch das Schlußwort schreibt.

Nach der Entnahme von Komponentenmaterialien aus komplexen Materialien kann einem Kontextmaterial in der fachlichen Welt angesehen werden, daß Komponentenmaterialien entnommen worden sind und damit fehlen. So wie es selbstverständlich ist, daß der Leser eines Buchs den Verlust von Seiten oder der Besitzer eines Automobils das Fehlen des Fahrersitzes bemerkt, sieht selbst ein Laie einem Flugzeug an, daß eine Tragfläche fehlt, weil er entweder Wissen darüber hat, daß ein Flugzeug in der Regel zwei Tragflächen besitzt oder er die unbesetzte Tragflächenaufhängung am Flugzeugrumpf bemerkt.

[14] „Die Aggregation kann, muß aber nicht einen physikalischen Aufbau repräsentieren." ([Booch 94])

Es ist an dieser Stelle anzumerken, daß ein Kontextmaterial bei Verlust von Komponentenmaterialien seine Daseinsberechtigung verlieren kann. Dies läßt sich wie folgt motivieren: Ein Auto, aus dem der Fahrersitz ausgebaut worden ist, ist zwar durchaus noch als Auto erkennbar; wird ein Auto allerdings komplett in seine Komponenten zerlegt, ist diese Abstraktion nicht länger aufrechtzuerhalten. Ob ein Kontextmaterial als Stellvertreter und damit als eine Abstraktion einer Menge von Komponentenmaterialien fungieren kann, ist abhängig von dem prägenden Einfluß, den diese Komponentenmaterialien auf die bereitgestellte Abstraktion haben. So ist die Abstraktion, ein Auto zu sein, für das Automobil-Chassis und die Räder noch gegeben, während sie für ein Armaturenbrett, die Vordersitze und die Türen nicht adäquat erscheint. Die Entscheidung, ob eine Abstraktion adäquat ist, ist somit ausschließlich aus dem fachlich spezifischen Kontext entscheidbar, während das Erkennen des Verlusts von Komponentenmaterialien fachlich genereller Natur ist.

Es ist damit festzuhalten, daß ein Komponentenmaterial abhängig von seinem fachlichen Anwendungskontext aus einem komplexen Material entnommen, getrennt bearbeitet und an andere Kontextmaterialien gebunden werden kann. Dabei ist einem Kontextmaterial das Fehlen eines Komponentenmaterials anzusehen. Inwieweit sich diese Eigenständigkeit der Komponentenmaterialien auf die Abhängigkeit bzw. auf die Lebensdauer von Kontext- und Komponentenmaterialien auswirkt, werden wir im Unterabschnitt 4.1.4 darstellen.

4.1.2 Identität

So wie im Deutschen zwischen ,,dasselbe" und ,,das gleiche" unterschieden wird, soll im folgenden der Begriff *Identität*[15] gefaßt und gegen den Begriff der *Gleichheit* abgegrenzt werden.

Wenn ein Material nur mit sich selbst in allen Ausprägungen seiner Merkmale (Materialzustand) übereinstimmt, dann besitzt es die Eigenschaft der *abstrakten Identität* ([Kirschke 95]). Diese Definition von Identität ist zwar hilfreich, um ein Material gegen ein anderes zu einem bestimmten Zeitpunkt abgrenzen zu können ([Kirschke 95]), aber dem Wunsch ein Material zu einem beliebigen Zeitpunkt identifizieren[16] und damit als solches wiedererkennen zu können, wird diese Definition nicht gerecht. Der ausschließliche Vergleich der Merkmale ist nicht ausreichend, wie wir an einem kleinen Beispiel verdeutlichen wollen: Wird zum Beispiel ein Auto umgespritzt, so ändert sich die Ausprägung seines Merkmals Autofarbe. Das umgespritzte Auto wäre somit nicht mehr mit dem Auto, das in

[15] identitas lat. (Wesenseinheit)

[16] Mit dem Begriff *Identifikation* wird der Vorgang der Feststellung der Identität bezeichnet.

die Lackiererei gebracht worden ist, identisch. Das widerspricht dem intuitiven Verständnis von Identität.

Festzuhalten ist damit, daß die Identität eines Materials nicht allein durch den Zustand des Materials festgelegt ist. Vielmehr bezeichnet die Identität eine ausgezeichnete Materialeigenschaft, deren Ausprägung unveränderlich und einmalig ist. Die Materialidentität wird damit eineindeutig einem zugehörigen Material zugeordnet, welches dadurch als solches identifizierbar wird. Ein Material ist somit nur mit sich selbst identisch und kann durch seine Identität von allen anderen Materialien unterschieden werden. In Anlehnung an [Josuttis 94] sind zwei Materialien dann identisch, wenn jede Änderung an dem einen Material das andere ändert.

> **Begriff 21 :** *Materialidentität*
> fachliche Interpretation:
> Die Materialidentität ist die Eigenschaft eines Materials, die es von allen anderen Materialien unterscheidet (nach [Booch 94]).

Für Materialien, Identität und Identitätsrepräsentationen gelten folgende Regeln (vgl. [Kirschke 95]) :

- Materialien sind individuell, d.h. jedes Material hat eine eigene, genau ihm zugehörige Identität.
- Es gibt keine Identität ohne ein dazugehöriges Material.
- Die Identität eines Materials wird technisch durch eine geeignete Form repräsentiert[17] (Materialidentifikator).

Auch in fachlichen Kontexten werden Materialien mit Identifikatoren versehen, die die Identität von Materialien repräsentieren und damit die Identifikation von Materialien ermöglichen. Jeder PKW hat zum Beispiel eine Fahrgestellnummer, die eineindeutig vergeben und in einem Straßenverkehrsamt als Stellvertreter für das eigentliche Automobil verwandt wird. Daneben ist jedem angemeldeten PKW ein eindeutiges Kraftfahrzeugzeichen zugeordnet, welches in seiner Ausprägung[18], Struktur[19] und Farbe[20] sowie durch angebrachte Plaketten[21] zusätzliche Informationen über das Auto und dessen Eigentümer zur Verfügung stellt. Fachliche Identitätsrepräsentationen sind damit abhängig von der fachlichen Domäne und unterliegen deshalb einer fachlich spezifischen Semantik. Sie sollten daher wie

[17] Identitätsrepräsentationen können auch als Stellvertreter für Materialien verwandt werden (vgl. [Kirschke 95]).
[18] Beispiel: HH-DX 288 (Das Kürzel HH steht für Hansestadt Hamburg)
[19] Beispiel: PI-3031 (behördliches Kennzeichen)
[20] grün, rot, schwarz
[21] Plakette des Straßenverkehrsamtes bzw. die TÜV-Plakette

auch andere fachlich spezifische Namen in der fachlich spezifischen Welt modelliert werden.

Auf der anderen Seite dienen in einigen Anwendungen fachlich spezifische Konzepte zur Adressierung von Materialien auf der technischen Ebene. In der Arbeit [Wulf, Weske 94] wird zum Beispiel das Konzept der Identität ausschließlich auf der Basis fachlich spezifischer Materialnamen bereitgestellt, die direkt an technische Hauptspeicheradressen gebunden[22] werden[23]. Dadurch kommt es zu einer starken Korrelation zwischen der fachlich spezifischen und der technischen Welt. Es erscheint daher sinnvoll, die beiden Konzepte durch Einführung des fachlich generellen Konzepts der Identität voneinander zu entkoppeln.

Durch die Entkopplung von fachlich spezifischen Materialnamen und technischen Details durch das fachliche Basiskonzept der Materialidentität erhalten wir ähnlich der 3-Schicht-Architektur drei aufeinander aufbauende Konzepte:

1. *Materialname (fachlich spezifisch)*:
   Ein fachlich spezifischer Materialname wird durch den Anwender vergeben, dient zur Bezeichnung eines Materials und kann fachlich spezifische Informationen über das Material enthalten. Die Materialnamen müssen den Materialien nicht eindeutig oder eineindeutig zugeordnet sein. Somit kann ein Material mehrere Materialnamen haben (*aliasing*) und ein Materialname mehr als ein Material bezeichnen. Die Materialnamen eines komplexen Materials entsprechen denen des Kontextmaterials. Die Materialnamen der Komponentenmaterialien sind beliebig wählbar.

   Fachliche Identitätsrepräsentationen (zum Beispiel Kraftfahrzeugzeichen, Personalausweisnummer, Seriennummer usw.) stellen spezielle fachliche Materialnamen dar, weil sie den Materialien eindeutig oder eineindeutig zugeordnet sind.

2. *Materialidentität (fachlich generell)*:
   Jedes fachliche Material hat eine Materialidentität, die es von allen anderen Materialien unterscheidet. Das Konzept der Materialidentität ist fachlich genereller Natur. Im Gegensatz zu fachlichen Namen wird die Vergabe von Materialidentitäten nicht durch den Anwender durchgeführt, sondern erfolgt automatisch zum Zeitpunkt der Materialerzeugung durch eine zusätzliche Instanz. Die Identität eines komplexen Materials wird durch die Materialidentität des Kontextmaterials bestimmt. Jedes Komponentenmaterial

[22] vgl. hierzu das Materialmagazin

[23] Das Fehlen eines Identitätskonzepts macht sich auf der technischen Ebene besonders bei der Anbindung persistenter Speicher negativ bemerkbar, weil die Referenzenbildung zwischen persistenten Materialrepräsentationen auf der Basis transienter Hauptspeicheradressen, die nur innerhalb eines Programmlaufs gültig sind, nicht möglich ist (vgl. hierzu Kapitel 5 und 6).

hat eine von der Identität des Kontextmaterials losgelöste eigene Materialidentität.

Meist wird die Materialidentität als Bestandteil des Materialzustands betrachtet, obwohl Identität fachlich eine andere Qualität als beispielsweise die Farbe eines Materials hat. Aus diesem Grund wollen wir die Materialidentität als eine ausgezeichnete fachlich generelle Eigenschaft eines Materials und getrennt von dem Materialzustand betrachten (vgl. hierzu Unterabschnitt 4.1.3). Durch die Einführung des Konzepts der Materialidentität wird es möglich, fachlich spezifische Materialnamen anstatt an technische Konzepte an Materialidentitäten und damit an die fachlichen Materialien selbst zu binden.

In der (programmier-)technischen Umsetzung kann die Identität eines Materials als ein Attribut eines Materialobjekts (Materialidentifikator (MID)) realisiert werden, dessen Ausprägung durch eine zusätzliche Instanz, die die Eineindeutigkeit der Zuordnung sicherstellt, vergeben wird. Die (programmier-)technische Umsetzung des fachlich generellen Konzepts der Materialidentität wird in Kapitel 6 ausführlich dargestellt.

3. *Materialadresse (technisch)*:
   Eine technische Materialadresse benennt den Ort im transienten oder persistenten Speicher, an dem sich die technische Repräsentation eines bestimmten Materials befindet. Damit unterscheidet sich die Materialadresse vom Materialidentifikator (MID), der lediglich eine Schlüsselinformation bereitstellt, mit deren Hilfe eine zugehörige technische Repräsentation ausfindig gemacht werden kann. Die Adressierung über einen Materialidentifikator (MID) kann damit im Gegensatz zur Adressierung über eine technische Materialadresse nur mittelbar durch eine Zuordnungsinstanz erfolgen, die den Materialidentifikator (MID) auf eine technische Materialadresse abbildet. Eine nähere Betrachtung dieser technischen Konzepte findet sich in Kapitel 6.

Die Materialidentität dient als fachlich generelles Konzept, durch das die Identifikation beliebiger fachlich spezifischer Materialien ermöglicht und damit ihre Individualität sichergestellt wird.

4.1.3 Gleichheit

Die Überprüfung von Materialien auf die Gleichheit ihrer Identität (*Identitätsgleichheit*) ist nicht ausreichend, und es bedarf weiterer Konzepte zur Gegenüberstellung von Materialien. Aus diesem Grund wollen wir uns im folgenden mit weiteren unterschiedlichen Arten von Materialgleichheit beschäftigen.

Nach [Booch 94] gibt es für das Konzept der Gleichheit zwei verschiedene Interpretationen. Erstens kann Gleichheit bedeuten, daß zwei Materialnamen dasselbe Material bezeichnen (*Aliasgleichheit*) und zweitens kann Gleichheit bedeuten, daß zwei Materialnamen zwei unterschiedliche Materialien bezeichnen, deren Zustand gleich ist (*Zustandsgleichheit*)[24]. Daneben gibt es in der Umkehrung der Aliasgleichheit die *Namensgleichheit*, die dann gegeben ist, wenn Materialien einen gemeinsamen Materialnamen haben. Eine vierte Form der Gleichheit führt [Josuttis 94] mit der *Semantikgleichheit* ein. Die Semantikgleichheit bezeichnet die Gleichheit von Materialien[25], die auf verschiedene Weise die gleiche Bedeutung (Semantik) haben (vgl. [Josuttis 94]). Beispielsweise sind die Uhrzeitrepräsentationen „5 Uhr 15", „viertel nach fünf" bzw. „viertel sechs" in einem fachlich spezifischen Kontext semantisch gleich, weil sie trotz unterschiedlicher Repräsentationen die gleiche Bedeutung haben.

Die Namens- und Aliasgleichheit sowie die Semantikgleichheit sind fachlich spezifisch geprägt, weil sie zum einen wie die Namens- und die Aliasgleichheit auf dem Konzept fachlich spezifischer Materialnamen beruhen und zum anderen wie die Semantikgleichheit eine bestehende gleiche Semantik von Materialien nur aus dem fachlich spezifischen Kontext, in dem sie genutzt werden, entscheidbar ist. Aus diesem Grund sollen die Namens-, die Alias- sowie die Semantikgleichheit nicht weiter behandelt werden. Wir wollen uns im folgenden auf eine nähere Betrachtung der Zustandsgleichheit und die Abgrenzung dieser zur Identitätsgleichheit beschränken.

Dazu definieren wir zunächst den Begriff der Identitätsgleichheit:

> **Begriff 22 :** *Materialidentitätsgleichheit*
> fachliche Interpretation:
> Zwei Materialien sind *identisch/identitätsgleich*, wenn sie dieselbe Materialidentität besitzen. Zwei komplexe Materialien sind genau dann identisch, wenn ihre Kontextmaterialien identisch sind.

Wie die Identitätsgleichheit ist auch die Zustandsgleichheit ein fachliches Basiskonzept und sollte daher auf beliebige fachlich spezifische Materialien anwendbar sein. Es ist allerdings zu klären, wann von der Zustandsgleichheit komplexer Materialien gesprochen werden kann. Im Gegensatz zu den in der Literatur[26] vorgestellten Konzepten von *shallow-equal* und *deep-equal* wollen wir den Begriff der Zustandsgleichheit auf die Betrachtung der Materialkomposition beschränken. Damit erhalten wir die folgenden Definitionen für die Konzepte der flachen und

[24] Diese Arten von Gleichheit werden zwar im programmiersprachlichen Kontext dargestellt, sind aber fachlich interpretierbar.

[25] auch Materialien unterschiedlichen Typs

[26] vgl. u.a. [Heuer 92] und [Meyer 90]

der tiefen Zustandsgleichheit, die wir im folgenden auch als flache bzw. tiefe Materialzustandsgleichheit bezeichnen wollen:

Begriff 23 : *Flache Materialzustandsgleichheit*
Zwei Materialien sind *flach zustandsgleich*, wenn die Ausprägungen ihrer Eigenschaften (Materialzustände) gleich sind. Eine Betrachtung von Materialidentitäten, Materialverknüpfungen und Materialkompositionen findet nicht statt.

Begriff 24 : *Tiefe Materialzustandsgleichheit*
Zwei Materialien sind *tief zustandsgleich*, wenn sie flach zustandsgleich sind und ihnen tief zustandsgleiche Komponentenmaterialien zugeordnet sind. Eine Betrachtung von Materialidentitäten und Materialverknüpfungen findet nicht statt.

Für Materialien, die nicht aus Komponentenmaterialien komponiert sind, besteht damit kein Unterschied in der Überprüfung der flachen und der tiefen Materialzustandsgleichheit.

4.1.4 Erzeugung, Kopie, Auflösung und Lebensdauer

Bezüglich der Lebensdauer komplexer Materialien läßt sich feststellen, daß das Kontextmaterial und die Komponentenmaterialien eines komplexen Materials keineswegs ausschließlich gleichzeitig entstehen bzw. vergehen. Auf der einen Seite kann zum Beispiel einem PKW im Austausch mit einem defekten Blinker ein neuer als Ersatzteil eingebaut werden, und auf der anderen Seite werden intakte Fahrzeugkomponenten bei der Autoverwertung vor der Verschrottung bewahrt, um sie als Ersatzteile in andere Automobile wieder einzusetzen. Damit haben das Kontextmaterial und die Komponentenmaterialien eines komplexen Materials prinzipiell voneinander unabhängig eine beliebige Lebensdauer.

Begriff 25 : *Lebensdauer eines Materials*
fachliche Interpretation:
Jedes Material hat eine beliebig lange Lebensdauer und existiert unabhängig von anderen Materialien vom Zeitpunkt seiner expliziten Erzeugung bis zum Zeitpunkt seiner expliziten Auflösung.

Analog zu der flachen und der tiefen Materialzustandsgleichheit definieren wir:

Begriff 26 : *Flache Materialerzeugung*
Bei der flachen Materialerzeugung wird ein (Kontext-)Material ohne adäquate Komponentenmaterialien erzeugt. Dem (Kontext-)Material wird eine Materialidentität eineindeutig zugeordnet. Materialverknüpfungen und Materialkompositionen werden als unbesetzt markiert.

Begriff 27 : *Tiefe Materialerzeugung*
Bei der tiefen Materialerzeugung erfolgt sowohl die flache Erzeugung des (Kontext-)Materials als auch die tiefe Erzeugung adäquater Komponentenmaterialien. Anschließend erfolgt die Materialkomposition von (Kontext-)Material und erzeugten Komponentenmaterialien. Materialverknüpfungen werden als unbesetzt markiert.

Im Gegensatz zur flachen Materialerzeugung dient die tiefe Materialerzeugung als Abstraktion von der expliziten Materialkomposition. Damit kann zum Beispiel von den komplexen Produktionsvorgängen einer Automobilproduktion abstrahiert werden. Alle Arbeitsschritte werden auf eine Umgangsform des Materials zusammengezogen - die tiefe Materialerzeugung. Eine solche Abstraktion ist immer dann erforderlich, wenn die Entstehung eines Materials nicht Schwerpunkt des Interesses oder eine explizite Beschreibung der Materialerzeugung nicht möglich bzw. zu aufwendig ist.

Das Erzeugen zustandsgleicher Materialduplikate (zum Beispiel bei der Produktion von Büchern in einer Druckerei), die im folgenden als *Materialkopien* bezeichnet werden sollen, ist eng mit dem Konzept der Materialzustandsgleichheit (alle gedruckten Bücher einer Auflage sind zustandsgleich) verbunden. Es ist zu unterscheiden, ob ein Material zusammen mit seinen oder ohne seine Komponentenmaterialien dupliziert werden soll. Dazu definieren wir analog zu der flachen und der tiefen Materialzustandsgleichheit die *flache Materialkopie* und die *tiefe Materialkopie*:

Begriff 28 : *Flache Materialkopie*
Bei der Erstellung einer flachen Materialkopie wird das (Kontext-) Material dupliziert. Zugeordnete Komponentenmaterialien und Materialien, die mit dem (Kontext-)Material verknüpft sind, werden nicht dupliziert. Dem duplizierten Material wird eine Materialidentität eineindeutig zugeordnet.

Begriff 29 : *Tiefe Materialkopie*
Bei der Erstellung einer tiefen Materialkopie erfolgt sowohl die Erstellung einer flachen Materialkopie des (Kontext-)Materials als auch die Erstellung tiefer Materialkopien der zugeordneten Komponentenmaterialien. Anschließend erfolgt die Materialkomposition des flach kopierten (Kontext-)Materials und der tief kopierten Komponentenmaterialien. Materialien, die mit dem (Kontext-)Material verknüpft sind, werden nicht dupliziert.

Damit handelt es sich bei der Erstellung einer Materialkopie um einen Kloning-Vorgang, bei dem von einem Material ein zustandsgleiches Replikat erstellt wird[27].

Analog zur Materialerzeugung definieren wir wie folgt die Materialauflösung:

> **Begriff 30 :** *Flache Materialauflösung*
> Bei der flachen Materialauflösung erfolgt ausschließlich die Vernichtung des (Kontext-)Materials. Zugeordnete Komponentenmaterialien und Materialien, die mit dem (Kontext-)Material verknüpft sind, bleiben bestehen.

> **Begriff 31 :** *Tiefe Materialauflösung*
> Bei der tiefen Materialauflösung erfolgt die flache Auflösung des (Kontext-)Materials und die tiefe Auflösung der zugeordneten Komponentenmaterialien. Materialien, die mit dem (Kontext-)Material verknüpft sind, bleiben bestehen.

Komplexe Materialien werden in der Regel komplett, d.h. tief aufgelöst. Allerdings gibt es auch fachliche Kontexte, in denen wie beispielsweise bei der Autoverwertung komplexe Materialien partiell aufgelöst werden können. In fachlichen Kontexten dieser Art ermöglicht die tiefe Materialauflösung analog zur tiefen Materialerzeugung die Abstraktion der eigentlichen Demontage des komplexen Materials in seine Komponentenmaterialien und deren anschließende Beseitigung.

4.1.5 Der erweiterte Materialbegriff

Abschließend wollen wir die in Kapitel 3 gegebene Materialdefinition präzisieren, indem wir den Materialbegriff um Konzepte wie Materialidentität, -verknüpfung, -komposition und -lebensdauer erweitern:

> **Begriff 32 :** *Material (2. Version)*
> Ein Material ist im Rahmen einer Handlung ein Arbeitsgegenstand, der durch Werkzeuge und Automaten bearbeitet, d.h. verändert und sondiert, werden kann. Ein Material hat eine Materialidentität, die es von allen anderen Materialien unterscheidet, und einen Materialzustand, der die kumulierten Ereignisse seines Verhaltens darstellt. Ein Material kann beliebig verknüpft (Materialverknüpfung) und komponiert (Materialkomposition) sein. Die Lebensdauer eines Materials wird ausschließlich durch seine explizite Erzeugung sowie Auflösung bestimmt.

[27] vgl. das Entwurfsmuster Prototyp in [Gamma et al. 95]

So wie im technischen Kontext zwischen objektspezifischen und generischen Operationen unterschieden wird[28], wollen wir die fachlich spezifischen von den fachlich generellen Umgangsformen[29] der Materialien trennen:

- *Fachlich spezifische Umgangsformen:*
  Fachlich spezifische Umgangsformen orientieren sich an einem bestimmten Materialtyp der fachlich spezifischen Arbeitsdomäne. Ihre Semantik ist explizit zu beschreiben und ausschließlich im fachlich spezifischen Kontext gültig. Es sind vier Arten zu unterscheiden:

  1. *Initialisierende Umgangsformen*:
     Den Zustand des Materials initialisierende Umgangsformen
     (Beispiel: Setzen einer initialen Wagenfarbe)
  2. *Terminierende Umgangsformen*:
     Den Zustand des Materials terminierende Umgangsformen
     (Beispiel: Die Autositze werden aus einem Auto ausgebaut,
     bevor es verschrottet wird.)
  3. *Modifizierende Umgangsformen*:
     Den Zustand des Materials verändernde Umgangsformen
     (Beispiel: Änderung der Wagenfarbe durch Umspritzen)
  4. *Sondierende Umgangsformen*:
     Den Zustand des Materials auslesende Umgangsformen
     (Beispiel: Feststellen der aktuellen Wagenfarbe)

- *Fachlich generelle Umgangsformen:*
  Fachlich generelle Umgangsformen müssen nicht im fachlich spezifischen Kontext explizit formuliert werden, sondern können dort implizit vorausgesetzt werden. Jedes fachliche Material verfügt damit über einen Standardsatz fachlich genereller Umgangsformen:

  1. *Materialerzeugung*:
     Ein Material wird als solches erzeugt.
     (Beispiel: Erzeugung eines Autos)
  2. *Materialauflösung*:
     Ein Material wird als solches aufgelöst.
     (Beispiel: Beseitigung eines Autos)
  3. *Materialidentitätsgleichheit*:
     Zwei Materialien können auf Identitätsgleichheit überprüft werden.
     (Beispiel: Identifizierung eines aufgefundenen, entwendeten Autos
     durch den Fahrzeughalter)

[28] vgl. [Heuer 92]

[29] Eine Umgangsform ist eine charakteristische Handlung an oder mit einem Material ([Gryczan 95]).

4. *Materialzustandsgleichheit*:
   Ein Material kann mit einem anderen auf Materialzustandsgleichheit überprüft werden.
   (Beispiel: Feststellung, ob zwei Autos die gleichen Ausstattungsmerkmale aufweisen)
5. *Materialkopie*:
   Ein Material wird dupliziert.
   (Beispiel: Ein Auto wird von einem Konkurrenzunternehmen nachgebaut.)

Die in diesem Abschnitt vorgestellten Konzepte sind zwar aus einer rein fachlichen Perspektive erarbeitet worden, zeigen aber Analogien zu bestehenden technischen Datenmodellen und Objektmodellen. Diese Ähnlichkeit ist darauf zurückzuführen, daß diese Modelle zur technischen Modellierung fachlicher Konzepte dienen. In Kapitel 5 werden wir unterschiedliche Modelle vorstellen und dort bestehende Konzepte vergleichend darstellen. In Kapitel 6 kann dann ausführlich auf die Übertragung der in diesem Abschnitt vorgestellten fachlichen Basiskonzepte in den (programmier-)technischen Kontext der objektorientierten Programmiersprache C++ eingegangen werden.

Nachdem wir unseren Materialbegriff motiviert und dargestellt haben, wollen wir uns im folgenden Abschnitt mit der gemeinsamen Nutzung von Materialien beschäftigen.

4.2 Gemeinsame Nutzung von Materialien

Eine handelnde Instanz (zum Beispiel ein Anwender) führt Handlungen an Materialien innerhalb von Arbeitsvorgängen aus, die ihrerseits in Arbeitsschritte aufteilbar sind[30]. Innerhalb eines Arbeitsvorgangs können prinzipiell beliebig viele Materialien bearbeitet werden. Eine charakteristische Handlung an oder mit einem Gegenstand wird als Umgangsform des Materials bezeichnet ([Gryczan 95]) und kann sondierenden oder modifizierenden Charakter aufweisen.

Eine sondierende Handlung an einem Material dient zur Gewinnung von Informationen, aus denen die handelnde Instanz in der Regel Parameter für eine spätere modifizierende Handlung an diesem oder einem anderen Material generiert. Es kommt zu einer zeitlichen Verzögerung (*timedelay*) zwischen Sondierung und Modifikation. Damit hat ein Arbeitsvorgang eine bestimmbare Dauer. Die Ausführung einer Handlung findet zu einem bestimmten Zeitpunkt statt und soll im folgenden im Gegensatz zum Arbeitsvorgang bzw. Arbeitsschritt als unteilbar und zeitverzugslos angenommen werden.

[30] vgl. den Begriff der Handlung in Unterabschnitt 3.2.1

Wie wir bereits in Kapitel 3 gesehen haben, ist das Konzept von Original und Kopie kein adäquates Mittel zur Modellierung einer gleichzeitig gemeinsamen Nutzung von Materialien. So ermöglicht das Konzept neben der sequentiellen gemeinsamen Nutzung von Materialien lediglich eine eingeschränkte Variante quasi-gleichzeitiger gemeinsamer Nutzung von Materialien und schränkt damit die inhärente Nebenläufigkeit bestehender Arten von gemeinsamer Nutzung in der fachlichen Welt ein.

> ,,The fact that people share tasks, work closely together in teams, know each other, exercise mutual solidarity, etc., is not only unreflected in current computer systems. It is often hampered[a] by the way the computer systems change the conditions of work."([Sörgaard 88])

[a] behindert

Ziel dieses Abschnitts soll es deshalb sein, Alternativen zu dem Konzept von Original und Kopie darzustellen sowie eine Reihe fachlicher Umgangsarten (Präsenz, Sicht, Kopie, Original) zu motivieren und diese in einem Konzept zur fachlichen Zugriffskontrolle (Konzept der Umgangsarten) zu vereinen.

4.2.1 Verfahren zur Synchronisation von Handlungen

In diesem Unterabschnitt wollen wir klären, welche Alternativen zum Konzept von Original und Kopie existieren. Da bis heute keine alternativen Verfahren zur fachlichen Nebenläufigkeitskontrolle formuliert worden sind, wollen wir im folgenden mehrere aus dem technischen Kontext entnommene Verfahren zur Nebenläufigkeitskontrolle kurz vorstellen, um auf diese Weise einen Überblick über bestehende Verfahren zur Synchronisation geben zu können.

Im folgenden stellen wir drei Verfahrensarten zur Synchronisation von Handlungen vor. Neben den *Sperrverfahren*[31] gehen wir dabei zum einen auf die *optimistischen Verfahren*[32] ein, die im fachlichen Kontext im Rahmen der gemeinsamen Nutzung von immateriellen Materialien[33] zum Einsatz kommen und damit auch im Bereich elektronischer Materialien Verwendung finden (zum Beispiel in Softwareentwicklungsumgebungen und CAD-Systemen)[34]. Bei diesen Verfahren werden gemeinsam genutzte Materialien dupliziert, so daß die handelnden Instanzen

[31] vgl. das Konzept von Original und Kopie

[32] das Konzept der Versionierung

[33] Neben materiellen Arbeitsgegenständen (zum Beispiel Mappen, Formulare) sind in der Arbeitsdomäne auch abstrakte, immaterielle Materialien (zum Beispiel Rendite, Zinssatz) zu finden. Auch diese Materialien ,,sind in der täglichen Arbeit durchaus gegenständlich. Sie können benannt werden und wir verbinden jeweils einen charakteristischen Umgang mit ihnen." ([Kilberth et al. 93])

[34] In diesem Zusammenhang ist anzumerken, daß materielle Arbeitsgegenstände durch die Umsetzung in Softwaresysteme elektronifiziert werden und damit immateriell sind. Durch diese

gleichzeitig und voneinander unabhängig auf einer Kopie des Materials arbeiten können. Erst bei der Aktualisierung des Originals werden Konflikte erkannt und beseitigt. Zum anderen stellen wir die sogenannten *Zeitstempelverfahren* vor, die ähnlich der Vergabe von Bearbeitungsnummern auf Ämtern und Behörden die anstehenden Arbeitsvorgänge zeitlich ordnen und damit eine Abarbeitungsreihenfolge festlegen. Das „Vordrängeln" kann erkannt und zurückgewiesen werden. Nach der eigentlichen Darstellung der Verfahren werden wir die vorgestellten Verfahren auf ihre fachliche Interpretierbarkeit und fachlich generelle Verwendbarkeit untersuchen.

Sperrverfahren

Ein Sperrverfahren (zum Beispiel: Original und Kopie) ist ein pessimistisches Verfahren, weil es von der Kollision nebenläufiger Handlungen als Regelfall ausgeht. Das Verfahren dient dazu, Konflikte bereits vor der Handlung an oder mit einem Material zu erkennen und damit Konflikte schon vor ihrer Entstehung zu verhindern. Dazu müssen sich die handelnden Instanzen vor dem Zugriff auf ein Material das Recht zur Durchführung von Sondierungen bzw. Modifikationen geben lassen, bevor sie sondierend oder modifizierend auf das Material zugreifen können. Zu diesem Zweck werden zwei Arten von Rechten (Sperren[35]) zur Verfügung gestellt:

1. *readlock bzw. shared lock:*
   Ein *readlock* wird vor einem ausschließlich sondierenden Zugriff auf ein Material gesetzt, um gleichzeitige, modifizierende Handlungen an oder mit dem Material auszuschließen. Das gleichzeitige Sondieren ist allerdings zugelassen, so daß gleichzeitig mehrere *readlock*-Sperren gesetzt werden können. Aus diesem Grund wird diese Sperrart auch als *shared lock* bezeichnet.

2. *writelock bzw. exclusive lock:*
   Der *writelock* wird vor einer modifizierenden Handlung an oder mit einem Material gesetzt, um gleichzeitige, sondierende bzw. modifizierende Handlungen auszuschließen. Der *writelock* kann deshalb nur dann gesetzt werden, wenn das zu sperrende Material nicht bereits durch einen *readlock* oder *writelock* gesperrt ist. Da der Zugriff exklusiv ist, wird diese Sperrart auch *exclusive lock* genannt.

Veränderung eröffnen sich neue Perspektiven im Umgang mit diesen Materialien, die vorher in der realen Anwendungsdomäne so nicht anzutreffen waren (vgl. [Sörgaard 88]). Diese Problemstellung ist nicht Teil dieser Arbeit, so daß auf eine tiefergehende Darstellung dieses Aspekts verzichtet werden soll.

[35] Im technischen Kontext werden die angeforderten und vergebenen Rechte auch als Sperren bezeichnet. Im folgenden wollen wir beide Begriffe daher interimistisch synonym verwenden, bevor wir im Abschnitt 4.2.2 den Begriff der Umgangsart einführen.

Im Gegensatz zum klassischen *writelock* gibt es auch eine weniger strenge Form dieser Sperrart, die das gleichzeitige Sondieren zuläßt und nur das gleichzeitige Modifizieren verbietet[36].

Sollte eine Sperre nicht gewährt werden, so muß die anfordernde handelnde Instanz warten, bis das Setzen der Sperre möglich wird. Dies kann auf unterschiedliche Weise geschehen:

- *aktives Warten (busy waiting)*:
  Die handelnde Instanz muß kontinuierlich die Sperre wieder anfordern (*polling*).

- *passives, synchrones Warten*:
  Die handelnde Instanz wartet solange, bis sie die Sperre erhält.

- *passives, asynchrones Warten*:
  Die handelnde Instanz ist zum Sperren angemeldet und kann die Ausführung anderer Handlungen vorziehen.

Neben dem unnötigen Overhead bei konfliktfreien Handlungen haben Sperrverfahren zwei weitere wesentliche Probleme. Zum einen kann ein Material unnötig gesperrt sein, wodurch sondierende bzw. modifizierende Handlungen an dem gesperrten Material durch andere handelnde Instanzen verzögert werden. Zum anderen besteht stets die Gefahr der Entstehung sogenannter *deadlocks*[37], die Folge zyklischer Abhängigkeiten zwischen den Arbeitsvorgängen mindestens zweier handelnder Instanzen sind.

Die Verwaltung der Sperren und die Erkennung von *deadlocks* ist Aufgabe eines Koordinators[38]. Dieser dient auch zur Beseitigung der *deadlocks*, für die es eine Reihe unterschiedlicher Verfahren gibt:

- *Pessimistische Prevention:*
  Bereits vor Beginn eines Arbeitsvorgangs muß eine handelnde Instanz alle Sperren setzen. Zudem ist während des Arbeitsvorgangs keine *lock promotion*[39] möglich. Die Folge dieses Verfahrens ist eine weitere Einschränkung der Nebenläufigkeit. Zudem ist die Menge der zu sperrenden Materialien vor

[36] vgl. hierzu das Konzept von Original und Kopie

[37] Ein *deadlock* (Verklemmung) ist die Situation, bei der mindestens zwei handelnde Instanzen auf die Freigabe von Sperren durch die jeweils andere handelnde Instanz warten.

[38] vgl. Lockmanager ([Coulouris et al. 94]) bzw. Materialkoordinator ([Wulf, Weske 94])

[39] Unter einer *lock promotion* wird die Transformation einer Sperre zu einer exklusiveren Sperre verstanden (zum Beispiel *readlock* zu *writelock*). Eine *lock promotion* ist nur dann möglich, wenn dadurch die Rechte anderer handelnder Instanzen nicht beeinträchtigt werden. So kann ein *readlock* nicht in einen *writelock* transformiert werden, wenn andere handelnde Instanzen ebenfalls *readlocks* auf das Material halten.

einem Arbeitsvorgang selten bekannt[40], so daß die pessimistische Prevention vor dem Hintergrund kooperativer qualifizierter menschlicher Tätigkeit wenig sinnvoll erscheint.

- *Optimistische Prevention:*
  Bei der optimistischen Prevention von *deadlocks* sucht ein Koordinator ununterbrochen nach zyklischen Abhängigkeiten zwischen nebenläufigen Arbeitsvorgängen. Wenn eine zyklische Abhängigkeit erkannt wurde, wird die *deadlock*-Situation aufgehoben, indem der Zyklus durch die Rücksetzung eines Arbeitsvorgangs aufgebrochen wird.

- *Timeout:*
  Jedem Arbeitsvorgang wird eine maximale Bearbeitungszeit (*timeout*) zugeordnet, nach deren Ablauf der Arbeitsvorgang abzubrechen ist. Das Problem dieses Verfahrens liegt in der Festlegung des *timeout*: Erstens ist die Zeit vor allem im Kontext interaktiver Anwendungen selten vorhersagbar, zweitens kann es durch die Vergabe gleicher *timeouts* wieder zu einer *deadlock*-Situation kommen und drittens besteht die Möglichkeit, daß ein Arbeitsvorgang abgebrochen wird, obwohl kein *deadlock* vorliegt.

Im allgemeinen unterscheiden sich die Sperrverfahren hinsichtlich ihrer Sperrgranularität. So gibt es zum Beispiel Verfahren, die eine Menge unterschiedlicher hierarchischer Sperrgranulate zur Verfügung stellen, aus der nach Bedarf ausgewählt werden kann[41]. Durch die Wahl eines adäquaten Sperrgranulats kann der Overhead des Sperrverfahrens reduziert werden, weil zum Beispiel statt mehrerer fein granularer Sperren eine gröbere gewählt werden kann. Die Wahl einer gröberen Sperre führt zur Reduzierung der Anzahl der Sperren bei gleichzeitiger Erhöhung der Anzahl gesperrter Materialien. Daneben wurden Ansätze zur automatischen Sperreskalation (selbständige Vergröberung des Sperrgranulats) formuliert (vgl. [Herrmann et al. 89]).

Optimistische Verfahren

Die Grundannahme sogenannter optimistischer Sperrverfahren ist die Hypothese, daß die gemeinsame Nutzung von Materialien selten zu Konflikten führt. Deshalb dürfen alle Arbeitsvorgänge bis zu ihrer Terminierung ungestört ausgeführt werden. Erst dann wird in einer Validierungsprozedur überprüft, ob Serialisierbarkeit[42] und damit Konfliktfreiheit gegeben ist. Liegt ein Konflikt vor, muß dieser durch Rücksetzung von Arbeitsvorgängen aufgelöst werden.

---

[40] vgl. hierzu den Umgang mit interaktiven Anwendungen
[41] vgl. ORACLE 7.1 ([Stürner 93])
[42] vgl. [Jessen, Valk 87]

Damit gliedert sich bei der Nutzung eines optimistischen Verfahrens ein Arbeitsvorgang in drei Phasen:

1. *Phase: Lesen*
   Vor jeder Handlung an oder mit einem Material ist das Material zu duplizieren. Die so erzeugten Materialkopien können dann beliebig modifiziert bzw. sondiert werden. Dabei kommt es bei nebenläufigen Arbeitsvorgängen zu einer Koexistenz mehrerer Materialkopien des Materials.

2. *Phase: Validierung*
   Bevor das Materialoriginal wieder durch Vergleich mit den Materialkopien aktualisiert werden kann (*update*), wird geprüft, ob es einen Konflikt mit anderen Arbeitsvorgängen gibt:

   - In dem Arbeitvorgang darf kein Material sondiert worden sein, das in einem anderen Arbeitsvorgang bereits modifiziert worden ist.
   - In anderen Arbeitvorgängen darf kein Material sondiert worden sein, das in dem zu validierenden Arbeitsvorgang modifiziert worden ist.
   - In zwei Arbeitvorgängen darf nicht dasselbe Material modifiziert worden sein.

3. *Phase: Schreiben*
   Nach erfolgter positiver Validierung erfolgt die Aktualisierung der Materialoriginale durch die modifizierten Materialkopien.

Weil durch den Einsatz optimistischer Verfahren im Konfliktfall die Rücksetzung langer Arbeitsvorgänge erforderlich werden kann, wodurch die Arbeit von mehreren Tagen ungeschehen gemacht wird, bieten sich diese Verfahren nur dann an, wenn Konflikte eher die seltene Ausnahme als die Regel sind.

Zeitstempelverfahren

Beim Einsatz von Zeitstempelverfahren wird jedem Arbeitsvorgang und jedem Material ein sogenannter Zeitstempel (*timestamp*) zugeordnet, der den Zeitpunkt der Ausführung (Arbeitsvorgang) oder den Zeitpunkt der letzten Modifikation bzw. Sondierung (Material) modelliert. Die Zeitstempel legen damit die zeitliche Ordnung der Arbeitsvorgänge fest[43] und werden entweder in Form eines heraufzählenden Zählers oder aber durch eine hinreichend genaue[44] Diskretisierung der kontinuierlichen Realzeit bestimmt.

Bevor eine handelnde Instanz einen modifizierenden oder sondierenden Zugriff auf ein Material durchführen kann, wird zunächst anhand der folgenden Regeln geprüft, ob eine in den Arbeitsvorgang eingebettete Handlung „nicht zu spät ist":

[43] vgl. hierzu die Vergabe von Bearbeitungsnummern in Behörden.
[44] Es darf nicht zu Gleichzeitigkeit kommen.

- In einem Arbeitsvorgang darf kein Material sondiert werden, dessen Zustand bereits in einem späteren Arbeitsvorgang modifiziert wurde.

- In einem Arbeitsvorgang darf kein Material modifiziert werden, dessen Zustand bereits in einem späteren Arbeitsvorgang modifiziert oder sondiert worden ist.

Trifft eine dieser Regeln nicht zu, so wird der Arbeitsvorgang als ,,verspätet" bezeichnet und ist abzubrechen. Damit kann sichergestellt werden, daß handelnde Instanzen nur auf die Beendigung früherer Arbeitsvorgänge warten. Auf diese Weise kann die Bildung von zyklischen Abhängigkeiten zwischen den Arbeitsvorgängen und damit die Entstehung von *deadlock*-Situationen verhindert werden.

Durch Zeitstempelverfahren wird nur dann die Nebenläufigkeit sequentialisiert, wenn es zu Konflikten kommt, andernfalls ist die nebenläufige Nutzung von Materialien uneingeschränkt möglich. Ein Nachteil dieser Verfahren ist der im Gegensatz zu Sperrverfahren und optimistischen Verfahren hohe Aufwand. Es ist vor jeder Ausführung einer Handlung zu prüfen, ob sie ,,nicht zu spät" und damit gültig ist.

Bewertung

Bei der Bewertung der dargestellten Verfahren zur Synchronisation von Handlungen wollen wir von den Gegebenheiten einer technischen Realisierung abstrahieren und uns ausschließlich auf die Untersuchung der fachlich generellen Verwendbarkeit der Verfahren beschränken.

Die optimistischen Verfahren sind, wie wir bereits in der Einleitung dieses Unterabschnitts angemerkt haben, für die Synchronisation von Handlungen an oder mit immateriellen Materialien geeignet. Eine Duplizierung materieller Gegenstände vor jeder Handlung an oder mit ihnen erscheint fachlich nicht adäquat. Zudem ist es längst nicht in jedem fachlich spezifischen Kontext wünschenswert, daß Zugriffskonflikte erst am Ende der Arbeitsvorgänge erkannt werden. Ein solches Vorgehen kann dazu führen, daß die Arbeit mehrerer Wochen überflüssig wird und verloren geht. Damit sind optimistische Verfahren für die Synchronisation von Handlungen, die in lange Arbeitsvorgänge eingebettet sind, ungeeignet.

Sowohl die optimistischen Verfahren als auch die Zeitstempelverfahren sind abhängig von dem Konzept des Arbeitsvorgangs, der explizit durch die handelnde Instanz zu starten und zu beenden ist. Damit sind diese Verfahren in Kontexten, in denen handelnde Instanzen losgelöst von explizit festgelegten Arbeitsvorgängen agieren, nicht einsetzbar. Da das Konzept des Arbeitsvorgangs nach

unserer Ansicht fachlich spezifischer Natur ist, entziehen sich nicht nur die optimistischen Verfahren, sondern auch die Zeitstempelverfahren einer fachlich generellen Verwendbarkeit[45].

Sperrverfahren haben im Gegensatz zu den optimistischen Verfahren und den Zeitstempelverfahren am ehesten einen fachlich generellen Charakter: Sie können losgelöst von Arbeitsvorgängen angewendet werden und entsprechen dem intuitiven Umgang mit Materialien in der fachlichen Welt. Dieses liegt vor allem daran, daß Sperrverfahren ein rudimentäres Konzept von Besitz realisieren. Damit ist es einer handelnden Instanz möglich, zu erkennen, daß eine andere handelnde Instanz ein bestimmtes Material bereits in Bearbeitung hat oder nicht. Somit bieten sich Sperrverfahren als Grundlage eines fachlich generellen Konzepts zur Synchronisation nebenläufiger Handlungen an. Da Sperrverfahren die Nebenläufigkeit zum Teil stark einschränken[46] und zudem die Modellierung (quasi-)gleichzeitiger gemeinsamer Nutzung auf Basis von Sperrverfahren schwierig ist[47], wollen wir im folgenden Unterabschnitt ein fachlich generelles Konzept zur Synchronisation vorstellen, das auf Sperrverfahren basiert, aber zusätzliche Konzepte (zum Beispiel die aktuelle Sicht auf ein Material) bereitstellt.

4.2.2 Das Konzept von Original, Kopie, Sicht und Präsenz

Wie wir im vorhergehenden Unterabschnitt gesehen haben, ist keines der vorgestellten Verfahren ausreichend, und es bedarf analog zur Nebenläufigkeitskontrolle im technischen Kontext eines hybriden Verfahrens[48]. Bevor wir zur Beschreibung unseres Ansatzes kommen, wollen wir zunächst die Anforderungen, die wir an ein solches Verfahren stellen, formulieren.

> „However, .. [multi-user] applications have two new requirements relating to concurrency control: (i) users require immediate notification of changes made by other users - which is contrary to the idea of isolation, (ii) users need to be able to access data items before other users have completed their transactions - which has lead to the development of new types of locks that trigger actions when data items are accessed." ([Coulouris et al. 94])

[45] Mit dieser Bewertung wollen wir keineswegs ausdrücken, daß diese Verfahren nicht in fachlichen Kontexten einsetzbar sind. Wir sprechen ihnen lediglich ihre fachlich generelle Einsatzfähigkeit ab.

[46] vgl. Darstellung des Konzepts von Original und Kopie in den Unterabschnitten 3.3.2 und 3.5.2

[47] vgl. Unterabschnitt 3.5.2

[48] Das OODBS GemStone bietet beispielsweise neben einer optimistischen Nebenläufigkeitskontrolle die Möglichkeit einzelne Objekte zu sperren, um auf diese Weise die Entstehung unterschiedlicher Objektversionen und die damit verbundenen Konflikte beim Einchecken der Objektversionen vermeiden zu können ([Heuer 92]).

Ein fachlich generelles Verfahren zur Synchronisation von Handlungen sollte Mechanismen bereitstellen, mit deren Hilfe im Fall einer Modifikation andere handelnde Instanzen benachrichtigt werden können. Darüber hinaus sollte es möglich sein, daß eine handelnde Instanz eine aktuelle Sicht auf ein Material haben kann, an der alle Modifikationen, die an dem Material vollzogen werden, unmittelbar nachvollzogen werden. Die Bereitstellung einer solchen Sicht würde im Gegensatz zum Konzept von Original und Kopie eine erweiterte quasi-gleichzeitige gemeinsame Nutzung von Materialien erlauben, weil sondierende Handlungen an dieser Sicht im Vergleich zum Umgang mit nicht aktuellen Kopien keine veralteten Resultate liefern.

Bei der (quasi-)gleichzeitigen gemeinsamen Nutzung von Materialien durch mehrere verschiedene handelnde Instanzen ist es erforderlich, die unterschiedliche Sensibilität im Umgang mit Materialien zu berücksichtigen. Es gibt beispielsweise Materialien, deren Existenz lediglich einer Auswahl handelnder Instanzen bekannt sein sollte (zum Beispiel Tagebuch, Liebesbrief etc.), deren Zustand für sondierende Instanzen stets als konsistent gesehen werden sollte (zum Beispiel Konto) oder bei denen ein semikonsistenter[49] Zustand ausreichend ist (zum Beispiel nicht komplett ausgefüllter Kreditantrag). Wieder andere Materialien sind einzigartig und sollten nicht kopierbar sein (zum Beispiel Terminkalender) oder unterliegen keiner Beschränkung und sind daher beliebig sondierbar, modifizierbar und kopierbar (zum Beispiel ein Stück Papier).

Im folgenden wollen wir das Konzept von Präsenz, Sicht, Kopie und Original, das wir in der weiteren Diskussion als das Konzept der Umgangsarten bezeichnen wollen, darstellen. Eine handelnde Instanz hat eine Umgangsart mit einem Material und kann es gemäß der mit der Umgangsart verbundenen Rechte (zum Beispiel Recht zur Durchführung von Modifikationen) bearbeiten. Umgangsarten sind durch die handelnde Instanz bei einer zusätzlichen Instanz, die zur Verwaltung und Koordination der Umgangsarten dient, explizit zu akquirieren bzw. freizugeben. Die Grundlage für das Konzept der Umgangsarten bildet das Konzept von Original und Kopie[50]. Neben der Erweiterung um die Umgangsarten Sicht und Präsenz wird der Kopiebegriff überarbeitet und präzisiert sowie der Begriff des Originals hinsichtlich der unterschiedlichen Materialsensibilitäten differenziert.

- *Präsenz*:
  Mit *Präsenz* bezeichnen wir das Wissen über die Existenz eines Materials. Einer handelnden Instanz ist bekannt, daß ein Material mit einer bestimmten Materialidentität existiert.

[49] Ein Material hat einen semikonsistenten Zustand, wenn es keine unbemerkten fachlichen Inkonsistenzen enthält (nach [Leikauf 89]).

[50] vgl. die Unterabschnitte 3.3.2 und 3.5.2

- *Sicht*:
  Der Inhaber einer *Sicht* auf ein Material kann das Original „sehen" und hat damit das Recht, ausschließlich sondierend auf das Materialoriginal zuzugreifen. Sichten haben keine eigene Identität und sind durch den Sichtinhaber nicht modifizierbar. Am Original vollzogene Änderungen werden unmittelbar und zeitverzugslos[51] an der Sicht nachvollzogen. Damit stellen Sichten ein Schlüsselkonzept zur Realisierung von WYSIWIS-Systemen dar[52].

- *Kopie*:
  Der Begriff der *Kopie* unterscheidet sich von dem in [Wulf, Weske 94] geprägten Begriff der Kopie[53]. Das Materialoriginal wird asynchron repliziert (vgl. Erstellung einer Materialkopie), so daß eine Kopie erzeugt wird, die wieder ein eigenständiges Material ist. Diese Materialkopie hat damit eine eigene Identität und ist prinzipiell modifizierbar.

Auch wenn das Kopieren eines Materials zu einem neuen Materialoriginal führt, kann die Kopie auch als eine Umgangsart aufgefaßt werden, die eine handelnde Instanz mit dem kopierten Materialoriginal hat. Der Grund dafür liegt darin, daß eine Materialkopie im fachlichen Kontext auch als Stellvertreter für das Materialoriginal, dessen Duplizierung zu der Materialkopie geführt hat, betrachtet werden kann.

Eine Sicht eignet sich immer dann, wenn eine zeitgenaue Darstellung von Modifikationen erforderlich ist. Dabei ist zu beachten, daß ein Anwender bei zu vielen Sichten leicht den Überblick verlieren kann. Wenn eine gewisse „Unschärfe" des Replikats bezüglich des Originals erlaubt ist, kann die Umgangsart Kopie mit dem Material akquiriert werden.

Aufgrund der Einführung der Umgangsarten Sicht, Präsenz und Kopie muß das Konzept des Originals differenziert werden. Der Grund dafür liegt in den oben motivierten, unterschiedlichen Sensibilitäten im Umgang mit Materialien. So sollte zum Beispiel, während ein Bankangestellter an einem Konto eine Änderung vornimmt, kein anderer Bankangestellter eine Sicht auf das zwischenzeitlich inkonsistente Konto nehmen können. In diesem Zusammenhang wollen wir feststellen, daß die Sensibilität des Umgangs nicht durch das Material determiniert wird. So kann beispielsweise die Existenz eines Tagebuchs durchaus mehreren Personen bekannt sein, ein nicht adäquater Umgang mit Konten zu Inkonsistenzen führen

[51] In der realen Welt erfolgt die Änderung der Sicht in Lichtgeschwindigkeit.

[52] Auf der technischen Ebene wird eine Sicht durch synchrone Replikation eines Objekts realisiert, wobei die nachzuvollziehenden Änderungen am Replikat mit Hilfe eines Triggermechanismus angestoßen werden können.

[53] vgl. Unterabschnitt 3.5.2

oder die Entwicklung von Know-how zur Kopierbarkeit fälschungssicherer Personalausweise führen.

Wir haben die folgenden unterschiedlichen Umgangsarten, die auf dem in der Arbeit [Wulf, Weske 94] geprägten Original-Begriff basieren, identifiziert:

- *Original-private:*
  Die Umgangsart *Original-private* modelliert den nicht öffentlichen Umgang mit Materialien der Privatsphäre (zum Beispiel Tagebuch). Der Inhaber der Umgangsart Original-private kann das Material ohne Einschränkungen bearbeiten. Andere handelnde Instanzen sind nicht in der Lage, die Umgangsarten Original[54], Präsenz, Kopie oder Sicht mit dem Material zu akquirieren.

- *Original-protected:*
  Die Umgangsart *Original-protected* modelliert den Umgang mit öffentlichen, immer konsistenten Materialien (zum Beispiel Konto). Der Inhaber der Umgangsart Original-protected kann das Material ohne Einschränkungen bearbeiten. Andere handelnde Instanzen sind in der Lage, die Umgangsart Präsenz mit dem Material zu akquirieren. Die Umgangsarten Original, Kopie und Sicht werden ihnen nicht gewährt.

- *Original-unique:*
  Die Umgangsart *Original-unique* modelliert den Umgang mit öffentlichen, aber nicht kopierbaren Materialien (zum Beispiel Terminkalender). Der Inhaber der Umgangsart Original-unique kann das Material beliebig sondieren und modifizieren, aber nicht kopieren. Andere handelnde Instanzen können lediglich die Umgangsarten Präsenz und Sicht mit dem Material akquirieren.

- *Original-public:*
  Die Umgangsart *Original-public* modelliert den Umgang mit öffentlichen Materialien. Der Inhaber der Umgangsart Original-public kann das Material ohne Einschränkungen bearbeiten. Andere handelnde Instanzen sind in der Lage, die Umgangsarten Präsenz, Kopie oder Sicht mit dem Material zu akquirieren.

Jede Original-Umgangsart kann beliebig durch den jeweiligen Originalinhaber während der gesamten Lebensdauer des Materials verändert werden. Im folgenden wollen wir den Originalinhaber als Besitzer des Materials bezeichnen. Bereits bei der Erzeugung eines Materials kann eine initiale Umgangsart gesetzt werden.

[54] Unter dem Begriff Original fassen wir die Umgangsarten Original-private, Original-protected, Original-unique und Original-public zusammen.

Zudem ist es möglich, daß keine handelnde Instanz das Material in einer Umgangsart hält.

Da die dargestellten Original-Umgangsarten von unten nach oben sensibler und damit strenger werden, stellt die Umgangsart Original-private die strengste Umgangsart dar. Weil sie ein rudimentäres Konzept von Eigentum unterstützt, kann der Begriff des Eigentums auf der fachlich generellen Ebene ausgedrückt werden. Hat eine handelnde Instanz die Umgangsart Original-private mit einem Material akquiriert, so ist sie nicht nur Besitzer, sondern gleichzeitig auch Eigentümer[55] des Materials. Ausgereiftere Konzepte von Besitz und Eigentum, die zum Beispiel auch die Zuordnung von Besitz und Eigentum zu unterschiedlichen handelnden Instanzen erlauben, sind domänenabhängig und damit fachlich spezifischer Natur.

Eine Übersicht über die Verträglichkeiten der einzelnen Umgangsarten kann der Abbildung 4.1 entnommen werden.

Angeforderte Umgangsart	Bestehende Umgangsart				
	kein Original_xxx	Original_public	Original_unique	Original_protected	Original_private
Präsenz	J	J	J	J	B
Sicht	J	J	J		B
Kopie	J	J			B
Original_public	J	*	B	B	B
Original_unique	J	B	*	B	B
Original_protected	J	B	B	*	B
Original_private	J	B	B	B	*

Legende:	
J	Jedem Anwender kann die gewünschte Umgangsart gewährt werden.
B	Nur dem Besitzer kann die gewünschte Umgangsart gewährt werden.
*	Besitzer erhält Benachrichtigung über bestehenden Besitz.

Abbildung 4.1: Übersicht über die Verträglichkeiten der Umgangsarten

Bei der Betrachtung der Granularität der Umgangsarten bewegt man sich innerhalb eines Spannungsfeldes: Je gröber die Granularität, desto eingeschränkter ist

[55] Die Begriffe Besitz und Eigentum sind zu unterscheiden: „Der Eigentümer hat ein Recht an einer Sache, der Besitzer hat tatsächl. Gewalt über sie. Eigentum und .. [Besitz] können, aber müssen sich nicht in derselben Hand vereinigen." ([Universal 82])

die Nebenläufigkeit. Je feiner das Granulat, desto größer wird die Zahl der Akquisitionen und damit der Aufwand für die Synchronisation (*overhead*). Die optimale Granularität ist eigentlich nur durch die handelnde Instanz selbst bestimmbar ([Herrmann et al. 89]). Ein Synchronisationsverfahren für komplexe Materialien sollte daher unterschiedliche Granulate bereitstellen, so daß bei geeigneter Wahl nicht das ganze komplexe Material „gesperrt" werden muß und somit eine quasigleichzeitige gemeinsame Nutzung des komplexen Materials möglich ist.

Analog zur flachen und tiefen Materialgleichheit, -kopie, -erzeugung und -auflösung führen wir die flache und tiefe Umgangsartakquisition ein:

Begriff 33 : *Flache Umgangsartakquisition*
Bei der flachen Umgangsartakquisition erfolgt ausschließlich die Akquisition einer gewählten Umgangsart mit dem (Kontext-)Material. Umgangsarten mit den dem (Kontext-)Material zugeordneten Komponentenmaterialien und den Materialien, die mit dem (Kontext-)Material verknüpft sind, werden nicht akquiriert.

Begriff 34 : *Tiefe Umgangsartakquisition*
Bei der tiefen Umgangsartakquisition erfolgt die flache Akquisition der gewählten Umgangsart mit dem (Kontext-)Material und die tiefe Akquisition der gleichen Umgangsart mit den dem (Kontext-)Material zugeordneten Komponentenmaterialien. Umgangsarten mit den Materialien, die mit dem (Kontext-)Material verknüpft sind, werden nicht akquiriert.

In der Umkehrung zur Umgangsartakquisition definieren wir die Umgangsartfreigabe:

Begriff 35 : *Flache Umgangsartfreigabe*
Bei der flachen Umgangsartfreigabe erfolgt ausschließlich die Freigabe der Umgangsart mit dem (Kontext-)Material. Bestehende Umgangsarten mit den dem (Kontext-)Material zugeordneten Komponentenmaterialien und mit den Materialien, die mit dem (Kontext-)Material verknüpft sind, bleiben erhalten.

Begriff 36 : *Tiefe Umgangsartfreigabe*
Bei der tiefen Umgangsartfreigabe erfolgt die flache Freigabe der Umgangsart mit dem (Kontext-)Material und die tiefe Freigabe der Umgangsarten mit den dem (Kontext-)Material zugeordneten Komponentenmaterialien. Bestehende Umgangsarten mit den Materialien, die mit dem (Kontext-)Material verknüpft sind, bleiben erhalten.

Durch diese Definitionen der Umgangsartakquisition bzw. der Umgangsartfreigabe wird es möglich, daß eine handelnde Instanz beliebige (auch unterschiedliche) Umgangsarten mit dem Kontextmaterial und den Komponentenmaterialien eines komplexen Materials halten kann. Dies ist insbesondere im Hinblick auf die quasi-gleichzeitige gemeinsame Nutzung eines Materials interessant. Zum Beispiel können die einzelnen Mitglieder eines Autorenteams bestimmte Kapitel eines Buchs in der Umgangsart Original bearbeiten, während sie andere Kapitel in der Umgangsart Sicht halten und damit Modifikationen, die dort von anderen Mitgliedern des Teams vorgenommen werden, unmittelbar erkennen[56].

Wie wir oben gesehen haben, besteht beim Einsatz von Sperrverfahren und damit auch bei der Nutzung des Konzepts der Umgangsarten die Gefahr von *deadlock*-Situationen. Die genannten Verfahren zur *deadlock*-Prevention (pessimistische Prevention, optimistische Prevention und Timeout) sind nicht fachlich motivierbar, da sie selbst die Wahl treffen, welcher Arbeitsvorgang abzubrechen ist. Zudem kann es zum Abbruch von Arbeitsvorgängen kommen, obwohl kein *deadlock* vorliegt. Aus diesen Gründen favorisieren wir die explizite *deadlock*-Auflösung in Folge einer Diskussion und Konsensbildung der handelnden Instanzen. Zu diesem Zweck ist auf der fachlich spezifischen Ebene ein Verfahren anzubieten, das eine solche explizite Koordination (zum Beispiel per e-mail) ermöglicht.

Ein Verfahren, das zur expliziten *deadlock*-Auflösung auf der fachlich spezifischen Ebene eingesetzt werden kann, ist die sogenannte *schwache Konsistenz* ([Arens 93]). Dieses Verfahren unterstützt die Abstimmung mit anderen handelnden Instanzen, indem vor der Ausführung einer Modifikation den anderen handelnden Instanzen ein Vorschlag unterbreitet wird, den diese annehmen oder ablehnen bzw. durch einen Gegenvorschlag beantworten können[57].

4.3 Der Materialraum als Basisabstraktion

Dieser Abschnitt beschäftigt sich mit dem fachlichen Basiskonzept des *Materialraums*. Dieser ist aus der Revision der Konzepte zur Materialverwaltung aus [Wulf, Weske 94], [Riehle 95] und [Gryczan 95] hervorgegangen und kann nach unserer Ansicht, die vorgestellten Konzepte zur Materialverwaltung substituieren.

Der Begriff des Raums

Der Begriff des Raums wird in den unterschiedlichen Disziplinen verschieden gefaßt. In [Brockhaus 81], [Universal 79] und [Universal 82] werden sie wie folgt dargestellt:

[56] vgl. hierzu [Sörgaard 88]

[57] vgl. hierzu den Sprechakttyp als Dimension der Kooperation

1. *Philosophie*:
   In der Philosophie ist der Raum einer der grundlegenden Begriffe für eine Beschreibung der Natur. In der abendländischen Antike wurde der Raum als reale endliche Erscheinung gedeutet. Aristoteles versuchte sich dem Begriff des Raums mit Hilfe des Begriffs des Orts zu nähern. Nach seiner Auffassung wird ein Ort durch die Grenzen eines Körpers bestimmt. Durch die Orte ergibt sich eine gestufte Raumordnung des Universums, dessen Raum als endlich gedacht wird. In der Renaissance setzte sich immer mehr die Vorstellung eines absoluten, ruhenden, homogen unendlichen (auch leeren) Raums durch (Galileo Galilei und Isaac Newton). Immanuel Kant war der Überzeugung, daß der Raum nichts anderes als eine notwendige Form unserer Anschauung ist, die den Dingen selbst nicht zugehört.

2. *Mathematik*
   Der mathematische Raumbegriff leitet sich wie der philosophische von der intuitiven Raumvorstellung ab. Dabei wird der gewöhnliche dreidimensionale Raum durch die dreidimensionale euklidsche Geometrie beschrieben. Der Raum setzt sich aus Raumpunkten zusammen, deren Position im Raum durch die Angabe eines Zahlentripels (Länge, Breite, Höhe) festgelegt wird. In der modernen Mathematik wird der Raumbegriff für unterschiedliche Arten von konstruktiven Bereichen, denen eine Menge von Axiomen zugrundeliegen, verwendet.

3. *Physik*
   Aus der Geometrie hat sich die Vorstellung vom physikalischen Raum entwickelt, in dem sich die physikalischen Körper aufhalten oder bewegen. Zu einem bestimmten Zeitpunkt haben alle Körper im Raum einen bestimmbaren Abstand voneinander. Der leer gedachte physikalische Raum ist isotrop, zusammenhängend und kontinuierlich. Somit gibt es keine Stelle, an der der Raum aufhört, um an anderer Stelle von neuem zu beginnen. Die klassische Raumvorstellung wurde durch das relativistische Raumverständnis in der allgemeinen und in der speziellen Relativitätstheorie (Albert Einstein) abgelöst. Nach dieser Theorie ist der Raum nicht unendlich, sondern unbegrenzt. Der dreidimensionale Raum ist in einer vierten Dimension gekrümmt und endlich.

4. *Psychologie*
   Der psychologische Raum beschreibt die gegliederte Gesamtheit (Ordnung) der dreidimensional erfaßten Erscheinungsgegenstände in ihrem Bezug zueinander und zum wahrnehmenden Subjekt. Das Gefüge einer geordneten Raumstruktur ergibt sich dabei aus Wahrnehmungseindrücken und kognitiven Prozessen.

Bereitgestellte fachlich generelle Konzepte

Die in Kapitel 3 beschriebenen Konzepte zur Materialverwaltung dienen ausschließlich dazu, die aus der Datenbank ausgecheckten Materialien zu verwalten und auf Anforderung den Werkzeugen bereitzustellen sowie die ausgecheckten Materialien wieder in die Datenbank einzuchecken. Das Konzept des Materialraums hingegen verwaltet keinen technischen Status und stellt nach außen statt einer technisch motivierten Schnittstelle rein fachlich generelle Konzepte (zum Beispiel Umgangsarten) zur Verfügung. Es kann direkt auf die Materialien im *Materialraum* zugegriffen werden, so daß der Umweg über eine Materialverwaltung nicht erforderlich ist.

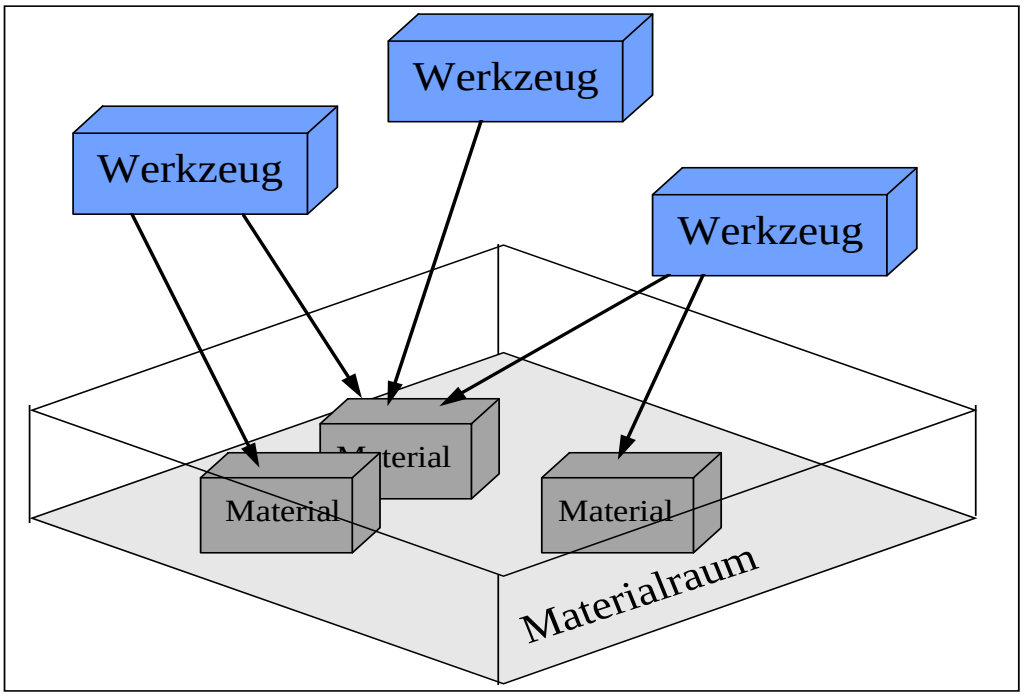


Abbildung 4.2: Der Materialraum

Begriff 37 : *Materialraum*
Der Materialraum ist ein nicht partitionierter, entfernungsloser, unbegrenzter Bereich, der alle existierenden Materialien enthält und diese zugreifbar macht. Alle Materialien liegen nach ihrer Erzeugung dauerhaft und unabhängig von ihrem Zustand sowie dem Status einer sie benutzenden Instanz im Materialraum. Sie werden aus diesem erst bei ihrer expliziten Auflösung entfernt.

Damit hat der Materialraum drei Aufgaben:

1. *Abstraktion von den technischen Komponenten*:
   Der Umgang mit technischen Komponenten erfolgt technisch transparent mit Hilfe fachlich genereller Konzepte. Jede handelnde Instanz sieht prinzipiell denselben Materialraum und geht, soweit die Rechte anderer handelnder Instanzen dadurch nicht beeinträchtigt werden, mit den Materialien und dem Materialraum in der gleichen Weise um.

2. *Direkte Zugreifbarkeit der Materialien*:
   Jedes im Materialraum befindliche Material ist eindeutig identifizierbar und unabhängig von technischen Gegebenheiten im Materialraum direkt zugreifbar. Der Materialraum stellt dafür einen Mechanismus zur Adressierung von Materialien bereit.

3. *Bereitstellung des Konzepts der Umgangsarten*:
   Beliebig viele handelnde Instanzen können den Materialraum und die in ihm verwahrten Materialien benutzen. Dazu können die handelnden Instanzen zu ihrer Tätigkeit adäquate Umgangsarten mit Materialien akquirieren. Damit verwaltet der Materialraum die Umgangsarten und sorgt somit für eine implizite Synchronisation der Handlungen an oder mit den Materialien im Materialraum. Zudem ist der Materialraum von den handelnden Instanzen beobachtbar, so daß diese „sehen" können, wenn eine andere handelnde Instanz

   - eine Umgangsart mit einem bestimmten Material akquiriert,
   - eine Umgangsart mit einem bestimmten Material freigibt,
   - ein bestimmtes Material modifiziert oder
   - ein bestimmtes Material auflöst.

Neben diesen direkt vom Materialraum bereitgestellten Diensten dient der Materialraum als Basis für aufbauende fachlich spezifische Konzepte, wie zum Beispiel die Begriffe Ort[58] und Lokalität. Diese Konzepte sind fachlich spezifischer Natur, weil sie, obwohl sie überall anzutreffen sind, in ihrer Ausprägung äußerst unterschiedlich sein können. Aus diesem Grund modellieren wir den Materialraum entfernungslos, so daß der Materialraum keine Einschränkungen vorgibt und beliebige fachlich spezifische Raumkonzepte wie zum Beispiel die Umgebung[59], die Arbeitsumgebung[60] oder die Raummetapher[61] losgelößt von technischen Details

[58] vgl. hierzu den philosophischen Raumbegriff
[59] vgl. [Riehle 95]
[60] vgl. [Gryczan 95]
[61] Henning Wolf und Stefan Roock, Fachbereich Informatik an der Universität Hamburg, Arbeitsbereich Softwaretechnik

auf dem Materialraum basierend realisiert werden können. Diese fachlich spezifischen Konzepte modellieren den philosophischen und psychologischen Raumbegriff unter besonderer Berücksichtigung soziologischer Gesichtspunkte.

Die fachlich generelle Transparenz

Damit läßt sich die durch den Materialraum bereitgestellte *fachlich generelle Transparenz* wie folgt begrifflich fassen[62]:

- *Ortstransparenz:*
  Alle Materialien befinden sich in einem entfernungslosen Materialraum.

- *Zugriffstransparenz:*
  entfällt (vgl. Ortstransparenz)

- *Migrationstransparenz:*
  entfällt (vgl. Ortstransparenz)

- *Nebenläufigkeitstransparenz:*
  Handelnde Instanzen können nebenläufige Handlungen und deren Synchronisation sehen.

- *Replikationstransparenz:*
  Materialien sind prinzipiell[63] kopierbar. Jede Materialkopie ist wieder ein eigenständiges Materialoriginal mit eigener Materialidentität.

- *Fehlertransparenz:*
  Systembedingte Einschränkungen der technischen Komponenten werden gegenüber der handelnden Instanz im fachlichen Kontext auf Basis fachlich genereller Konzepte ausgedrückt (zum Beispiel „Material kann im Materialraum nicht gefunden werden.").

Der Materialraum stellt somit eine fachlich generelle Abstraktion der technischen Komponenten dar, dient als Bindeglied (*gateway*) zwischen fachlichen und technischen Komponenten und bildet damit aus der fachlichen Sicht den unteren Abschluß. Das Konzept des Materialraums erfüllt damit die in Kapitel 3 genannten Anforderungen an eine fachlich generelle Abstraktion zur Kapselung der technischen Welt, so daß der Materialraum als ein fachliches Basiskonzept in die Entwicklung beliebiger fachlich spezifischer Anwendungen einfließen kann[64].

[62] in Anlehnung an [Gryczan 95]

[63] Eine Ausnahme bilden die Materialien, die in der Umgangsart *Original-unique* gehalten werden.

[64] Die Konzepte zur technischen Realisierung des Materialraums werden in den folgenden Kapiteln dargestellt.

Aufbauende Konzepte

Um das Konzept des Materialraums bezüglich fachlich spezifischer Konzepte einzuordnen, wollen wir im folgenden drei fachlich spezifische Konzepte, die basierend auf dem Materialraum modellierbar sind, kurz darstellen:

1. *Arbeitsvorgänge und Arbeitsschritte*:
   Ein Arbeitsvorgang, der sich in Arbeitsschritte aufteilen kann, dient zur expliziten Synchronisation nebenläufiger Handlungen und stellt ein Mittel zur fachlichen Fehlerbehebung (Recovery) bereit. Arbeitsvorgänge sind somit das fachliche Äquivalent technischer Transaktionen[65].

2. *Beweglichkeit*:
   Da der Materialraum entfernungslos ist und damit kein Konzept fachlicher Lokalität von Materialien unterstützt wird, gibt es auf der fachlich generellen Ebene keine Beweglichkeit von Materialien. Aus diesem Grund müssen die Konzepte der Lokalität, der Entfernung und damit auch der Beweglichkeit auf der fachlich spezifischen Ebene beispielsweise basierend auf dem Begriff der Arbeitsumgebung ([Gryczan 95]) modelliert werden[66]. Der Materialraum kann bei der Modellierung fachlicher Orte einen wichtigen Beitrag leisten, indem er von den Orten auf der technischen Ebene (Hauptspeicher, Datenbank, Netzwerk) abstrahiert und damit die fachlich spezifischen Orte (zum Beispiel Arbeitsplätze in einer Automobilfabrik oder Archivschränke in einem Büro) losgelöst von den technischen Komponenten modelliert werden können.

3. *Besitz und Eigentum*:
   Im fachlich spezifischen Kontext könnte das durch die Umgangsarten rudimentär bereitgestellte Konzept von Besitz und Eigentum verfeinert werden. So könnten materialgebundene Zugriffsrechte (Gruppenrechte, Autorisierung) realisiert werden und damit auch für Materialien Datenschutzaspekte berücksichtigt werden. In einer zu dieser Arbeit parallel entstehenden Diplomarbeit[67] werden daher Zugriffsrechte auf Materialien an Räume gebunden. Anwender können, wenn sie eine Zutrittsberechtigung (Schlüssel für den Raum) besitzen, in den Raum eintreten und die sich darin befindlichen Materialien benutzen.

Die beschriebenen fachlich spezifischen Konzepte stellen lediglich eine exemplarische Auswahl dar. Daneben gibt es eine Reihe weiterer fachlich spezifischer

[65] vgl. hierzu Kapitel 5

[66] Fachliche Interpretation der Beweglichkeit eines Materials: Ein Material kann abhängig von seinem fachlichen Anwendungskontext prinzipiell beliebig zwischen kooperierenden Arbeitsumgebungen ausgetauscht werden.

[67] Henning Wolf und Stefan Roock, Fachbereich Informatik an der Universität Hamburg, Arbeitsbereich Softwaretechnik

Konzepte, die basierend auf dem Materialraum modelliert werden können. Da eine tiefergehende Betrachtung den Rahmen dieser Arbeit sprengen würde, müssen wir auf eine konkretere Formulierung der Konzepte verzichten und verweisen auf weiterführende Arbeiten, die sich dieser Thematik annehmen und die aufgezeigten fachlich spezifischen Konzepte auf Basis des fachlich generellen Materialraums modellieren werden.

Kapitel 5

Zusammenführung der Funktionalität paradigmatisch unterschiedlicher Datenbanksysteme und objektorientierter Programmiersprachen

Nachdem sich diese Diplomarbeit in den letzten beiden Kapiteln ausführlich mit fachlichen Konzepten auseinandergesetzt hat, gilt es deren technische Realisierbarkeit näher zu betrachten. Bevor jedoch der Bogen zum Konzept des *Persistent Distributed Shared Memory*, dem technischen Pendant des Materialraums, gespannt werden kann, setzt sich dieses Kapitel zunächst mit der Zusammenführung der Funktionalität von Datenbanksystemen und objektorientierten Programmiersprachen, insbesondere mit der Programmiersprache C++, auseinander.

Nach der Definition des *impedance mismatch*, dessen Dimensionen wir bereits in Kapitel 2 aufgeworfen haben, werden Alternativen zu dessen Überbrückung diskutiert und die Abbildung zwischen Relationen und Objekten beleuchtet. Dazu bedarf es der Gegenüberstellung des Relationenmodells und der Objektmodelle der ODMG (C++-binding) sowie des X3J16-Komitees (C++) und einer Vorstellung eines allgemeinen Überblicks über die Merkmale von Objektmodellen.

Darauf aufbauend werden die Anforderungen an persistente Objektsysteme genannt und die *persistente Programmiersprache* als ein solches System mit ihren Konzepten vorgestellt. Obwohl im Zentrum dieses Kapitels die Zusammenführung der Funktionalität paradigmatisch unterschiedlicher Datenbanksysteme und ob-

jektorientierter Programmiersprachen steht, unterbreiten wir im letzten Abschnitt durch das Konzept des *Distributed Shared Memory* einen Vorschlag zur technisch transparenten Bereitstellung von beliebiger Beweglichkeit durch persistente objektorientierte Programmiersprachen. Dieses Konzept stellt sowohl die natürliche Erweiterung des Konzepts des *virtuellen Speichers* persistenter Programmiersprachen als auch die technische Umsetzung des Materialraums dar.

5.1 Der impedance mismatch

In Abschnitt 2.3 wurde bereits angedeutet, daß

- eine Kluft zwischen Programmiersprachen, Datenbanksystemen und Betriebssystemen besteht.
- es eine Kluft zwischen persistenten und mobilität-unterstützenden Programmiersystemen gibt.
- Programmiersprachenmodelle und Datenbankmodelle unterschiedlichen Paradigmen und Sichten entspringen.

Betrachten wir uns ferner die Modellierung von Informationssystemen, die in Abschnitt 2.2 beschrieben worden sind, mit Hilfe relationaler Datenbanksysteme, so ist ersichtlich, daß ein weiterer Strukturbruch bei der Abbildung zwischen der Anwendungsdomäne und dem Relationenmodell besteht. Objektorientierte Modelle vereinfachen diese Abbildung konzeptionell durch eine gesteigerte Modellierungsmächtigkeit. Jedoch steht die Standardisierung eines einheitlichen Modells von objektorientierten Programmiersprachen (OOPL[1]) und objektorientierten Datenbanksystemen aus.

> **Begriff 38 :** *impedance mismatch ([Heuer, Saake 95])*
> Der impedance mismatch ist ein Strukturbruch, der zwischen den Datenstrukturen unterschiedlicher Datenmodelle besteht.

Die Qualität und der Entwicklungsaufwand von Informationssystemen ist im wesentlichen dadurch bestimmt, ,,inwiefern eine direkte, unkomplizierte Abbildung der Anwendungsstrukturen in korrespondierende Programmiersprachenkonstrukte gelingt" ([Mathiske 96b]). Damit der Anwendungsentwickler von der Last der Abbildungen zwischen Programmiersprache, Datenbanksystem, Betriebssystem und der Anwendungswelt befreit werden kann, gilt es, ihn zukünftig bei der Entwicklung von Informationssystemen von der aufwendigen Aufgabe der Kombination von Programmiersprache und den Diensten der Betriebs- und Datenbanksysteme zu befreien (vgl. Abbildung 5.1, erster Kasten). Hierzu haben wir bereits in Kapitel 4 das Konzept des Materialraums vorgestellt, bei dem sich ein

[1] Abkürzung, **o**bject-**o**riented **p**rogramming **l**anguage

Anwendungsentwickler weder um die persistente Ablage von Materialien noch um die technische Verteilung von Materialien zu kümmern braucht. Zur technischen Umsetzung des Materialraums beschreibt die vorliegende Diplomarbeit im folgenden die Bereitstellung von Datenbankfunktionalität auf der Ebene einer Programmiersprache[2].

Ein erster Schritt ist die Zusammenführung der Funktionalität von Datenbanksystemen und Programmiersprachen (vgl. Abbildung 5.1, zweiter Kasten) zu einer *persistenten Programmiersprache* (*PPL*[3]), auf die Abschnitt 5.5 näher eingeht. Dabei steht die Abstraktion vom persistenten Speicher und die technisch transparente Anbindung von (unterschiedlichen) Datenbanksystemen im Mittelpunkt. Ein weiterer Schritt ist die Integration von Diensten des Betriebssystems in eine PPL (vgl. Abbildung 5.1, dritter Kasten), bei der eine technisch transparente Mobilität, d.h. der implizite Austausch von Daten zwischen Rechnerknoten, bereitgestellt wird und die Abbildung der Datenstrukturen auf Byte-Ströme für den Anwendungsentwickler entfällt. In Abschnitt 5.6 wird in diesem Zusammenhang das Konzept des *Distributed Shared Memory* diskutiert. Entspringt die mobilität-unterstützende persistente Programmiersprache dem objektorientierten Paradigma, so vereinfacht sich zusätzlich die Abbildung der Strukturen der Anwendungsdomäne auf die Programmstrukturen ([Kilberth et al. 93]).

Eine objektorientierte mobilität-unterstützende persistente Programmiersprache bietet demnach folgende Vorteile:

- Reduzierung des Programmieraufwands[4]
- Das Erlernen unterschiedlicher Sprachen[5] entfällt[6]
- Entwurf eines einzigen Schemas nach einem einheitlichen Objektmodell[7]
- Einfaches Programmiermodell durch Abstraktion von der Persistenz und der Mobilität

[2] „The advantage is a seamless world for the application programmer, where there is one coherent model of computation, not just calculation, covering everything necessary for application programming, in a single consistent framework. The conceptual advantage to the application's programmer is obvious, and there are many more application programmers than system programmers." ([Atkinson, Morrison 89])

[3] Abkürzung, **p**ersistent **p**rogramming **l**anguage

[4] „Erfahrungen in konkreten Anwendungen haben gezeigt, daß ca. 30 bis 70% des Codes für die Programmierung der Einbettung [der Datenbanken] erforderlich sind." ([Hohenstein et al. 96])

[5] Programmier- und Datenbanksprache; Interface Definition Language von Programmiersprachen, Datenbanksprachen und Diensten zur Unterstützung von Verteilung

[6] [Matthes 95]

[7] Damit entfällt der Entwurf mehrerer Schemata und die Konsistenzzusicherung bei der Abbildung zwischen diesen.

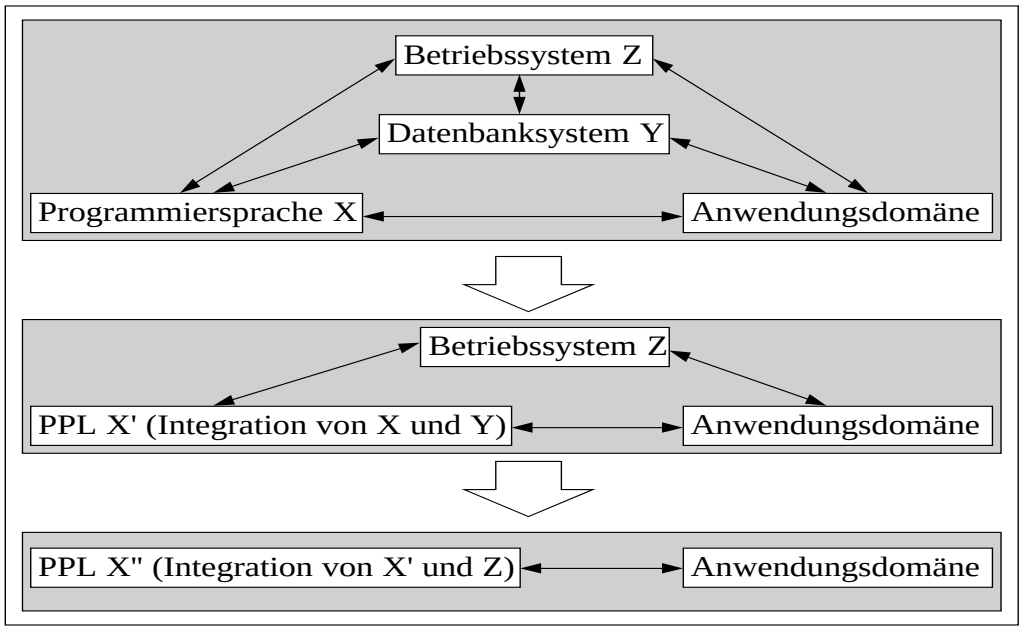


Abbildung 5.1: Zusammenführung von Betriebssystem, Datenbanksystem und Programmiersprache

- Entwicklung eines kompakteren und knapperen Codes[8]

Abschließend sei zur weiteren Motivation, warum sich diese Arbeit mit der Zusammenführung von Programmiersprache, Datenbanksystem und Betriebssystem beschäftigt, auf die Feststellungen des X3J16-Komitees[9] und der ODMG hingewiesen:

> ,,FUTURE TRENDS: In the C++ arena, the interest in application and possibly extension of C++ in several technical areas is stabilizing. These areas include concurrent and distributed programming, persistency, object management, and management of class libraries. The interest in object-oriented technology, which C++ is a part of, continues to grow. This may create in the future new dependencies with the national and international technical committees involved with object-oriented technology."([X3J16 97])

> ,,In the future, ODMG hopes to influence the C++ standard to include hooks that would make a completely seamless interface possibly without a preprocessor, or at least to standardize the preprocessor such that existing compiler and programming tools would integrate well with object-oriented DBMSs."([Cattell 94])

[8] [Wai 89]
[9] Komitee zur Standardisierung der Programmiersprache C++

Bevor auf die Anforderungen an eine PPL und die Konzepte einer PPL bzw. in der Fortsetzung einer mobilität-unterstützenden PPL eingegangen werden kann, soll der nächste Abschnitt zunächst das Relationenmodell und die objektorientierten Modelle von ODMG-konformen[10] Datenbanken und der Programmiersprache C++ vorstellen. Dabei wird klar, daß weder das Objektmodell von C++ noch das der ODMG die volle Bandbreite der Objektorientierung unterstützen. Ferner schafft der nächste Abschnitt die Grundlage zur Abbildung zwischen Relationen und Objekten.

5.2 Datenmodelle

Nach [Heuer, Saake 95] und [Wetzel 94] dient ein Datenmodell zur Erfassung und Darstellung der Informationsstruktur einer Anwendung, jedoch nicht der Information selber. Die Informationsstruktur wird durch folgende Konzepte ausgedrückt[11]:

1. *Strukturteil*
   Konzepte zur Beschreibung der Daten-/Objekttypen der Anwendungswelt und deren Beziehungen.

2. *Operationenteil*
   Menge von implizit im Modell vorhandenen Operationen (Anfrage- oder Änderungsoperationen) auf den Ausprägungen der Daten-/Objekttypen.

3. *Höhere Konzepte*
   Konzepte, die über die im Struktur- und Operationenteil gefaßten Konzepte hinausgehen, wie zum Beispiel Integritätsbedingungen und Transaktionen.

Der Strukturteil beinhaltet eine Menge von vordefinierten (built-in) Datentypen und stellt eine Reihe von Konstruktoren, mit deren Hilfe neue Datentypen definiert werden können, sowie eine Menge von Regeln bereit, die bei der Bildung neuer Typen eingehalten werden müssen. Ein Datenmodell definiert somit die Strukturen, in denen die Daten einer Anwendung ausgedrückt werden, und legt fest, wie sie dem Anwender zur Manipulation zur Verfügung gestellt werden.

Damit die nachfolgenden Unterabschnitte eine vergleichbare Darstellung des Relationenmodells und der Objektmodelle der ODMG (C++-binding des ODMG-93-Standards) und der Programmiersprache C++ gewährleisten, werden die Modelle anhand ihres Strukturteils, ihres Operationenteils und ihrer höheren Kon-

[10] eingeschränkt auf die C++-binding

[11] Da es keinen Konsenz über die Mindestanforderungen eines Datenmodells gibt, bilden die im folgenden genannten Konzepte lediglich einen Ausschnitt und dienen ausschließlich als Orientierung.

zepte beschrieben. Damit die Integritätsbedingungen und die Transaktionen vergleichend dargestellt werden können, soll im folgenden kurz auf ihre Ausprägungen eingegangen werden.

Integritätsbedingungen

Integritätsbedingungen sind Bedingungen für die Zulässigkeit und Korrektheit von

- (bestimmten) Datenbankzuständen und
- Zustandsübergängen

und sichern somit die langfristige Datenbankentwicklung. Integritätsbedingungen können prinzipiell durch Dienste einer beliebigen Schicht innerhalb einer Architektur[12] zugesichert werden ([Heuer, Saake 95]). Jedoch haben die von einem Datenbanksystem verwalteten Integritätsbedingungen den Vorteil, daß die Konsistenz des Datenbestands zentral für alle Anwendungen definiert und kontrolliert werden kann[13] ([Hohenstein et al. 96]).

Integritätsbedingungen lassen sich wie folgt charakterisieren:

1. *Integritätsarten*:
   - Wertebereich bezogen (domänen-abhängig)
   - funktional (funktional-abhängig)
   - referentiell (Referentielle Integrität)
     - Kardinalität
     - Reflexivität
2. *Zeitliche Gültigkeit*:
   - statisch (dauerhaft gültig)
   - dynamisch (zeitweise gültig)
     - transitional (abhängig von einem aktuellen Zustand)
       Spezialfälle:
       * initial (auf den Anfangszustand bezogen)
       * final (auf einen Endzustand bezogen)
     - temporal (abhängig von der Vorgeschichte, Zustandsfolge)

[12] vgl. Unterabschnitt 2.3.5

[13] Die Integrität ist direkt im Datenmodell verankert oder wird vom DBS durch *Trigger* oder über *stored procedures* sichergestellt.

3. *Lokalität der Einflußfaktoren und des Wirkungsbereichs*:
   - lokal (ein Exemplar eines Daten-/Objekttyps)
   - global (mehrere Exemplare von Daten-/Objekttypen)
4. *Urheber/Verantwortung*:
   - modellinhärent (automatisch durch das Modell gegeben)
   - explizit definierbar (explizit vom Datenbankentwickler definiert[14])

Transaktionen

Transaktionen erfüllen zwei wesentliche Aufgaben zur Konsistenzsicherung: Zum einen dienen sie der Synchronisation nebenläufiger Aktivitäten (*concurrency control*) und zum anderen dienen sie zur Behebung von Fehlern (*recovery*). Damit eine Datenbank immer in einem konsistenten Zustand ist, müssen alle Änderungen, die innerhalb einer Transaktion durchgeführt werden, entweder vollkommen und vollständig durchgeführt oder rückgängig gemacht werden. Dazu hat sich bei den relationalen Datenbanken das ACID-Prinzip[15] bewährt. [Heuer 92] stellt fest, daß komplexere Anwendungen „teilweise ganz andere Mechanismen [fordern], die durch ACID-Transaktionen und herkömmliche Concurrency Control nicht abgedeckt werden." Bei diesen Anwendungen (vgl. auch Abschnitt 2.1) müssen Transaktionen komplizierter aufgebaut und in Teiltransaktionen aufteilbar sein. In manchen Fällen (zum Beispiel CAD-Systeme) ist es sogar wünschenswert, Einschränkungen bei der Isoliertheit und der Konsistenz zu akzeptieren.

Die verzahnte Ausführung verschiedener Transaktionen ist im Prinzip immer dann korrekt, wenn der erhaltene Zustand auch bei einer ungeteilten Ausführung aller beteiligten Transaktionen in irgendeiner Reihenfolge herstellbar ist. In diesem Zusammenhang wird von der Serialisierbarkeit der Transaktionen gesprochen. Transaktionen haben dieselbe Wirkung (*same effect*), wenn ihre Lese-Operationen stets dieselben Werte liefern und alle genutzten Daten nach Abschluß der Transaktionen denselben Wert haben.

Transaktionen lassen sich im wesentlichen wie folgt charakterisieren:

---

[14] „Wird die Gewährleistung inhärenter Integritätsbedingungen dem Anwendungsprogrammierer aufgebürdet, so ruft dies die stupide Implementierung wiederkehrender Programmuster hervor. Ferner ist die Korrektheit der datenintensiven Anwendungen so nicht zu garantieren." ([Wetzel 94])

[15] **A**tomarität: Die Ausführung einer Transaktion ist unteilbar.
Konsistenz (**C**onsistency): Übergang von einem konsistenten Zustand in einen anderen konsistenten Zustand.
Isoliertheit: Transaktionen sehen nicht inkonsistente Zwischenzustände anderer Transaktionen.
Dauerhaftigkeit: Permanente Sicherung der Zustandsänderung.

1. *Strukturiertheit*:

   - flach, einfach, unstrukturiert
     Eine flache Transaktion ist eine Transaktion, die mit keinen anderen (Teil-)Transaktionen in Beziehung steht.
   - geschachtelt
     Eine geschachtelte Transaktion ist eine Transaktion, die hierarchisch andere Transaktionen beinhaltet. Zu unterscheiden ist der Einfluß des Abbruchs einer Teiltransaktion auf die umgebenden Transaktionen:
     - Eine abbrechende Teiltransaktion führt zum Rollback auf den letzten konsistenten Zustand.
     - Der Abbruch einer Teiltransaktion kann ignoriert und als Fehlernachricht propagiert werden.

2. *Länge*

   - kurz
     Eine Transaktion heißt kurz, wenn sie ihren erzeugenden Prozeß nicht überlebt.
   - lang
     Eine Transaktion heißt lang, wenn sie ihren erzeugenden Prozeß überlebt.

3. *Lokalität*

   - lokal
     Jede Transaktion, deren Lese- und Schreibzugriffe auf genau einen Server (Datenbank) wirken, heißt lokal.
   - verteilt
     Jede Transaktion, deren Lese- und Schreibzugriffe auf mehr als einen Server (Datenbank) wirken, heißt verteilt.

4. *Kontinuierlichkeit*

   - diskret
     Jede Transaktion, deren Änderungen erst zum Commit-Zeitpunkt durchgeführt werden, heißt diskret[16].
   - kontinuierlich
     Jede Transaktion, die nicht diskret ist, heißt kontinuierlich[17].

[16] Es werden auch keine Änderungen im Rollback-Segment durchgeführt.

[17] Es findet sowohl eine Propagierung der Änderungen zur Datenbank als auch zum Rollback-Segment statt.

5.2.1 Das Relationenmodell

Das Relationenmodell ist von Codd 1970 eingeführt worden. Nach Meyer in [Schneider 91] basiert eine Datenbank (genauer die betreffende Ebene der Datenbank) ,,auf dem relationalen Datenmodell genau dann, wenn der Benutzer dieser Ebene die Datenbank folgendermaßen sieht:

- (R-1)
  Es gibt eine Menge **R** von benannten Relationentypen unterschiedlichen Grades über Attributen A $\in$ **A**.

- (R-2)
  Jede Relation R (eines Typs aus **R**) besitzt einen Primärschlüssel:
  R$\hat{=}$R-Name(R-KEY, R-A-1,..., R-A-n)

- (R-3)
  Die Relationen R sind zeitlich nicht konstant."

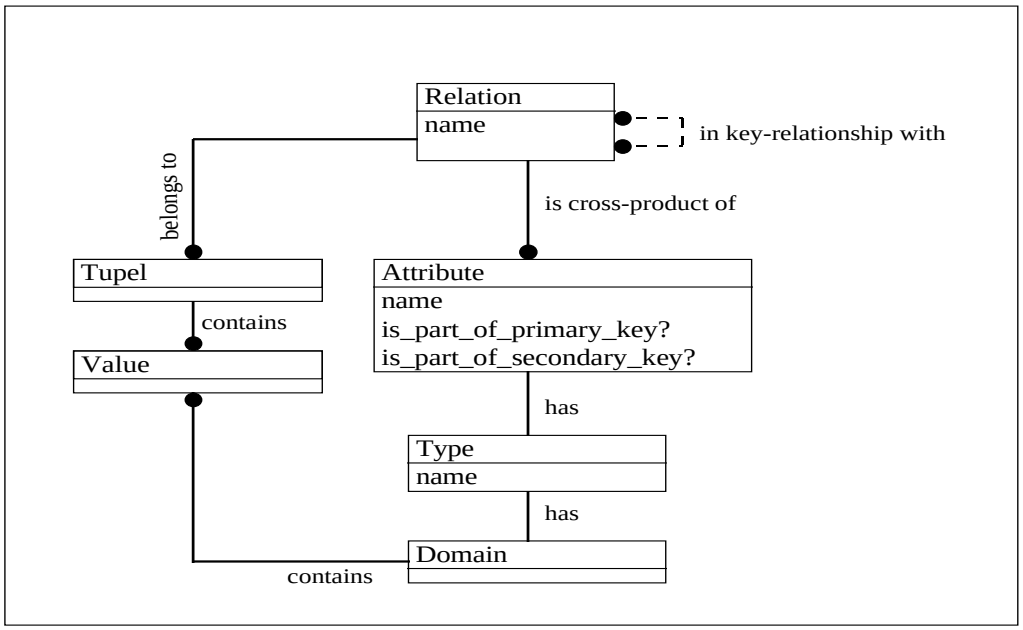


Abbildung 5.2: Grafische Visualisierung des Strukturteils des Relationenmodells

Das Relationenmodell, das ein stichhaltiges und komplettes Axiomensystem ist, basiert auf der Prädikatenlogik erster Ordnung ([Berri 90]). Es besteht aus:

1. *Strukturteil*:

   - Tabellenstruktur (Wertebereich/Domäne, Attribut, Attributwert, Relationenschema, Relation, Tupel, Datenbankschema, Datenbank)

- Identifizierende Attributmengen (Schlüssel: Primär-, Fremdschlüssel)

2. *Operationenteil*:
   Selektion, Projektion, natürlicher Verbund (*join*), Mengenoperationen, Umbenennung

3. *Höhere Konzepte*:

   - Integritätsbedingungen
   - Transaktionen

Die Abbildung 5.2 gibt einen Überblick über die Zusammenhänge des Strukturteils, und die Abbildung 5.3 stellt die vordefinierten Datentypen und die kennzeichnenden Merkmale des Relationenmodells dar.

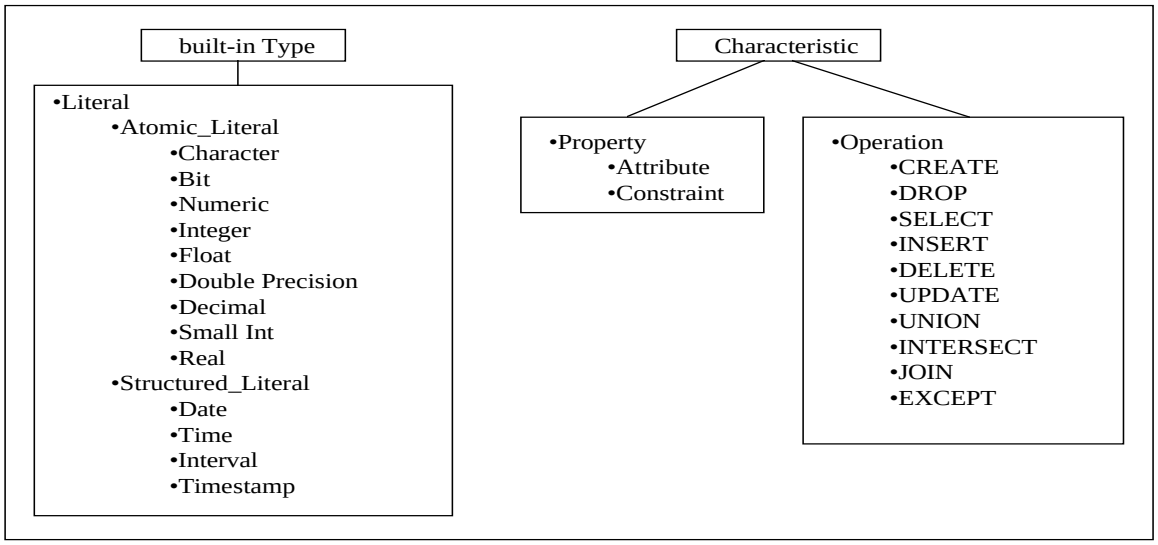


Abbildung 5.3: Übersicht über vordefinierte Datentypen und kennzeichnende Merkmale des Relationenmodells

Die Eigenschaften einer Relation sind:

1. *Es gibt keine zwei Tupel in einer Relation, die identisch zueinander sind,*
   d.h., die Zeilen in einer Tabelle sind paarweise verschieden.

2. *Die Tupel einer Relation unterliegen keiner Ordnung,*
   d.h., die Reihenfolge der Zeilen ist irrelevant.

3. *Die Attribute einer Relation unterliegen keiner Ordnung,*
   d.h., das Tauschen der Spalten verändert die Relation nicht[18].

4. *Die Attributwerte von Relationen sind atomar,*
   d.h., sie enthalten nur einen einzigen Wert.

5. *Die Spalten einer Tabelle sind homogen,*
   d.h., alle Werte in einer Spalte sind vom gleichen Datentyp.

Das Relationenmodell stellt nur einfache, kurze, lokale Transaktionen bereit und hat folgende modellinhärente Integritätsbedingungen:

- *Wertebereich bezogen, statisch, global*:
  Attributwerte sind durch Domänen festgelegt.

- *Funktional, statisch, lokal*:
  Kein Schlüsselattribut kann doppelt vergeben werden.

- *Referentiell, statisch, global*:
  Fremdschlüssel-Primärschlüssel-Beziehung.

5.2.2 Objektorientierte Datenmodelle

Das Verständnis von Objektorientierung ist sehr unterschiedlich. So ist es nicht verwunderlich, daß selbst in der gängigen Literatur der Begriff Objektorientierung abweichend gefaßt wird. Insbesondere das Verständnis von Objektorientierung in der Datenbankwelt ist von dem der Programmiersprachenentwickler zu unterscheiden. Trotz der unterschiedlichen Sichtweisen soll dieser Unterabschnitt einen Überblick über die wesentlichen Merkmale der Objektorientierung verschaffen.

Wir wollen uns dem Begriff der Objektorientierung aus der Programmiersprachensicht nähern. Bertrand Meyer (vgl. [Meyer 90]) versteht darunter die Modularisierung einer Anwendung nach Objekten, die aufgrund der Daten und nicht der Funktionen vorgenommen wird. Für ihn werden Objekte als abstrakte Datentypen (ADT) beschrieben, die auch als Klassen bezeichnet werden. Eine Klasse ist nach Meyer ein nicht triviales Modul, so daß weder fundamentale Datentypen noch Funktionen allein als Klasse bezeichnet werden. Unserer Meinung nach werden dabei die Konzepte von Typ und Klasse vermischt, denn ein Typ (nach [Booch 91], [Geppert 97], [Kilberth et al. 93], [Wetzel 94])

[18]Bestehende relationale Datenbanksysteme definieren eine Relation durch ein kartesisches Produkt. Nach [Heuer, Saake 95] ist die „Definition einer Relation als Teilmenge des kartesischen Produkts .. problematisch, da die verschiedenen Spalten einer Tabelle damit in ihrer Reihenfolge fixiert sind." Um das zu vermeiden, sollte eine Relation als Menge von Abbildungen definiert werden.

- spezifiziert ein Protokoll, d.h. benennt die Prozeduren und Funktionen (Operationen), mit denen ein Exemplar des Typs aufrufbar ist.
- enthält keine Informationen über die Speicherstruktur und den Operationskörper.
- liefert ein unveränderliches, vollständiges Reservoir „gleichartiger" Werte.

Im Gegensatz dazu ist eine Klasse ein Erzeugungsmuster, das die Eigenschaften aller Objekte (Exemplare dieser Klasse) definiert. Diese Definition umfaßt die den Objekten zugeordneten Operationen und die innere Konstruktion (Datenstrukturen und Algorithmen) der Objekte. Eine Klasse ist somit die Implementation eines abstrakten Datentyps (nach [Booch 91], [Geppert 97], [Kilberth et al. 93]). In typisierten objektorientierten Sprachen (C++, Eiffel) werden die Konzepte von Klasse und Typ jedoch vermischt, so daß bei diesen Sprachen die Klassendefinition der Typdefinition entspricht ([Kilberth et al. 93]). Deshalb hat sich der Gebrauch des Begriffs Klasse gegenüber dem des Typs durchgesetzt. Lediglich bei den von der Programmiersprache bereitgestellten built-in Typen ist der Begriff des Typs beibehalten worden, weil die Begriffe Objekt (Exemplar einer Klasse) und Wert (Wertausprägung eines built-in Typs) voneinander zu trennen sind. Die Eigenschaften von Objekten und Werten werden in Abbildung 5.4 gegenübergestellt.

Objekt	Wert
– anwendungsabhängig	– universell
– erzeug- und zerstörbar (Lebenszeit)	– ewig existierend/zeitlos
– zustandsbehaftet	– zustandslos
– Zustand veränderbar	– unveränderlich
– wird u.a. durch Werte beschrieben	– dient der Beschreibung von Objekten
– besitzt Identität	– selbstidentifizierend (keine ID)
– Operationen auf Objekten ändern den Objektzustand	– Operationen über Werten liefern neue Werte

Abbildung 5.4: Gegenüberstellung der Eigenschaften von Objekt und Wert

Ein wesentliches Merkmal der Objektorientierung ist die Vererbung. Sie ist eine Spezialisierungs-/Generalisierungsbeziehung zwischen Klassen und zwischen Typen, aus der sich eine Vererbungshierarchie ergibt. Eine Klasse bzw. ein Typ

kann von einer oder mehreren anderen Klassen bzw. Typen erben, so daß eine Vererbungsbeziehung hinsichtlich ihrer Kardinalität (einfach oder mehrfach) unterschieden werden kann. Es können folgende Dinge vererbt werden:

1. *Verhaltenssignatur*
2. *Objektstruktur*
3. *Verhaltensimplementation*

Die Klassenvererbung und die Typvererbung sind zwei voneinander zu unterscheidende Konzepte. Während bei der Typvererbung ausschließlich die Verhaltenssignatur vererbt wird, werden bei der Klassenvererbung neben der Verhaltenssignatur auch die Objektstruktur und die Verhaltensimplementation vererbt.

Die Typvererbung ist ein Mechanismus, der die Spezifikation der Schnittstelle eines Obertyps in seinen Untertypen bereitstellt. Die Spezifikation des Obertyps kann im Untertyp erweitert werden. Die Klassenvererbung hingegen ist ein Mechanismus, der Beschreibungen einer Oberklasse in ihren Unterklassen verfügbar macht. In der Unterklasse können die geerbten Beschreibungen spezialisiert werden und Methoden definiert (implementiert), redefiniert (*overriding*[19]) und hinzugefügt werden. Die Klassenvererbung impliziert die Typvererbung, weil eine Implementation von der durch den Typ festgelegten Verhaltenssignatur abhängt. Durch die Klassenvererbung entstehen Klassenhierarchien, in denen Methoden mit gleicher Schnittstelle existieren. Ein Aufruf einer Operation kann sich somit auf Objekte verschiedener Klassen beziehen (*Polymorphismus*). Erst zur Laufzeit kann entschieden werden, welche Operation zur Ausführung kommt (*late binding*[20]). Nach Wegener (in [Heuer 92]) kann eine Programmiersprache erst dann als objektorientiert bezeichnet werden, wenn Vererbung zwischen Klassen unterstützt wird.

Jedes Objekt besitzt eine eindeutige Objektidentität. Mit Hilfe des Konzepts der Identität ist es möglich, die Individualität eines Objekts sicherzustellen. Jedem Objekt wird bei seiner Instantiierung eine Objektidentität eineindeutig zugeordnet, die sich während seiner Lebenszeit nicht ändert. Die Objektidentität ist eine interne Objektqualität und bleibt somit dem Benutzer verborgen. Die Objektidentität ist sowohl vom Aufenthaltsort als auch vom Zustand des Objekts unabhängig. Es gibt keine Identität losgelöst von einem Objekt.

Objekte können auch über Objektschlüssel eindeutig innerhalb eines bestimmten

[19]Der Begriff des *overriding*, bei dem eine geerbte Methode redefiniert wird, ist von dem Begriff des *overloading* zu trennen. Eine überladene Methode behält zwar den Namen bei, kann jedoch die Art und die Anzahl der Parameter variieren.

[20]Synonyme: spätes Binden und dynamisches Binden

Bereichs identifiziert werden. Ein Objektschlüssel kann sich aus einer oder mehreren Eigenschaften und Beziehungen des Objekts zusammensetzen. Er stellt eine externe Objektqualität dar und sollte daher nicht vom System vergeben werden[21]. Obwohl sowohl die Objektidentität als auch Objektschlüssel ein Objekt eindeutig identifizieren, ist eine konzeptionelle Trennung dieser beiden Aspekte sinnvoll, weil Schlüssel eine zusätzliche Eindeutigkeitsforderung an die Nutzinformationen stellen ([Hohenstein et al. 96]).

Der Begriff des Objekts, auf den wir im Abschnitt 2.2 bereits kurz eingegangen sind, und die Beziehungen, die zwischen Typen, Klassen und Objekten bestehen, sollen an dieser Stelle nicht weiter vertieft werden. Vielmehr sei an dieser Stelle auf den Anhang A verwiesen, in dem wir einen ausführlichen Überblick über die Eigenschaften von Objekten[22] und die Eigenschaften von Beziehungen zwischen Objekten[23] sowie Beziehungen zwischen Klassen[24], Beziehungen zwischen Typen[25] und Beziehungen zwischen Objekt, Klasse und Typ[26] geben.

Allgemein besteht ein Objektmodell aus:

1. *Strukturteil*:
   - built-in Typen (integer, character, float etc.)
   - built-in Klassen (zum Beispiel Sammlungen)
   - Klassen und Typen, Objekte und Werte
   - Komplexe Objekte
   - Extents
   - Objektidentität
   - Objektschlüssel (Menge von Objektattributen)
   - Objektname(n)

[21] „In fast allen Systemen, die wertdefinierte Schlüssel unterstützen, wird die Schlüsseleigenschaft bisher nur indirekt über die Erzeugung eines eindeutigen Indexes (Unique Index) hergestellt. Die Sicherstellung der Integritätsbedingungen für Schlüssel ist dabei an das Vorhandensein des Indexes gekoppelt. In einigen Systemen können Schlüssel bzw. Indexe nur über einzelne Attribute definiert werden." ([Hohenstein et al. 96])

[22] Bekanntheit, Abhängigkeit, Sichtbarkeit, Rekursivität, Änderbarkeit, Duplizierbarkeit, Lebensdauer, Mobilität

[23] Kardinalität, Reflexivität, Partnerkardinalität, Attributiertheit

[24] Is-Subclass-Of, Is-Superclass-Of

[25] Is-Subtype-Of, Is-Supertype-Of

[26] Objekt-Klasse-Beziehung, Klasse-Objekt-Beziehung, Klasse-Typ-Beziehung, Typ-Klasse-Beziehung

- Beziehungen:
  - zwischen Klassen und Objekten
  - zwischen Klassen und Typen
  - zwischen Objekten

2. *Operationenteil*:
   - Anfrage- und Änderungsoperationen
3. *Höhere Konzepte*:
   - Verhalten: Methoden und Nachrichten
   - Kapselung (von Datenstrukturen und deren Verhalten)
   - Objektlebensdauer: transiente und persistente Ausprägungen der Lebensdauer
   - Objektbeweglichkeit: mobil, immobil und ubiquitär
   - Typvererbung und Klassenvererbung
   - Polymorphie (Überladen (*overloading*), Überschreiben (*overriding*))
   - Generizität[27]
   - Metaklassen/Meta-Objekt-Wissen (Schema-Informationen)
   - Integrität
   - Transaktionen

Die Ergebnisse der Anfrage- und Änderungsoperationen unterliegen unterschiedlichen Semantiken ([Heuer 92], [Heuer, Saake 95]). Besteht ein Ergebnis lediglich aus einem Teil des Zustands eines Objekts oder mehrerer Objekte, so spricht [Heuer, Saake 95] von *relationaler Semantik*. Dabei wird auf die Berücksichtigung der Objektidentität, des Klassenkonzepts und der Strukturhierarchie verzichtet und (Teil-)Zustände von Objekten als geschachtelte Relationen oder Mengen komplexer Werte angesehen. Bei der *objekterzeugenden Semantik* werden neue Objekte für Anfrageergebnisse mit den Zuständen erzeugt, die von vorhandenen Objekten extrahiert werden. Das Ergebnis einer Anfrage ist ein Objekt einer gegebenenfalls dynamisch erzeugten Klasse, die parallel zur bisherigen Klassenhierarchie angeordnet wird. Von *objekterhaltender Semantik* wird gesprochen, wenn ein Anfrageergebnis aus einer Auswahl der in der Datenbank vorkommenden Objekte besteht. Dabei sind die zwei Fälle zu unterscheiden, bei denen entweder neue

[27] „Eine parametrisierte Klasse (auch generische Klasse genannt) ist eine Klasse, die als Schablone für andere Klassen dient - eine Schablone, die durch andere Klassen, Objekte und/oder Operationen parametrisiert werden kann. Eine parametrisierte Klasse muß instantiiert werden (d.h. ihre Parameter müssen mit Werten versehen werden), bevor Objekte dieser Klasse erzeugt werden können." ([Booch 94])

Extensionen zu einer schon bestehenden Klasse (statische Klassifizierung) oder dynamisch erzeugte Ober- oder Unterklassen zu den schon bestehenden Klassen (dynamische Klassifizierung) gebildet werden.

Ein objektorientiertes Datenmodell stellt prinzipiell folgende modellinhärente Integritätsbedingungen bereit:

- *Wertebereich bezogen, statisch, global*:
  Attributwerte (von built-in Typen) sind durch ihre Wertebereiche festgelegt.

- *Funktional, statisch, global*:

  1. Objektidentitäten werden eineindeutig vergeben. Sie können nicht doppelt vergeben werden.
  2. Objektschlüssel, eine Menge von Attributen und/oder Beziehungen, können nur eindeutig vergeben werden.

- *Referentiell, statisch, global*:
  Die Integrität bi-direktionaler Beziehungen wird zugesichert.

In objektorientierten Systemen können auf einfache Weise Integritätsbedingungen explizit definiert werden, weil Daten zusammen mit ihren Funktionen in Objekten gekapselt werden. Darüber hinaus stellen einige objektorientierte Datenmodelle dem Entwickler von Anwendungen Konstrukte bereit, mit denen er in der Lage ist, Integritätsbedingungen explizit zu definieren. So stellt die Programmiersprache Eiffel die Konstrukte *ensure*, *require* und *invariant* bereit, mit denen Bedingungen festgelegt werden können, die beim Eintritt in eine Operation, beim Austritt aus dieser bzw. zwischen Operationsaufrufen (zum Zeitpunkt stabiler Objektzustände) gelten müssen.

5.2.3 ODMG-93 Release 1.1

ODMG-93 Release 1.1 ist ein Standard für ein objektorientiertes Datenbankmodell, der im Jahr 1993 von der Object Database Management Group (ODMG) veröffentlicht worden ist. Die Mitglieder der ODMG[28] verfolgen das Ziel der Vereinheitlichung der Datenbankmodelle und -sprachen der kommerziellen Anbieter. Dabei stehen folgende leitende Richtlinien im Vordergrund:

[28] ObjectDesign International, Objectivity, Ontos, O2 Technology, Versant Object Technology, POET

- Vereinheitlichung der Typsysteme von OOPL und OODBPL[29]
- Bereitstellung gleicher Funktionalität zur Behandlung von transienten und persistenten Objekten (transparente Persistenz)
- Einbindung der Datenbankoperationen in die Syntax der OOPL
- Freie Kombinierbarkeit von Datenbankoperationen und OOPL-Operationen

Der ODMG-93-Standard definiert ein Objektmodell, eine *object definition language* (ODL), eine *object query language* (OQL) und die Spracheinbettung[30] in die Programmiersprachen C++ und Smalltalk. 1995 veröffentlichte die ODMG das Release 1.2[31], in dem nur geringfügige Änderungen vorgenommen worden sind. Im Laufe des Jahres 1997 soll der Standard in der Version 2.0 erscheinen.

Der ODMG-93-Standard - eingeschränkt auf die C++-binding - besteht aus:

1. *Strukturteil*[32] :
   - built-in Typen (int, char, float, etc.)
   - built-in Klassen (String, Interval, Date, Time, Timestamp, Collection, Set, Bag, List, Varray)
   - Klassen und Typen, Werte und Objekte
   - Objektidentität (für Entwickler verborgen)
   - Objektname
   - Beziehungen:
     - zwischen Klassen und Objekten: Is-Instance-Of, Is-Template-Of
     - zwischen Objekten (uni-direktional, bi-direktional (*inverse*); 1:1, 1:n, m:n; binär; nicht-attributiert)
2. *Operationenteil*:
   - Anfrageoperationen: rudimentärer Anfragemechanismus (OQL)
   - Änderungsoperationen
     - built-in Operationen auf Objekttypen: Erzeugen, Löschen, als modifiziert markieren, Identitätsvergleich, Zuweisung, Kopieren, Einchecken, Auschecken

[29] **o**bject-**o**riented **d**atabase **p**rogramming **l**anguage
[30] auch als *binding* bezeichnet
[31] auch als ODMG-95 bezeichnet
[32] Im Vergleich zur C++-binding bietet das ODMG-Modell Objektschlüssel, Extents und mehrere Objektnamen.

3. *Höhere Konzepte*:

   - Verhalten: Methoden
   - Kapselung (von Datenstrukturen und deren Verhalten)
     - einschränkbar durch public, private, protected, friend, static
   - Objektlebensdauer[33]: innerhalb des Gültigkeitsbereichs von Prozeduren, Prozessen, Datenbanken
   - Polymorphie (Überladen (*overloading*), Überschreiben (*overriding*))
   - Templates (Klassenschablonen zur Bereitstellung von Generizität)
   - Integrität: modellinhärente Integritätsbedingungen
   - Klassenvererbung (Mehrfachvererbung)
   - Transaktionen

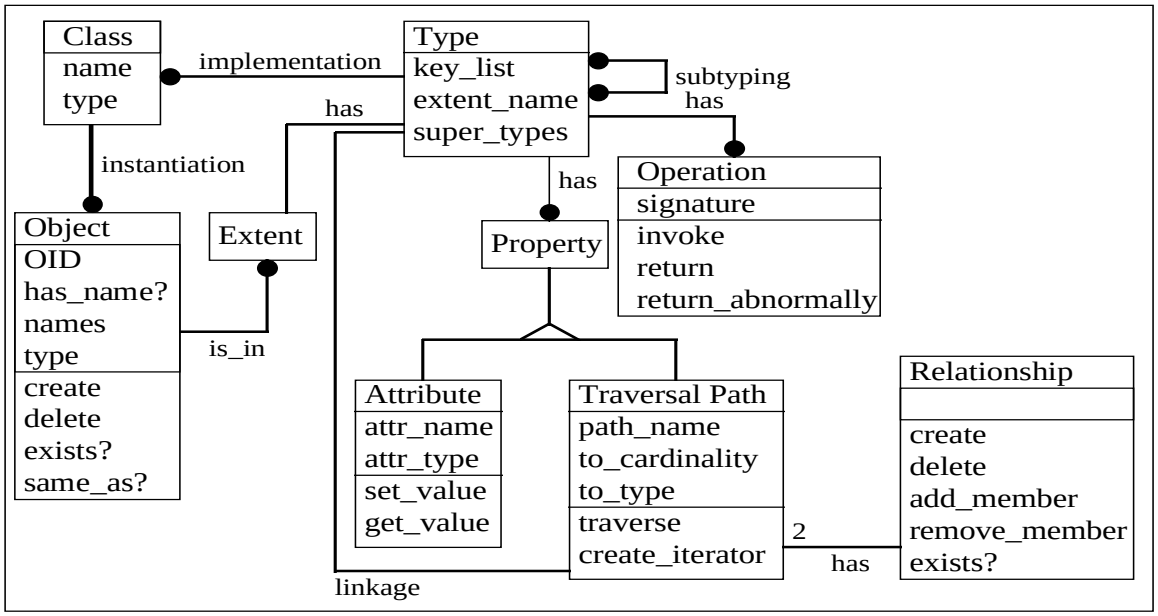


Abbildung 5.5: Grafische Visualisierung des Strukturteils des ODMG-93-Objektmodells (nach [Loomis 93b])

Die Abbildung 5.5[34] gibt einen Überblick über die Zusammenhänge des Strukturteils des im Vergleich zur C++-binding semantisch reicheren ODMG-93-Objektmodells, und die Abbildung 5.6 stellt die vordefinierten Datentypen und die kennzeichnenden Merkmale des ODMG-93-Objektmodells dar.

[33] Die Lebensdauer wird bei Erzeugung eines Objekts festgelegt und ist unveränderlich.

[34] Auf die Darstellung der Klassen *Collection* und *Transaction* wurde bewußt verzichtet.

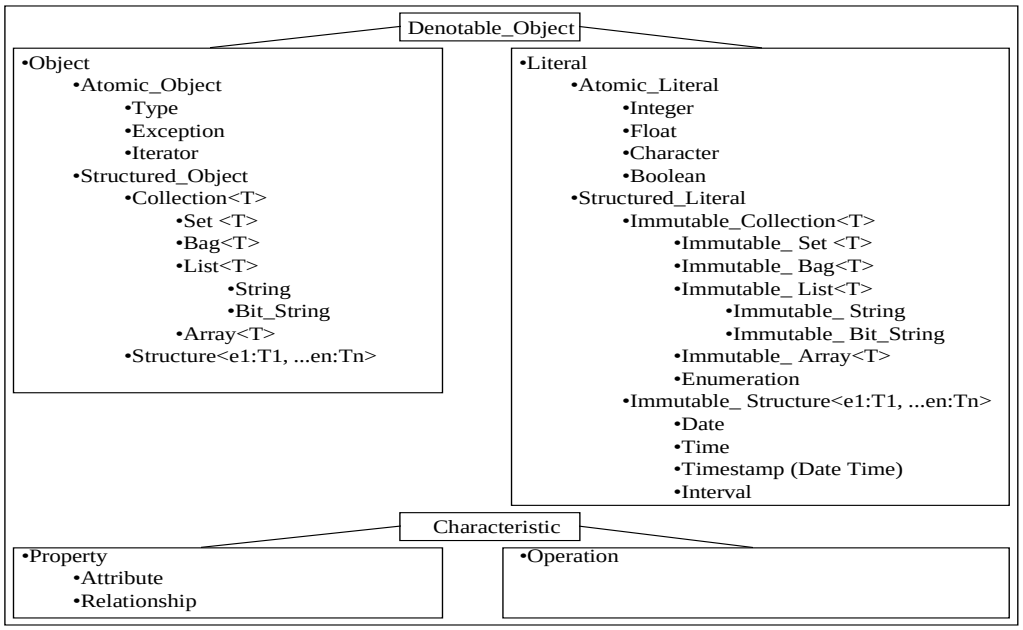


Abbildung 5.6: Übersicht über vordefinierte Datentypen und kennzeichnende Merkmale des ODMG-93-Objektmodells

Das Datenmodell der ODMG stellt einfache, kurze, lokale und geschachtelte, kurze, lokale Transaktionen bereit und bietet die folgenden modellinhärenten Integritätsbedingungen:

- *Wertebereich bezogen, statisch, global*:
  Attributwerte (Literale) sind durch Wertebereiche festgelegt.
- *Funktional, statisch, global*:
  1. Objektidentitäten werden eineindeutig vergeben. Sie können nicht doppelt vergeben werden.
  2. Objektnamen werden durch den Benutzer vergeben. Das System stellt die Eindeutigkeit sicher.
- *Referentiell, statisch, global*:
  Die Integrität bi-direktionaler Beziehungen wird zugesichert.

In zukünftigen Versionen der C++-binding sind u.a. folgende Erweiterungen geplant:

- Benennung und Verwaltung von Anfrageergebnissen
- Transaktionen, die einfach oder geschachtelt, kurz oder lang und lokal oder verteilt sind.

- Größere Auswahl impliziter Integritätsbedingungen
- Default-Werte für Attribute
- Definition von transitiven und reflexiven Beziehungen
- Definition von Attributen eines Objekttyps als Schlüssel

5.2.4 ANSI C++ (X3J16)

Die Programmiersprache C++, die nach ANSI X3J16 standardisiert ist, befindet sich seit 1985 auf dem Markt. Sie wurde von Bjarne Stroustrup, AT&T, entwickelt. C++ ist eine Hybridsprache, die aus der prozedural-ablauforientierten Sprache C besteht und um Konzepte der Objektorientierung erweitert wurde. Im Laufe des Jahres 1997 soll eine neue und weltweit einheitliche Sprachspezifikation des ANSI/ISO-Komitees veröffentlicht werden, die bereits in einer Vorabversion [X3J16 97] vorliegt.

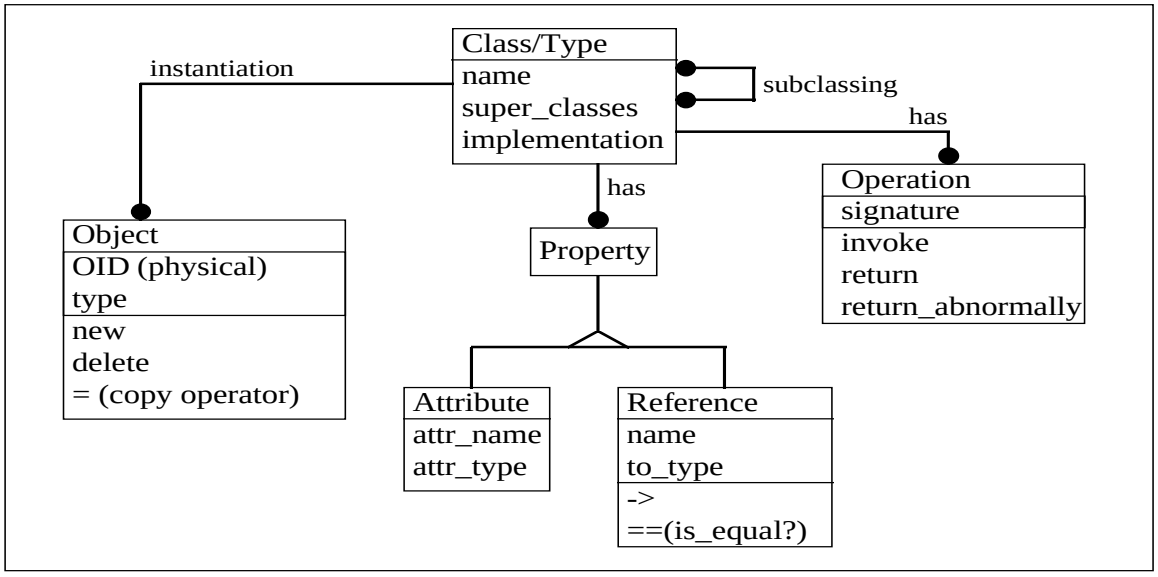


Abbildung 5.7: Grafische Visualisierung des Strukturteils des C++-Objektmodells

Das Objektmodell besteht aus[35]:

1. *Strukturteil*:
   - built-in Typen (int, char, float etc.)

[35] Wir beschränken uns auf die objektorientierten Konzepte von C++.

- Klassen (Synonym: Typen), Objekte und Werte
- Beziehungen:
  - zwischen Klassen und Objekten: Is-Instance-Of, Is-Template-Of
  - zwischen Objekten (uni-direktional, 1:1, binär, nicht attributiert)

2. *Operationenteil*:

   - Änderungsoperationen
     - built-in Operationen auf built-in-Typen: Vergleich, Zuweisung
     - built-in Operationen auf Objekten: Kopieren des Objektzustands

3. *Höhere Konzepte*:

   - Verhalten: Methoden
   - Kapselung (von Datenstrukturen und deren Verhalten)
     - einschränkbar durch public, private, protected, friend, static
   - Objektlebensdauer: innerhalb des Gültigkeitsbereichs von Prozeduren, Prozessen
   - Polymorphie (Überladen (*overloading*), Überschreiben (*overriding*))
   - Templates (Klassenschablonen zur Bereitstellung von Generizität)
   - Metaklassen (Typinformationen durch RTTI[36])
   - Klassenvererbung (Mehrfachvererbung)

Die Abbildung 5.7 gibt einen Überblick über die Zusammenhänge des Strukturteils, und die Abbildung 5.8 stellt die vordefinierten Datentypen und die kennzeichnenden Merkmale des C++-Objektmodells dar. Die Programmiersprache C++ stellt weder Transaktionen noch modellinhärente Integritätsbedingungen bereit.

5.2.5 Vergleich

In den letzten Unterabschnitten sind wir ausführlich auf das Relationenmodell sowie auf die Objektmodelle der ODMG und der Programmiersprache C++ eingegangen. Dabei ist auffällig, daß

- ein Strukturbruch zwischen dem Relationenmodell und den Objektmodellen besteht und
- eine Kluft zwischen den dargestellten Objektmodellen existiert.

[36] **R**un-**T**ime **T**ype **I**nformation

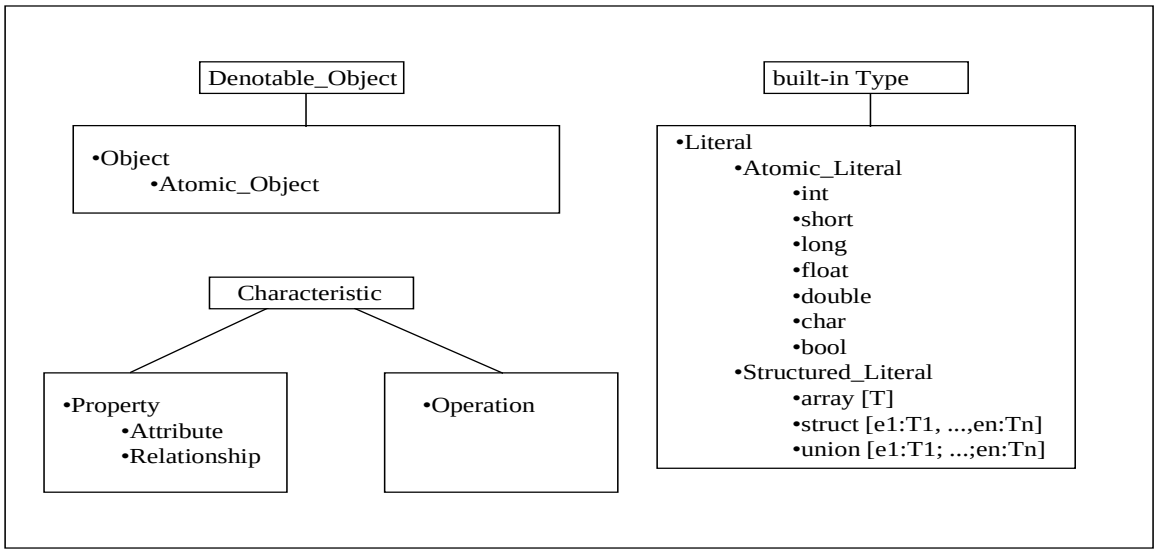


Abbildung 5.8: Übersicht über vordefinierte Datentypen und kennzeichnende Merkmale des C++-Objektmodells

Da für objektorientierte Programmiersprachen und objektorientierte Datenbanksysteme kein einheitliches Objektmodell existiert, weisen die Objektmodelle der ODMG und des X3J16-Komitees Differenzen auf. Diese führen jedoch zu keinem Strukturbruch zwischen den Modellen, sondern lediglich zu geringfügigen Abweichungen bzw. der Vernachlässigung von Teilkonzepten. Die Tabelle 5.1 stellt das Objektmodell der C++-binding des ODMG-93-Standards dem C++-Objektmodell des X3J16-Komitees gegenüber. Interessanter als der Vergleich der objektorientierten Modelle ist eine Betrachtung des Strukturbruchs zwischen Relationenmodell und Objektmodellen. Das Relationenmodell ist ein attribut- und mengenbasierter Ansatz, bei dem die anwendungsrelevanten Gegenstände in Form von Attributen, Tupeln und Tabellen dargestellt werden. Diese relationalen Strukturen bauen hierarchisch aufeinander auf, so daß eine Tabelle aus Tupeln und ein Tupel aus Attributen besteht. Tabellen können demnach keine Tabellen und Tupel keine Tupel beinhalten. Bei der Deklaration von Attributen muß auf vorgegebene Attributtypen zurückgegriffen werden, weil keine anwendungsspezifischen Attributtypen definiert werden können. Ein Zugriff auf die Daten einer Datenbank erfolgt ausschließlich global, so daß unmittelbar auf alle Daten in gleicher Weise zugegriffen werden kann. Lediglich die Tupel einer Tabelle sind eindeutig über Primärschlüssel identifizierbar. Da das Relationenmodell ein rein strukturorientierter Ansatz ist, werden keine Mechanismen zur Realisierung von Verhalten (Funktionen, Prozeduren) zur Verfügung gestellt.

Demgegenüber stellen die Objektmodelle, denen es im Vergleich zum Relation-

	ODMG-93 C++-binding	*C++*
built-in Typen	int, char, float etc.	int, char, float etc.
built-in Klassen	String, Interval, Date, Time, Timestamp Sammlungen: Collection, Set, Bag, List, Varray	keine
Klassen, Typen, Objekte, Werte	unterstützt	unterstützt, aber Typ als Synonym für Klasse
komplexe Objekte	keine Unterstützung	keine Unterstützung
Extents	keine Unterstützung	keine Unterstützung
Objektidentität	bleibt verborgen	keine Unterstützung
Objektschlüssel	keine Unterstützung	keine Unterstützung
Objektnamen	max. ein Objektname	keine Unterstützung
Beziehungen zwischen Klassen und Objekten	Is-Instance-Of, Is-Template-Of	Is-Instance-Of, Is-Template-Of
Beziehungen zwischen Objekten	uni-direktional, bi-direktional (inverse); 1:1, 1:n, n:m; binär; nicht-attributiert	uni-direktional; 1:1; binär; nicht-attributiert
Anfrageoperationen	rudimentäre Anfragefunktion	keine
Änderungsoperationen	Erzeugen, Löschen, als modifiziert markieren, Identitätsvergleich, Zuweisung, Kopieren, Ein-/Auschecken	built-in Typen: Vergleich, Zuweisung Objekte: Kopieren des Objektzustands
Verhalten	Methoden	Methoden
Kapselung	einschränkbar durch public, private, protected, friend, static	einschränkbar durch public, private, protected, friend, static
Objektlebensdauer	innerhalb des Gültigkeitsbereichs von Prozeduren, Prozessen, Datenbanken	innerhalb des Gültigkeitsbereichs von Prozeduren, Prozessen
Objektbeweglichkeit	keine Einordnung möglich	keine Einordnung möglich
Typvererbung, Klassenvererbung	Klassenvererbung (Mehrfachvererbung)	Klassenvererbung (Mehrfachvererbung)
Polymorphie	ja	ja
Generizität	Templates	Templates
Meta-Objekt-Wissen	für Entwickler verborgen	Typinformationen (RTTI)
Integrität	modellinhärente Integritätsbedingungen	keine
Transaktionen	einfach, kurz und lokal; geschachtelt, kurz, lokal	keine

Tabelle 5.1: Gegenüberstellung der Objektmodelle

enmodell an formalen Grundlagen mangelt ([Rahayu, Chang 93]), einen objektorientierten Ansatz dar. Die anwendungsrelevanten Gegenstände werden durch Objekte modelliert. Ein Objekt besteht aus einer Struktur, die aus weiteren Objekten und/oder Attributen bestehen kann, und einem objekt-spezifischen Verhalten. Objekte mit gleicher Struktur und gleichem Verhalten werden in Klassen zusammengefaßt. Die objektorientierten Strukturelemente (Klasse, Objekt, Attribut) bauen demnach nicht hierarchisch aufeinander auf. Da neben den Attributtypen (auch als built-in Typen bezeichnet), die vom System bereitgestellt werden, beliebige anwendungsspezifische Klassen definiert werden und Objekte beliebig und auch rekursiv aus Objekten bestehen können, stellen die Objektmodelle mächtigere Strukturierungsmöglichkeiten bereit. Durch die Zusammenführung von Struktur und Verhalten sind die Daten durch Funktionen bzw. Prozeduren gekapselt, so daß nicht wie beim Relationenmodell global auf Daten zugegriffen werden kann. Ferner besitzt jedes Objekt eine vom System vergebene, systemweit eindeutige, unveränderbare Identität. Die grundsätzliche Vergabe von Objektidentitäten vereinfacht das Objektmodell und dessen Handhabung.

Die Merkmale des Relationenmodells können ähnlichen Merkmalen der Objektmodelle zugeordnet werden. Abbildung 5.9 stellt ausgewählte Merkmale der Objektmodelle denen des Relationenmodells gegenüber. Trotz dieser möglichen Zuordnung besteht zwischen den Merkmalen ein Strukturkonflikt, der sich u.a. an folgenden Punkten festmachen läßt:

1. Ein Tupel kommt aufgrund seiner Struktur einem Objekt am nähesten. Jedoch kapselt ein Tupel weder Verhalten noch besitzt ein Tupel eine systemweit eindeutige Identität. Ferner kann sich ein Tupel nicht aus anderen Tupeln zusammensetzen.

2. Während in einer Relation keine zwei Tupel existieren dürfen, die in allen Attributen übereinstimmen, können zustandsgleiche Objekte, die immer eine unterschiedliche Identität besitzen, sich in einem Klassen-Extent befinden.

3. Die Objektmodelle bieten unterschiedliche Beziehungen (zum Beispiel Vererbungsbeziehungen, Komponentenbeziehungen[37]) an, mit denen Klassen verbunden werden können. Unterschiedliche semantik-tragende Beziehungen können im Relationenmodell nicht dargestellt werden. Eine Beziehung kann ausschließlich über Fremdschlüssel-Beziehungen dargestellt werden.

4. Dem Konzept der Objektidentität steht das Konzept des Primärschlüssels gegenüber. Während ein Primärschlüssel im Laufe der Zeit veränderbar ist

[37]Es sei angemerkt, daß die Semantik von Beziehungen zuweilen unklar bleibt. Bei einer Komponentenbeziehung stellt sich zum Beispiel die Frage, ob das Komponentenobjekt ohne sein Kontextobjekt eine Lebensberechtigung hat ([Heuer 92]).

und vom Datenbankanwender vergeben werden kann, ist eine Objektidentität unveränderbar und ausschließlich vom System vergeben. Hinzu kommt, daß ein Primärschlüssel nur innerhalb einer Tabelle und eine Objektidentität systemweit eindeutig ist.

Objektmodell	Relationenmodell
• Klasse • Objekt • built-in Typ (Basisklasse) • systemweite Identität • Objekt enthält Objekte • Funktionen kapseln Daten • Objektreferenz • identität-basiert	• Tabelle • Tupel • Attributtyp • Primärschlüssel • Tupel enthalten Attribute • globaler Zugriff • Fremdschlüssel • wert-basiert

Abbildung 5.9: Gegenüberstellung von Merkmalen des Relationenmodells und der Objektmodelle

Weiteres Konfliktpotential zwischen den Modellen besteht beim Operationenteil und bei den höheren Konzepten:

1. Das Relationenmodell stellt nur eine begrenzte Anzahl an vordefinierten (generischen) Operatoren zur Manipulation von atomaren Attributen und Mengen zur Verfügung. Bei den Objektmodellen können beliebige Operationen, die die Schnittstellen der Objekte bilden und das Verhalten repräsentieren, definiert werden.

2. Während die Objektmodelle prinzipiell Transaktionen beliebiger Strukturiertheit, Länge und Lokalität bereitstellen, bietet das Relationenmodell lediglich einfache, kurze und lokale Transaktionen.

3. Integritätsbedingungen werden, sofern sie nicht bereits in den Modellen verankert sind, auf unterschiedliche Weise definiert. Während im Objektmodell die Integritätsbedingungen implizit im Verhalten der Klassen bzw. Objekte festgelegt werden, sind sie im Relationenmodell explizit über Bedingungen oder Integritätsregeln zu definieren.

5.3 Überbrückung des impedance mismatch zwischen Datenbanksystemen

Zur Überbrückung des *impedance mismatch* zwischen Datenbanksystemen gibt es grundsätzlich unterschiedliche Möglichkeiten. Es wäre zum Beispiel denkbar, auf die mächtigen Strukturen der OODBS zu verzichten und Objekttypen, die die Datenstrukturen der RDBS nachbilden, zu spezifizieren. Eine andere Möglichkeit besteht darin, die Konzepte und die Datenstrukturen sowohl der RDBS als auch der OODBS in einem neuen System zu vereinen. Bei diesem Vorgehen ist jedoch zu bedenken, daß sowohl die Schnittstelle der RDBS als auch die der OODBS erweitert werden müssen, damit sowohl die OODBS die Strukturen und die Funktionalität von RDBS anbieten als auch umgekehrt RDBS die Strukturen der OODBS verarbeiten und Dienste der OODBS bereitstellen können ([Ananthanarayanan et al. 93]).

5.3.1 Der ODMG-93-Standard als abstrakte Schnittstelle

Bei der Überbrückung des *impedance mismatch* sind der anwendungsspezifische Kontext, das Paradigma der Analyse, des Designs, der Spezifikation und der Programmierung sowie die zur Verfügung stehenden Softwareressourcen wie Programmiersprachen und Datenbanksysteme zu betrachten. Wird ein Informationssystem nach der strukturierten Analyse entworfen und mit einer imperativen Programmiersprache umgesetzt, scheint es sinnvoll, Datenbanksysteme mit einer Schnittstelle, die ähnlich der der relationalen Datenbanksysteme ist, anzubinden.

Unser Fokus bei der Entwicklung von Informationssystemen richtet sich auf die Objektorientierung, die aus heutiger Sicht ein natürliches Strukturierungskonzept zur Erfassung der Anwendungsstrukturen bereitstellt. Neben dem objektorientierten Entwurf dieser Systeme setzen wir objektorientierte Programmiersprachen, im Rahmen dieser Arbeit C++, ein. Da bei der Objektorientierung keine relationalen Konzepte eingesetzt werden und darüber hinaus eine direkte Abbildung zwischen den Gegenständen der Anwendungsdomäne, den Objekten der Programmiersprache und denen der Datenbanksysteme möglich sein sollte, schlagen wir als zugrundeliegendes Objektmodell und als Schnittstelle einer Abstraktionsschicht, die Datenbanksysteme beliebigen Paradigmas kapselt, die C++-binding des ODMG-93-Standards mit der bereitgestellten Funktionalität vor[38]. Diese scheint für uns erfolgversprechend zu sein, weil

- der ODMG-93-Standard der einzige objektorientierte DBS-Standard ist.
- OODBS bereits eine ODMG-93-konforme Schnittstelle bieten.

[38] Diese Festlegung bedeutet nicht, daß zukünftige ODMG-Versionen nicht berücksichtigt werden sollen.

- die Akzeptanz der Objektorientierung wächst.
- der Marktanteil von OODBS wächst[39].
- relationale Konzepte der heutigen Sichtweise nicht mehr gerecht werden.
- eine Abbildung zwischen relationalen und objektorientierten Strukturen möglich ist.

Die Entscheidung für die C++-binding des ODMG-93-Standards hat folgende Auswirkungen:

- ODMG-konforme Datenbanksysteme können direkt angeschlossen werden[40]. Da viele unterschiedliche Hersteller OODBS mit ODMG-93-Schnittstelle anbieten, entsteht keine Abhängigkeit von einem Datenbanksystementwickler. Auf spezielle Leistungsmerkmale, die von unterschiedlichen OODBS angeboten werden aber nicht von der ODMG standardisiert sind, muß verzichtet werden.
- Auf die relationalen Datenbanksysteme muß eine Abstraktionsschicht aufgesetzt werden, die nach außen eine ODMG-konforme Schnittstelle bereitstellt, zwischen dem Objektmodell und dem Relationenmodell abbildet und sich dabei des Leistungsspektrums der relationalen Datenbanksysteme bedient. Wenn die Abstraktionsschicht ausschließlich die SQL-Schnittstelle gebraucht, dann können beliebige relationale Datenbanksysteme unterschiedlicher Hersteller angebunden werden[41].
- Bestehende relationale Datenbanksysteme können durch den Einsatz einer Abstraktionsschicht nutzbar gemacht werden[42], ohne daß auf den Einsatz der objektorientierten Technologie verzichtet werden muß. Ein Austausch der Datenbanksysteme insbesondere ein Umstieg auf objektorientierte ist leicht und jederzeit möglich.

Obwohl wir die C++-binding der ODMG als gemeinsame Schnittstelle von Datenbanksystemen ausschließlich zur Überbrückung des *impedance mismatch* zwischen paradigmatisch unterschiedlichen Datenbanksystemen nutzen wollen, leistet die

[39] „Today there are a variety of commercial OODBs. In aggregate, it is a $75M/year market growing at about 50% per year." ([Silberschatz et al. 96])

[40] Das Datenbanksystem Poet zeigt auf, daß der ODMG-93-Standard von den Datenbanksystementwicklern nicht vollständig befolgt wird (zum Beispiel veränderte Namensgebung). Deshalb bedarf es trotz gleichen Paradigmas einer Abbildung.

[41] Neben der Einführung einer Abstraktionsschicht gibt es noch die Alternative, in jeder Klasse Konzepte von (relationalen) Datenbanksystemen zu modellieren (load-/store-Prinzip). Die komplexe Schnittstelle zur Abbildung zwischen den Modellen spiegelt sich dann allerdings in jedem Objekt wider.

[42] Dies bildet für Unternehmen einen Investitionsschutz.

C++-binding zusätzlich einen Beitrag zur Überbrückung der Kluft zwischen der OOPL C++ und den OODBS, weil die Funktionalität von Datenbanksystemen mit der Funktionalität der Programmiersprache C++ zusammengeführt wird. Dieser Beitrag der ODMG kann als ein Schritt zur Zusammenführung der Objektmodelle und der Funktionalität von OODBS und der OOPL C++ gewertet werden.

5.3.2 Abbildung zwischen Relationenmodell und Objektmodellen

Im letzten Unterabschnitt haben wir die Designentscheidung getroffen, die C++-binding des ODMG-93-Standards als gemeinsame Schnittstelle von Datenbanksystemen beliebiger Paradigmen zu wählen. Damit sich RDBS ODMG-konform präsentieren können, müssen sie in der Abstraktionsschicht, die die ODMG-konforme Schnittstelle bereitstellt, gekapselt werden.

Damit die Leistungsmerkmale relationaler Datenbanksysteme bei der persistenten Ablage von Objekten und deren Verwaltung genutzt werden können, muß folgendes beachtet werden:

- Abstimmung von Anfragen und Updates auf die Belange von RDBS
  (zum Beispiel Vermeidung von kostenintensiven *join*-Operationen)
- Minimierung des Zugriffs auf das RDBS
  (zum Beispiel durch Caching von Objekten oder deren relationalen Repräsentationen)
- Nutzung von Leistungsmerkmalen der RDBS
  (zum Beispiel Batch-Operationen über Mengen von Tupeln)[43]
- Angemessene Abbildung zwischen Objekten und Relationen

Während Kapitel 6 kurz auf die Gesichtspunkte der Kapselung eines Datenbanksystems eingeht, soll dieser Unterabschnitt einen Überblick über mögliche Abbildungen zwischen Objekten und Relationen darlegen[44].

> „Developers will find that the simplest mapping between objects and normalized table typically provides the best performance."
> ([Agarwal et al. 95])

[43] Weitere Leistungsmerkmale nennt [Agarwal et al. 95] (asynchrone Anfragen, stored procedures) und [Attia 96] (Indizes, Cluster)

[44] In diesem Zusammenhang gehen wir nicht auf die Zusicherung der Konsistenz zwischen den beiden Modellrepräsentationen durch Sperrenmanagement und Transaktionen ein. Ferner lassen wir an dieser Stelle die Abbildungen von Transaktionen, Integritätsbedingungen und Anfragesprachen außer acht, weil sie den Rahmen dieser Arbeit sprengen würden.

Diesem Leitsatz von Agarwal, Keene und Keller folgend schlagen wir die folgenden Abbildungen zwischen den fundamentalen Modellierungskonzepten des Relationen- und der Objektmodelle vor, die sich an dem in Abbildung 5.9 dargestellten Sachverhalt orientieren:

- *Klassen <-> relationale Tabellen*
- *Objekte <-> Tupel*
- *Beziehungen <-> Fremdschlüssel*
- *Vererbung <-> relationale Zeilen oder Verknüpfung von Tabellen*

„The most efficient mapping is most often the direct mapping of a class to a relational table."[45] [46] Nach [Agarwal et al. 95] bereitet diese Abbildung nur in zwei Fällen Probleme, die jedoch durch geeignete Maßnahmen bereinigt werden können.

- *Effizienzverlust, wenn nur Teile eines Objekts benötigt werden.*
  Lösung: „projection objects", d.h. Minimierung des Datenaustauschs durch Erstellung von Projektionen auf Ausschnitte einer Tabelle
- *Effizienzverlust, wenn mehrere Objekte immer gleichzeitig benötigt werden.*
  Lösung: „view objects", d.h. Minimierung des Datenaustauschs durch vordefinierte Datenbankabfragen

Die Abbildung von built-in Typen auf Attributtypen erweist sich als komplexer, weil nicht alle built-in Typen eines Objektmodells ihre Entsprechung im Relationenmodell finden (vgl. Abbildungen 5.3, 5.6 und 5.8). Zudem können Unterschiede bei der Repräsentation und der Größe gleicher Typen (zum Beispiel integer, float) bestehen. Die built-in Typen eines Objektmodells, die keine Entsprechung finden, und entwickler-definierte Typen können entweder bis auf die Ebene von built-in Typen fragmentiert[47] oder auf uninterpretierbare Bit- oder Zeichenfelder abgebildet werden. Diese sogenannten BLOB[48]-Felder können in Abhängigkeit des relationalen Datenbanksystems nur einmal je Tabelle verwendet werden. Darüber hinaus lassen sich keine Funktionen des Datenbanksystems auf diesen Feldern anwenden (zum Beispiel die Akkumulation von Werten).

„Relationships between objects map to foreign keys between rows."[49] Dabei gibt es zwei Varianten ([Agarwal et al. 95]):

[45] Dieses Vorgehen bedarf einer Abwandlung unter Einbezug von Vererbung.
[46] [Agarwal et al. 95]
[47] vgl. Datenbankautomat ([Kilberth et al. 93])
[48] **b**inary **l**arge **ob**ject
[49] [Agarwal et al. 95]

- *Ablage der Fremdschlüssel (Objektbeziehungen) in den zu den Klassen zugehörigen Tabellen*
  Dieses intuitive Vorgehen wird in der Praxis am häufigsten verwendet.

- *Bündelung aller Fremdschlüssel (Objektbeziehungen) in einer Tabelle*
  Bei dieser Variante werden alle existierenden Fremdschlüssel-Beziehungen zwischen Tabellen in einer Tabelle zusammengefaßt. Während das Hinzufügen oder das Entfernen von neuen Beziehungen in einem Datenbankschema, das nach der ersten Variante entwickelt wurde, zu erheblichen Veränderungen an den definierten Tabellenstrukturen führen kann, werden bei dieser Variante lediglich Tupel in einer Tabelle hinzugefügt oder entfernt. Neben der flexiblen Modellierung von Beziehungen zwischen Tabellen, überwiegt der Effizienzverlust beim Verfolgen einer Beziehung, weil eine Indirektion über eine Tabelle vorgenommen werden muß.

Neben der Abbildung von Beziehungen zwischen Objekten gibt es prinzipiell die Möglichkeit, Objekte, die miteinander in Beziehung stehen, in einer Tabelle abzubilden. Dieses Vorgehen stellt zwar die höchste Performanz bereit, verletzt jedoch die Normalisierung. Ferner schränkt dieses Vorgehen die Kardinalität der Objektbeziehung ein, weil bereits bei der Modellierung der 1-zu-n und n-zu-m Beziehungen die Größen n und m festgelegt werden müssen. Dies ist darauf zurückzuführen, daß für jedes Objekt, mit dem ein Objekt in Beziehung steht, eine Spalte in der Tabelle anzulegen und die Tabellenstruktur nicht dynamisch veränderbar ist. Abschließend sei noch erwähnt, daß die Klassen, die der Anwendungsdomäne entsprungen sind, so stark miteinander vernetzt sind, daß alle Klassen in einer Tabelle dargestellt werden müßten.

Für die Abbildung der Vererbungsbeziehungen stehen drei Alternativen zur Auswahl, die in Abhängigkeit der zu erwartenden Anfragen und der Verwendung von abstrakten und konkreten Klassen in einer Vererbungshierarchie vorteilhaft sind[50] [51]:

- *Vertikale Partitionierung*
  Jede Klasse in einer Vererbungshierarchie wird auf eine Tabelle abgebildet. Die Merkmale einer Unterklasse, die von einer Oberklasse geerbt worden sind, werden in der der Oberklasse zugeordneten Tabelle abgelegt. Das bedeutet, daß zwischen den Tabellen Verknüpfungen gebildet werden müssen[52]. Deshalb ist die Effizienz eines Zugriffs auf Objekte abhängig von der Lage der zugehörigen Klassen in der Vererbungshierarchie.

[50] „..., particular attention to expected query paths plays an important role on the mapping choice." ([Agarwal et al. 95])

[51] Für eine ausführliche Diskussion sei neben [Agarwal et al. 95] auch auf [Attia 96] verwiesen.

[52] Je tiefer die Klassenhierarchie, desto mehr *join*-Operationen und desto mehr Performanzverlust.

- *Horizontale Partitionierung*
  Nur die Blattklassen einer Vererbungshierarchie werden auf Tabellen abgebildet. D.h., daß alle Merkmale (auch die geerbten) auf eine Tabelle abgebildet werden. Während für abstrakte Klassen keine Tabellen benötigt werden, müssen konkrete Oberklassen in den Tabellen ihrer Unterklassen mit aufgenommen werden. Dieses Vorgehen stellt einen effizienten Zugriff auf Objekte der Blattklassen bereit.

- *Getypte Partitionierung*
  Alle Klassen einer Vererbungshierarchie werden in einer Tabelle abgebildet. Dabei wird eine zusätzliche Spalte für den Klassentyp eingeführt. Diese Art der Abbildung der Vererbung bietet die größte Performanz für Anfragen, jedoch wird die Normalisierung verletzt.

Damit eine korrekte Abbildung zwischen Klassen und Tabellen durchgeführt werden kann, bedarf es eines Meta-Objekt-Protokolls. Ein Meta-Objekt-Protokoll stellt Daten zur Verfügung, durch die Schlußfolgerungen über den Typ, die Struktur und das Verhalten möglich sind. Über ein Meta-Objekt-Protokoll kann unabhängig von der verwendeten Art der Partitionierung sichergestellt werden, daß Anfragen gegen Objekte auf die entsprechenden Tabellen und weiter auf die entsprechenden Zeilen und Spalten einer Tabelle umgesetzt werden. Ferner dient es zur Abbildung zwischen Merkmalen eines Objekts und den Attributen von Tabellen und spielt damit bei der Umsetzung von Polymorphie eine entscheidende Rolle.

Zur Umsetzung des Konzepts der Kapselung bedarf es des Einsatzes von Funktionen, die entweder mit Hilfe eines Meta-Objekt-Protokolls oder anhand ,,hartverdrahteter" Operationen ein korrektes Erzeugen, Zerstören, Laden und Speichern von Objekten gewährleisten.

Das Konzept der Objektidentität läßt sich am besten mit dem Konzept des Primärschlüssels umsetzen. Dabei ist jedoch Vorsicht geboten, weil die Primärschlüssel im Gegensatz zu Objektidentitäten veränderbar und nur innerhalb einer Relation eindeutig sind. Ein weiteres Problem, das sich bei einer Veränderung von Primärschlüsseln ergibt, ist das Auffinden von Objekten. Deshalb sollte die Unveränderbarkeit der Primärschlüssel zugesichert werden.

Das Basiskonzept der Zusammenführung von Daten und Funktionen in der Objektorientierung kann nicht auf relationale Datenbanksysteme abgebildet werden. So muß das Verhalten von Objekten von ihrer Struktur bei der persistenten Ablage getrennt werden. Ferner gibt es einen prinzipiellen Unterschied beim Zugriff auf Objekte bzw. Tupel in Tabellen. Während beim Relationenmodell die Daten via (ad-hoc) Anfragen gewonnen werden, ist bei der Objektorientierung der

navigierende[53] Zugriff vorherrschend. Eine direkte Abbildung der Objektnavigation auf Anfragen erweist sich als ineffizient, weil viele langsame Datenbankanfragen für nur einen Navigationsschritt durchgeführt werden müßten. Eine effiziente Abbildung kann nicht auf Basis der Modelle vorgenommen werden. Sie ist Aufgabe einer Komponente der Abstraktionsschicht[54].

Abschließend sei an dieser Stelle die Thematik vernetzter Objektstrukturen wieder aufgegriffen. Eine Abbildung auf das Relationenmodell vermag die vernetzte Struktur der Objekte und insbesondere die Komponentenbeziehungen zwischen den Objekten nicht sichtbar zu machen, weil komplexe Objekte auf mehrere Tabellen aufgeteilt werden und die Beziehungen zwischen den Tabellen keiner unterschiedlichen Semantik unterliegen[55]. Ferner tritt das Problem auf, daß ein Zugriff auf ein komplexes Objekt ineffiziente *join*-Vorgänge nach sich zieht. Somit veranschaulicht das komplexe Objekt den Strukturbruch (*impedance mismatch*) zwischen den Typsystemen des Relationenmodells und der Objektmodelle auf besondere Art und Weise.

5.4 Anforderungen

Nachdem wir uns auf die ODMG-93-Schnittstelle als gemeinsame Schnittstelle von Datenbanksystemen festgelegt haben und wir ferner die objektorientierte Programmiersprache C++ nutzen wollen, fällt uns der Schritt bei der Zusammenführung von DBS und OOPL leichter. Da sowohl der ODMG-93-Standard als auch C++ dem objektorientierten Paradigma unterliegen, bedarf es keiner aufwendigen Abbildung zwischen Paradigmen, auch wenn der ODMG-93-Standard und C++ abweichende Konzepte der Objektorientierung umsetzen. Vielmehr steht bei der Überbrückung des *impedance mismatch* zwischen OODBS und OOPL die Zusammenführung der Leistungsmerkmale im Vordergrund. Während DBS die sichere persistente Ablage von Objekten und die dazu benötigten Dienste zur Verfügung stellen, bieten OOPL die algorithmische Mächtigkeit zur Entwicklung von Anwendungen an.

Bevor wir auf die Thesen von Atkinson und Cattell eingehen[56], die bei der Ent-

[53] Anstatt von Navigieren wird häufig auch von Traversieren gesprochen, weil durch das Verfolgen von Objektbeziehungen Objekte erreicht werden.

[54] „For the initial select of a set of objects, ad-hoc access via the relational query mechanism using indexes is significantly faster than navigational access. Once the data has been retrieved from the database then in-memory navigational queries allow efficient use of the cached object information." ([Agarwal et al. 95])

[55] Auch bei der Abbildung der Vererbungsbeziehung geht die Semantik dieser Beziehung verloren, weil eine Vererbungshierarchie nicht aus den Tabellen hervorgeht.

[56] An dieser Stelle sei ferner auf das Thesen-Papier „The Third Manifesto" verwiesen ([Darwen, Date 95]), das ausschließlich Anforderungen an zukünftige Datenbanksysteme stellt.

wicklung von persistenz-unterstützenden Systemen beachtet werden sollten, stellt der nächste Unterabschnitt zunächst die unterschiedlichen Herangehensweisen bei der Zusammenführung von OOPL und DBS vor.

5.4.1 Überbrückung des impedance mismatch zwischen Programmiersprachen und Datenbanksystemen

Bei der Zusammenführung der Funktionalität von Programmiersprachen und Datenbanksystemen gibt es nach [Heuer, Saake 95] prinzipiell fünf Herangehensweisen:

1. *Programmiersprache mit einzelnen Datenbankoperationen anreichern*
   Diese Möglichkeit ist im Zusammenhang mit den Datenbanken, die auf dem Netzwerk- und dem hierarchischen Modell beruhen, verwirklicht worden.

2. *Vollständige Einbettung einer DML*[57] *in eine Programmiersprache*
   Stellvertretend für diese Herangehensweise ist *embedded SQL*, deren Anteile im Programmcode durch einen Precompiler in Prozeduraufrufe an das DBS umgesetzt werden.

3. *Vollständige Integration einer DML und einer Programmiersprache (Datenbankprogrammiersprache DBPL)*
   Bei dieser Möglichkeit der Zusammenführung wird das Typsystem einer Programmiersprache um ein Datenmodell eines DBS erweitert.

4. *Erweiterung einer Datenbanksprache um Kontrollstrukturen oder eine Makrosprache*
   Dieser Ansatz ist insbesondere bei 4GL-Sprachen[58] umgesetzt worden, bei denen meist ein SQL-Dialekt um Sprachkonzepte erweitert worden ist.

5. *Erweiterung einer Programmiersprache zu einer persistenten Programmiersprache*
   Bei dieser Alternative steht die Abstraktion von einem DBS im Vordergrund. Der Strukturteil der Programmiersprache wird auf den eines Datenbanksystems transparent abgebildet, und der Programmierer braucht sich mit dem Austausch der Daten zwischen Programmiersprache und Datenbanksystem nicht auseinanderzusetzen.

Übertragen wir die Zusammenführung der Funktionalität von Programmiersprachen und Datenbanksystemen auf die Objektorientierung, so bestehen drei unterschiedliche Herangehensweisen, die in Abbildung 5.10 dargestellt sind[59]:

[57] **d**ata **m**anipulation **l**anguage

[58] 4GL-Sprachen (*fourth generation languages*) sind speziell als Programmiersprachen für Anwendungen mit großen Datenmengen konzipiert worden.

[59] vgl. auch Abschnitt 2.3

1. *Programmiersprachlicher Ansatz*
   Es besteht die Möglichkeit, vorhandene OOPL um Konzepte der Persistenz zu erweitern. Dabei wird das Typsystem der Programmiersprache zum Datenmodell des entstehenden persistenten Speichers[60]. Die so entstandene PPL stellt ein transparentes[61] Management der persistenten Daten bereit. In [Heuer 92] wird bei dieser Herangehensweise von verhaltensmäßiger/operationaler Objektorientierung gesprochen. Meist liegt hier ein Mangel im Strukturteil des Objektmodells vor[62].

2. *Evolutionärer Ansatz*
   Intention dieses Ansatzes ist es, ein Datenbanksystem um Konzepte der Objektorientierung zu erweitern (vgl. Unterabschnitt 5.3). Dabei wird das Datenmodell eines Datenbanksystems um objektorientierte Konzepte angereichert, so daß objektorientierte Strukturen unterstützt werden. Die Abbildung von Verhalten wird meist vernachlässigt. In [Heuer 92] wird bei dieser Herangehensweise von struktureller Objektorientierung gesprochen, weil es an Objektverhalten mangelt[63].

3. *Revolutionärer Ansatz*
   Die dritte Möglichkeit der Zusammenführung ist die Neuentwicklung von OODBS, die die Mängel der beiden ersten Herangehensweisen behebt. Dabei wird meist die Neuentwicklung aus objektorientierter Datenbanksicht[64] gegenüber der Neuentwicklung aus Programmiersprachensicht[65] bevorzugt. In beiden Fällen konvergieren die entstehenden Systeme gegen dasselbe Endprodukt, das häufig auch als *persistentes Objektsystem* (*POS*[66]) bezeichnet wird. Erst dieses System kann als voll objektorientiert bezeichnet werden[67].

Beim Einsatz von persistenten Objektsystemen verschwimmt ,,die Unterscheidung von Anwendungsobjekten und Datenbankobjekten - in letzter Konsequenz haben wir eine Anwendung bestehend aus persistenten Objekten, die die Datenbank darstellen. Diese Sichtweise kollidiert mit dem bekannten Konzept der

[60] ,,... the ODBMS interface must fit seamlessly into the programming language environment. The programmer would like to use the ODBMS as if it were part of the language; the ODBMS should be invisible!" ([Loomis 93c])

[61] Unter Transparenz wollen wir in diesem Zusammenhang die implizite Ausführung von Funktionalität verstehen. Damit bleibt diese Funktionalität dem Entwickler verborgen.

[62] Kommerzielle Produkte, die auf C++ basieren, sind Poet, Ontos, Versant, und ein kommerzielles Produkt, das auf Smalltalk beruht, ist GemStone. Zumindest die auf C++ basierenden Systeme weisen kein transparentes Speichermanagement auf.

[63] Kommerzielle Produkte sind AIM/P, DASDBS, StarBURST, Postgres.

[64] OODBS erweitert um algorithmische Mächtigkeit

[65] OOPL mit Persistenzkonzepten und -diensten

[66] Abkürzung, **p**ersistent **o**bject **s**ystem

[67] Kommerzielle Produkte sind O2 und ORION/ITASCA.

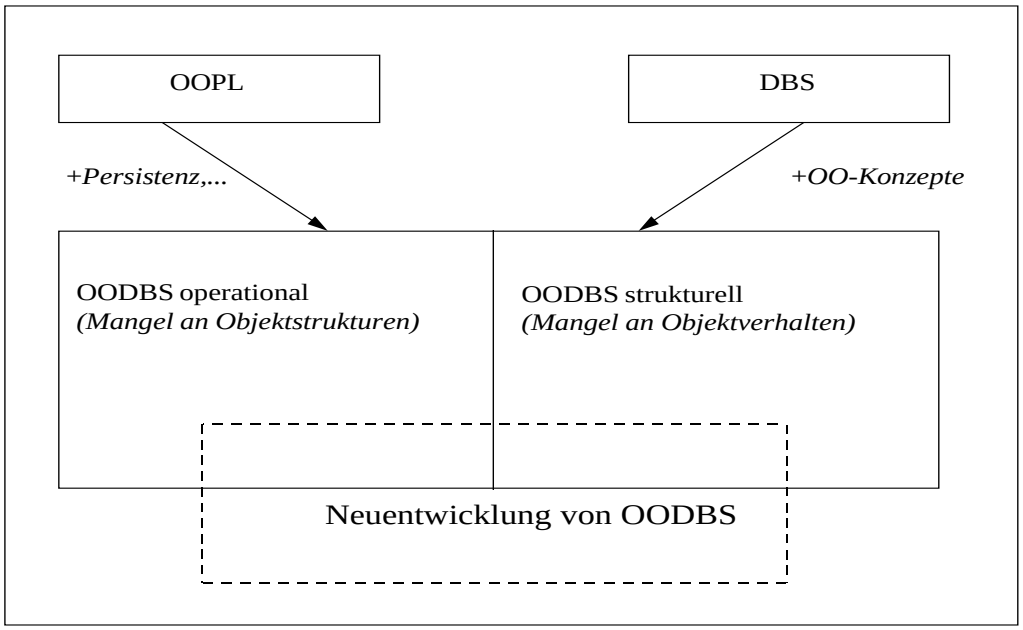


Abbildung 5.10: Zusammenführung von OOPL und DBS

logischen Datenunabhängigkeit, die der Schema-Architektur klassischer DBMS zugrunde liegt." ([Heuer, Saake 95])

Die nächsten Unterabschnitte gehen auf die Anforderungen ein, die an persistente Objektsysteme gestellt werden. Sowohl Atkinson als auch Cattell formulieren ihre Thesen aus der Perspektive der Entwicklung von Datenbanksystemen.

5.4.2 Thesen nach Atkinson

Atkinson unternimmt in [Atkinson et al. 90] den Versuch, Anforderungen an ein objektorientiertes Datenbanksystem zu definieren. Dazu beschreibt er die Leistungen und Charakteristika, die ein OODBS erfüllen muß. Obwohl dieses Thesen-Papier, das auch als Manifesto bezeichnet wird, bereits 1990 veröffentlicht worden ist, bildet es noch heute die Grundlage der Diskussion über die Merkmale von OODBS. Es ist bis heute nicht gelungen, einen Konsens über einen Standard für ein abstraktes Objektmodell aufzustellen ([Darwen, Date 95]).

Das Thesen-Papier untergliedert die Anforderungen in die drei Kategorien *unbedingt erforderlich*, *optional* und *Wahlmöglichkeit*. Diese können wiederum in Anforderungen an ein Datenmodell und an eine Datenbankprogrammiersprache[68] unterteilt werden. Die den Kategorien zugeordneten Punkte können nachfolgend unkommentiert genannt werden, weil diese Arbeit bereits das nötige Verständnis

[68] Anforderungen an eine Datenbankprogrammiersprache werden unterstrichen dargestellt.

für die Begrifflichkeit geschaffen hat:

- *UNBEDINGT ERFORDERLICH*:
  - Komplexe Objekte
  - Objektidentität
  - Kapselung
  - Klassen und Typen
  - Klassen- und Typhierarchien
  - Einfachvererbung
  - *overriding*, *overloading* und dynamisches Binden
  - *computational completeness*
  - Erweiterbarkeit[69]
  - Persistenz
  - Sekundärspeicher-Verwaltung
  - Nebenläufigkeit durch Transaktionen
  - Recovery-Mechanismen
  - Ad-hoc Anfragesprachen
- *OPTIONAL*:
  - Mehrfachvererbung
  - Statische Typisierung
  - Verteilung
  - Beliebige Transaktionen
  - Versionen[70]
- *WAHLMÖGLICHKEITEN*:
  - Zusätzliche Standard-Datentypen und Typkonstruktoren
  - Erweiterung des Typsystems (generische Typen, Typgeneratoren etc.)
  - Meta-Informationen
  - Paradigma der DBPL oder der Anfragesprache

[69] Definition eigener Klassen/Typen mit gleichem Verhalten wie Basis-Klassen/Basis-Typen

[70] Die Erzeugung von Versionen kann implizit durch das OODBS oder explizit vom Anwender angestoßen werden. Dazu bedarf es einer Verwaltung, die Konfigurationen zur Verfügung stellt und zwei Versionen wieder in eine überführen kann. Unter einer Konfiguration wird eine Menge von Versionen von Objekten verstanden, die untereinander konsistent sind.

Nach [Heuer 92] sind strittige Konzepte „von den verschiedenen Autoren des MANIFESTOS unterschiedlich beurteilt worden." Zu diesen zählen die Sichtdefinition[71], die Integritätsbedingungen[72], die Schemaevolution[73] sowie die Werkzeuge zur Datenbankadministration[74]. Bei diesen vier Punkten läßt das Manifesto eine Entscheidung offen, ob sie unbedingt erforderliche oder nur optionale Eigenschaften von OODBS sein sollten. Ob Verhalten und Struktur gemeinsam gespeichert werden sollen, wird im Thesen-Papier nicht beleuchtet.

5.4.3 Thesen nach Cattell

Cattell setzt sich in [Cattell 94] mit dem Entwurf von persistenten Objektsystemen auseinander, die er als *object data management systems* (ODMS) bezeichnet. An diese Systeme stellt Cattell in Form von 34 Thesen[75] Anforderungen, die er in *Datenbankfunktionalität*, *Objekte*, *Typsystem*, *Programmierung und Anfragesprachen*, *Datenbankarchitektur* sowie *weitere Fragestellungen* kategorisiert. Im folgenden präsentieren wir die Thesen, die wir frei aus dem Englischen übersetzt haben.

- Datenbankfunktionalität:
  - *Datenspeicher*
    1. These: **Persistenz**
       Ein ODMS muß Daten speichern, die ihren erzeugenden Prozeß überleben.
    2. These: **Schema**
       Ein ODMS muß ein Datenschema unterstützen, das der Benutzer einsehen und in dem er neue Typen erzeugen und löschen kann.
    3. These: **Sekundärspeicher**
       Ein ODMS muß effizient mit Hilfe von Sekundärspeichermanagement, Indizes, Links und anderen Techniken mit großen Datenvolumina umgehen.

[71] Wenn generische objekterhaltende Operationen möglich sind, dann lassen sich auf einfache Weise Sichten bilden.

[72] In [Heuer 92] wird die Meinung vertreten, daß neben den modellinhärenten Integritätsbedingungen zusätzliche fast überflüssig sind. So sind lediglich bei sehr komplexen und dynamischen Anwendungen allgemeine noch nicht inhärent vorhandene Integritätsbedingungen unverzichtbar.

[73] Eine Schemaevolution ist bei dynamischen Anwendungen wichtig.

[74] Ein werkzeug-gestützter Datenbankentwurf wird häufig ausgeklammert, wäre jedoch sehr nützlich.

[75] 33 Hauptthesen und eine Nebenthese

- *Datenzugriff*

  4. These: **Transaktionen**
     Ein ODMS muß Transaktionen oder einen äquivalenten Mechanismus für Nebenläufigkeitskontrolle und Fehlerbehebung bereitstellen.
  5. These: **Security**
     Ein ODMS sollte Mechanismen zur Benutzerautorisierung und zum Zugriffsschutz bereitstellen.

- Objekte:

  - *Objektidentität*

    6. These: **Objekte**
       Ein ODMS muß die Möglichkeit bereitstellen, Objekte zu speichern, und der Benutzer muß auf die Objekte über Primärschlüssel oder über vom System eineindeutig vergebene Objektidentitäten Bezug nehmen können.

  - *Attribute*

    7. These: **Attribute**
       Ein ODMS muß Objektattribute erlauben, die einfache Werte (integer, string), große Werte (Bitmap, Text) oder Sammlungen (Mengen, Listen) sein können.

  - *Beziehungen*

    8. These: **Beziehungen**
       Ein ODMS muß Mechanismen zur Darstellung von Beziehungen zwischen Objekten bereitstellen. Die Beziehungen sollten über Primärschlüssel oder Objektidentitäten definierbar sein.
    9. These: **Beziehungsintegrität**
       Der Mechanismus zur Darstellung von Beziehungen muß die Beziehungsintegrität bereitstellen, um Update-Probleme beim Löschen von Objekten und der Änderung von Beziehungen zu vermeiden.

  - *Zusammengesetzte Objekte*

    10. These: **Zusammengesetzte Objekte**
        Ein ODMS sollte die Möglichkeit bereitstellen, Objekte zu aggregieren, um Operationen wie Löschen, Kopieren und Nebenläufigkeitskontrolle auf diesem Objekt durchführen zu können. Wenn die Aggregation unterstützt wird, dann muß sie auch zum Zweck der physikalischen Clusterung nutzbar sein. Das Ausführen von Operationen auf das ganze Aggregat sollte optional sein.

  - *Prozeduren*

    11. These: **Prozeduren**
        Es sollte möglich sein, Prozeduren mit Objekten zu assoziieren.

- *Typsystem*

  - *Typrepräsentation*
    vgl. 2. These
  - *Vererbung*

    12. These: **Vererbung**
        Ein ODMS muß die Vererbung von Merkmalen von Oberklassen unterstützen. Optional sollten Mehrfachvererbung, mehrere Typen für ein Objekt und/oder „instance exceptions" (Singelton) bereitgestellt werden.
  - *Laufzeittypen*

    13. These: **Laufzeittypen**
        Ein ODMS sollte dynamisches Binden und einfache Änderungen an Typen erlauben.
  - *Schemaevolution*

    14. These: **Schemaevolution**
        Ein ODMS sollte Werkzeuge zur Unterstützung des Benutzers für Schematransformationen bereitstellen.

- Programmierung und Anfragesprachen:

  - *Anfragesprachen*

    15. These: **Anfragesprachen**
        Ein ODMS muß eine deklarative DML, die ad-hoc Operationen für den Benutzer bereitstellt, besitzen.
    16. These: **Physikalische Datenunabhängigkeit**
        Der Anfragesprachen-Verwalter muß einen hohen Grad an physikalischer Datenunabhängigkeit bereitstellen, bei der automatisch physikalische Zugriffsmethoden gewählt werden, um die Anfrage auszuführen.
    17. These: **Anfragen in Programmen**
        Ein ODMS muß eine Programmiersprache, die die Anfragesprache als Teilmenge enthält, unterstützen. Es muß möglich sein, die Anfragesprache von der Programmiersprache aus aufzurufen und umgekehrt.

- *Anfrageoptimierung*

  18. These: **Bevorzugung von Anfragen**
      Programme sollten den Datenzugriff besser in Anfragesprachen spezifizieren als direkt in der Programmiersprache, wenn die nötigen Operationen in der Anfragesprache bereitgestellt werden.

- *Anfragefähigkeiten*

  19. These: **Anfragesprachen-Vollständigkeit**
      Die DML muß mindestens die Leistungsmerkmale des Relationenkalküls mit Mengenergebnissen bereitstellen und sollte ferner den Zugriff auf alle Datenstrukturen einschließlich Sammlungen erlauben.

- *Programmiersprachenfähigkeiten*

  20. These: **Programmiersprachen-Vollständigkeit**
      Eine ODMS-Sprache muß die Fähigkeiten einer Programmiersprache (zum Beispiel *computational completeness*) unterstützen.

- *Integration von Programmiersprachen*

  21. These: **Integration von Programmiersprachen**
      Ein ODMS und bestehende Programmiersprachenumgebungen sollten integriert werden, um eine persistente Obermenge einer Anwendungsprogrammiersprache, mit der der Benutzer bereits vertraut ist, bereitzustellen.

- *Datenbankarchitektur und Integration*

  * Nebenthese
    Der Benutzer muß darüber urteilen, ob eine DBPL das Problem des *impedance mismatch* zwischen Sprachen löst.

- *Logische Datenunabhängigkeit*

  22. These: **Verbergung von Merkmalen**
      Ein ODMS muß es ermöglichen, öffentliche Merkmale (Attribute, Beziehungen, Prozeduren) von privaten zu unterscheiden.

  23. These: **Abbildung von Merkmalen**
      Ein ODMS muß es dem Benutzer erlauben, prozedurale und virtuelle Merkmale von Datenbankobjekten zu definieren. Virtuelle Merkmale sind Merkmale, die andere Merkmale verbergen können. Es sollte dem Benutzer transparent sein, ob ein Merkmal virtuell oder direkt gespeichert wird.

- *Standards*

  24. These: **Portabilität**
      Ein ODMS sollte es ermöglichen, Programme zu schreiben, die auch auf anderen ODMS lauffähig sind.

25. These: **SQL-Kompatibilität**
    Ein ODMS sollte abwärts kompatibel zu SQL sein oder ein Gateway zwischen dem ODMS und SQL-Datenbanksystemen bereitstellen.

- Datenbanksystemarchitektur:
  - *Zugriffsoverhead*

    26. These: **Minimaler Zugriffsoverhead**
        Ein ODMS muß den Overhead für einfache Datenoperationen, wie zum Beispiel Zugriff auf ein Objekt, minimieren.

  - *Caching und Clusterung*

    27. These: **Nutzung des Hauptspeichers**
        Ein ODMS muß die Wahrscheinlichkeit maximieren, daß die Daten sich beim Zugriff im Hauptspeicher befinden. Es sollte daher dem virtuellen Speicher der Anwendung mindestens ein Datencache bereitgestellt werden. Daneben sollte die Möglichkeit bestehen, die Daten auf geladenen Sekundärspeicherseiten/-segmenten zu clustern.

  - *Verteilung*

    28. These: **Operationen auf entfernte Daten**
        Es muß möglich sein, auf Daten zuzugreifen, die auf anderen Maschinen mit einem ODMS lokalisiert sind. Es muß außerdem möglich sein, ferne Operationen auszuführen, wenn diese Fähigkeit nicht durch Betriebssystemdienste bereitgestellt wird.

    29. These: **Verteilte Datenbanken**
        Ein ODMS sollte es ermöglichen, Anfragen und Programme auszuführen, die auf mehrere Datenbanken zugreifen. Objekte in einer Datenbank sollten Objekte in anderen Datenbanken referenzieren können.

- Weitere Fragestellungen
  - *Nebenläufigkeit*

    30. These: **Sperren**
        Ein ODMS muß einen Sperrmechanismus mit Objekt- oder Seitengranularität bereitstellen, so daß Daten über einen längeren Zeitraum ausgecheckt werden können.

    31. These: **Versionen**
        Ein ODMS sollte Primitive zur Verwaltung mehrfacher Versionen von Daten bereitstellen. Außerdem sollten Fähigkeiten von Konfigurationsverwaltungen bereitgestellt werden.

- *Regelsystem*

  32. These: **Regeln**
      Ein ODMS sollte einen Regelaufruf-Mechanismus bereitstellen, um Aktivitäten auszuführen, wenn festgelegte Prädikate wahr werden. Die mit einer Regel in Beziehung stehende Aktivität könnte eine beliebige Prozedur (Trigger) oder eine Fehlerbedingung (constraint) sein.

- *Höhere Werkzeuge*

  33. These: **Werkzeuge**
      Ein ODMS sollte höhere Werkzeuge bereitstellen, um den Zugriff auf die Daten und das Schema zu erlauben und den Programmierer bei der Generierung von Applikationen zu unterstützen.

Cattell fordert auf der einen Seite, die strikte Trennung von Datenbankprogrammiersprache und Programmiersprache (These 18), auf der anderen Seite schränkt er in der Nebenthese diese Aussage wieder ein, weil er die Entscheidung, ob eine Datenbankprogrammiersprache den *impedance mismatch* überbrückt, dem Benutzer überlassen möchte.

Im Gegensatz zu Cattell sind wir der Auffassung, daß die Datenbankfunktionalität vollständig in die Programmiersprache integriert werden sollte. Damit überwinden wir auf einfache Weise den *impedance mismatch* zwischen Datenbanksystem bzw. Datenbankprogrammiersprache und Programmiersprache. Somit muß der Entwickler nur noch eine Sprache erlernen und beherrschen. Darüber hinaus wollen wir noch einen Schritt weitergehen und einen Teil der Funktionalität der Datenbanksysteme transparent in die Programmiersprache integrieren. Diese Idee wird im weiteren Verlauf dieser Arbeit betrachtet.

5.5 Die persistente objektorientierte Programmiersprache

Von den drei unterschiedlichen Herangehensweisen zur Überbrückung des *impedance mismatch* setzt sich diese Diplomarbeit mit der Erweiterung der Programmiersprache C++ um Konzepte der Persistenz auseinander, d.h. wir erweitern C++ zu einer persistenten Programmiersprache. Diese Sprache sollte prinzipiell die Anforderungen sowohl des Manifestos ([Atkinson et al. 90]) als auch die Thesen von Cattell ([Cattell 94]) erfüllen. D.h. auch, daß eine Erweiterung einer Programmiersprache keinen Mangel im Strukturteil des Objektmodells aufweisen darf. Bevor wir im nächsten Kapitel unseren Lösungsansatz vorstellen, soll zunächst geklärt werden, was eine *persistente Programmiersprache* charakterisiert.

Die Entwicklung von *persistenten Programmiersprachen* ist seit 1979/1980 ein Forschungsgebiet. Die Initialzündung dafür waren die Schwierigkeiten bei der Ablage und der Rekonstruktion von CAD-/CAM-Datenstrukturen in bzw. aus einer Datenbank ([Atkinson, Morrison 89]). Programmiersprachen unterstützen typischerweise nur die Manipulation von transienten Datenstrukturen und bauen auf externen Datei- oder Datenbanksystemen zur persistenten Ablage von Datenstrukturen auf. Auf diese Kluft zwischen Programmiersprachen und Datenbanksystemen sowie auf die schwierige Verknüpfung beider wurde in dieser Diplomarbeit schon mehrmals hingewiesen. *Persistente Programmiersprachen* versprechen, diese Probleme zu überwinden. Die ersten Forschungsergebnisse mündeten in die persistenten Spracherweiterungen von Pascal (persistent PASCAL) und Algol (PS-Algol). Die Persistenz ist ein orthogonales Merkmal zum Paradigma einer Programmiersprache ([Atkinson, Morrison 89]). Wir gehen im Rahmen dieser Arbeit jedoch ausschließlich auf objektorientierte persistente Programmiersprachen ein.

Die Funktionalität einer Programmiersprache wird in der Regel um Funktionalität sowohl zum Datenbankzugriff als auch zur Behandlung persistenter Wurzelobjekte (*root objects*) und um ein einfaches Transaktionsmodell erweitert. In [Brössler, Freisleben 89] wird die Zielsetzung der persistenten Programmierung wie folgt definiert:

> „Persistent programming aims to provide a uniform method of accessing data structures, irrespective of their sizes and lifetimes and thus eliminates the conventional dichtomy between transient computational objects and permanent file or database objects. This not only avoids a very substantial number of duplicated mechanisms to access, protect and synchronize the two kinds of objects, but also does not force the programmer to flatten rich structures of computational objects before they can be made persistent. Thus, the programming effort is reduced, the code is simplified and the system performance is enhanced." ([Brössler, Freisleben 89])

Persistente Programmiersprachen (*PPL*) bieten einen unifizierenden sprachlichen Rahmen zur Entwicklung von Anwendungen, indem zur Modellierung ein einheitliches Objektmodell bereitgestellt wird und sowohl transiente als auch persistente Objekte in gleicher Weise manipulierbar und sondierbar sind. Eine persistente Programmiersprache stellt damit Konzepte zur persistenten Ablage ihrer Datenstrukturen zur Verfügung, ohne daß die Anbindung eines spezifischen persistenten Speichers in der Sprache sichtbar ist.

5.5.1 Funktionalität und spezielle Anforderungen

Aus diverser Literatur[76] haben wir die Funktionalität, die persistente Programmiersprachen bereitstellen können, identifiziert und zusammengetragen. Dabei sind uns viele Widersprüche (gegensätzliche Funktionalität) aufgefallen, die wir jeweils hinter einem Punkt in Klammern darstellen. Die unterstrichene Funktionalität sollte eine persistente Programmiersprache bereitstellen.

- Funktionen zum Öffnen und Schließen einer Datenbank (implizit oder explizit, abhängig von der Anbindung eines oder mehrerer Datenspeicher)
- einfaches Transaktionsmodell
- Funktionen zum Setzen, Erhalten und Löschen von Wurzelobjekten
- Unterstützung von Nebenläufigkeit
- Orthogonalität der Persistenz oder Persistenz beschränkt auf eine Untermenge von Typen
- Manipulation von persistenten Objekten (implizit oder explizit)
- CheckOut und CheckIn von Objekten (implizit oder explizit)
- (keine) ODL
- Ortsunabhängige Referenzen
- Unterstützung von Objektversionierung
- effiziente Umwandlung zwischen Objektrepräsentationen zur Unterstützung beliebiger Lebensdauern
- Transaktionsmanagement, Fehlererholung (Recovery)
- Zugriffsschutz

Viele persistente Programmiersprachen unterstützen nur einen Teil der nötigen Anforderungen. In [Brammer, Göhmann 96] und [Haines 95] wurden diese Mängel bei Vergleichen von PPL bereits herausgearbeitet.

Bisher wurde allgemein von Persistenz bzw. von der Bereitstellung von Persistenz gesprochen. Dabei haben wir außer acht gelassen, daß sich gerade bei der Bereitstellung der Persistenz drei grundlegende Prinzipien unterscheiden lassen ([Atkinson, Morrison 89]):

[76] u.a. [Haines 95], [Bischofberger et al. 96], [Heuer, Saake 95]

1. *persistence independence* (Transparenz der Persistenz)
   Die Lebensdauer eines Objekts ist unabhängig von der Art und Weise, wie das Programm es manipuliert. Auf der anderen Seite ist ein Programmfragment unabhängig von der Lebensdauer der Objekte, die es benutzt[77].

2. *persistent data type orthogonality* (Orthogonalität der Persistenz)
   Jedes Datenobjekt, egal welchen Typs, kann jede Lebensdauer haben.

3. *orthogonal persistent storage* (Orthogonalität des Sekundärspeichers)
   Das System, das Persistenz bereitstellt, ist unabhängig von der Wahl des persistenten Speichers.

Diese drei Prinzipien stellen zugleich Anforderungen an eine persistente Programmiersprache dar. Diese Sprachen sollten nämlich sowohl transparente als auch orthogonale Persistenz auf Basis beliebiger persistenter Speicher bereitstellen. Ferner lassen sich die Anforderungen aus Abschnitt 5.4 auf persistente Programmiersprachen übertragen, weil sie auch als persistente Objektsysteme und damit als objektorientierte Datenbanksysteme angesehen werden können[78] ([Haines 95]).

Der Orthogonalität des Sekundärspeichers wird in den meisten persistenten Programmiersprachen wenig Beachtung geschenkt. So wird bei Arjuna[79], Texas[80] und PANDA ausschließlich das Dateisystem als persistenter Speicher und bei E[81] und O++[82] die wenig bekannten Objektmanager EXODUS und Ode eingesetzt. Bei diesen Systemen findet eine enge Kopplung zwischen Programmiersprache und persistentem Speicher statt, bei der zwar die Gesichtspunkte der Performanz berücksichtigt sein mögen, jedoch auf die in der Praxis geforderte Flexibilität und

[77] „Aus Sicht der Applikationsentwicklung ist eine möglichst enge Integration der Datenbankoperationen mit der Programmiersprache wünschenswert, so daß auf persistente und transiente Objekte in gleicher Weise zugegriffen werden kann. Mehr oder weniger sind aber in allen Systemen besondere Zugriffsmechanismen notwendig, um persistente Objekte zu erzeugen, zu löschen und zu ändern." ([Hohenstein et al. 96])

[78] vgl. hierzu auch Unterabschnitt 5.4.1

[79] Arjuna ist Ende der achtziger Jahre unter der Leitung von S. K. Shrivastava an der University of Newcastle upon Tyne entwickelt worden ([Shrivastava et al. 88], [Dixon 88], [Dixon et al. 89]).

[80] Texas wurde Anfang der neunziger Jahre von Vivek Shingal, Sheetal V. Kakkad und Paul R. Wilson an der University of Texas in Austin entwickelt und für die Programmiersprache C++ implementiert ([Vaughan, Dearle 92], [Singhal et al. 92], [Kakkad et al. 92], [Wilson, Kakkad 92]).

[81] Die Programmiersprache E wird seit Ende der achtziger Jahre gemeinsam an der University of Wisconsin in Madison und dem IBM Almaden Research Center entwickelt ([Richardson, Carey 89],[Richardson, Carey 93]). E entstand im Rahmen des EXODUS Projekts an der University of Wisconsin.

[82] O++ ist eine in den AT&T Bell Laboratories in Murray Hill/New Jersey entwickelte persistente C++-Erweiterung ([Agrawal et al. 93]).

Anpaßbarkeit verzichtet wird[83]. Die Anbindung des Dateisystems mag zwar bei systemnahen Realisierungen als die einzig sinnvolle Lösung erscheinen, stellt aber für den realen Einsatz zur Entwicklung von Informationssystemen keinen praktikablen Weg dar. Der Datenbestand der Unternehmen ist so sensibel, daß nur den auf einem hohen Technologieniveau beruhenden Datenbanksystemen das nötige Vertrauen geschenkt wird. Trotzdem reicht die Anbindung eines Dateisystems zur Entwicklungszeit aus, wenn ein Wechsel auf ein beliebiges Datenbanksystem jederzeit möglich ist.

Persistente Programmiersprachen können durch konsequente Anwendung des Orthogonalitätsprinzips Modellierungs- und Manipulationsmechanismen, die in historisch unabhängig voneinander entstandenen Datenmodellen getrennt vorliegen, integrieren ([Mathiske 96b]). Damit wird die besonders ausgeprägte Trennung zwischen System- und Anwendungskomponenten von Informationssystemen überwunden. Persistente Programmiersprachen haben die Qualität datenintensiver Anwendungen auf zwei Ebenen entscheidend verbessert: Auf der Sprachebene liefern sie die algorithmische Mächtigkeit von Programmiersprachen ergänzt um ausgewählte Datenbankfunktionalität, wie zum Beispiel Anfragen, und erweiterte Objektmodelle, wie zum Beispiel eine Erweiterung um komplexe Objekte. Auf der Systemebene liefern sie eine passende integrierte Technologie, die die Kluft zwischen den nicht integrierten Datenbank- und Programmiersystemen überwindet.

5.5.2 Orthogonalität der Persistenz

Bereits in Unterabschnitt 2.2.2 haben wir die Frage nach der Art der Fortpflanzung der Lebensdauer in vernetzten Objektstrukturen aufgeworfen. Fassen wir die Frage allgemeiner, so lassen sich drei Möglichkeiten unterscheiden, wie Objekte die persistente Lebensdauer erlangen[84]:

1. *Persistence by Type*[85]
   Objekttypen, von denen persistente Exemplare erzeugt werden sollen, müssen in geeigneter Form gekennzeichnet werden: Entweder durch Deklaration mit Hilfe eines neuen Schlüsselwortes *persistent* oder durch Erben von ei-

[83] „Ein späterer Umstieg auf ein Konkurrenzprodukt ist allerdings wegen der Verschiedenartigkeit der [Datenbank-]Systeme mit erheblichem Portieraufwand verbunden. In der Vergangenheit scheuten aus diesem Grund noch viele potentielle Kunden vor dem Einsatz eines OODBMS zurück, obwohl ihnen die OODBMS-Technologie zahlreiche Vorteile gegenüber anderen Datenbank-Technologien oder gar Dateisystemen gebracht hätte." ([Hohenstein et al. 96])

[84] „In manchen [Datenbank-]Systemen wird die Persistenz über eine Mischform aus den .. genannten Ansätzen realisiert, in anderen wird die Persistenz von Objekten bei den verschiedenen Sprachschnittstellen (C++, Smalltalk etc.), die das System anbietet, unterschiedlich gehandhabt." ([Hohenstein et al. 96])

[85] klassenabhängige Persistenz ([Haines 95], [Heuer, Saake 95], [Cattell 94])

nem Basistyp *PERSISTENT*. Die Kennzeichnung eines Obertyps kann sich auch auf seine Untertypen übertragen. In Abhängigkeit der technischen Realisierung beider Alternativen sind entweder alle Exemplare des Objekttyps automatisch persistent, oder beim Erzeugen eines Exemplars wird entschieden, ob dieses transient oder persistent angelegt werden soll.

2. *Persistence by Explicit Call*[86]
   Bei dieser Möglichkeit wird zur Laufzeit die Lebensdauer eines Exemplars einer Klasse spezifiziert. Dieses kann bereits zum Erzeugungszeitpunkt aber auch zu beliebigen anderen Zeitpunkten vorgenommen werden. Daraus ergibt sich, daß die Lebensdauer eines Objekts zur Laufzeit veränderbar ist.

3. *Persistence by Reference/Reachability*
   Die dritte Alternative beruht auf der Auszeichnung von Objekten (persistenten Wurzelobjekten) im Objektgraphen. Alle von diesen Wurzelobjekten direkt oder indirekt erreichbaren Objekte sind ebenfalls persistent.

In [Heuer 92] wird bei *Persistence by Type* sowie bei *Persistence by Explicit Call* von expliziter Persistenz[87] und bei *Persistence by Reference/Reachability* von Persistenz durch Erreichbarkeit[88] gesprochen.

Bei vernetzten Objektstrukturen, die ausschließlich Kompositionsbeziehungen beinhalten, ist die implizite Persistenz (durch Erreichbarkeit) auf elegante Weise einsetzbar, weil durch die Auszeichnung des Kontextobjekts auch die Komponentenobjekte persistent sind. Bei vernetzten Objektstrukturen hingegen, die sowohl Kompositionsbeziehungen als auch Verknüpfungsbeziehungen beinhalten, erscheint diese Form der Persistenzfortpflanzung als wenig geeignet. Weil die Objekte eines Informationssystems auf undurchschaubare Weise miteinander in Beziehung stehen, könnte die Auszeichnung eines einzigen Objekts als persistentes Wurzelobjekt die Persistenz aller Objekte nach sich ziehen.

In Abhängigkeit von der Realisierungstechnik der Persistenz kann das Prinzip der Orthogonalität der Persistenz nur zu einem gewissen Grad umgesetzt werden[89].

[86] objektabhängige Persistenz ([Haines 95], [Heuer, Saake 95], [Cattell 94])

[87] „*Explizite Persistenz*: Jedes Objekt, das persistent sein soll, muß einer persistenten Klasse (klassenabhängige Persistenz) angehören oder als persistent erzeugt worden sein (objektabhängige Persistenz). Explizite Persistenz gibt es etwa in ObjectStore." ([Heuer 92])

[88] „*Persistenz durch Erreichbarkeit*: Jedes Objekt, daß in einem Objektgraphen von einem anderen, persistenten Objekt aus erreicht werden kann, ist auch persistent. Persistenz durch Erreichbarkeit gibt es etwa in PS-Algol, GemStone oder O2." ([Heuer 92])

[89] „Von einem OODBMS [(nach Abschnitt 5.4 also auch von PPL)] wird orthogonale Persistenz gefordert, d.h. jedes Objekt kann unabhängig von seiner Klassenzugehörigkeit persistent werden. Persistenz ist somit keine Eigenschaft von bestimmten, ausgewählten Klassen. Schließlich sollte Persistenz implizit sein: es darf nicht in der Verantwortung des Programmierers/Benutzers liegen, persistente Objekte explizit zu speichern. Diese Forderung wäre beispielsweise durch erforderliche Aufrufe einer Routine store_object verletzt." ([Geppert 97])

Deshalb werden auch *halb-persistente objektorientierte Programmiersprachen* von *persistenten objektorientierten Programmiersprachen* unterschieden. Bei halb-persistenten objektorientierten Programmiersprachen müssen alle Objekte, die persistent ablegbar sein sollen, von einer Klasse[90] erben[91], in der gewisse Datenbankfunktionalität spezifiziert oder bereits implementiert ist. Über diese Klassen können dann Anfrage- und Update-Operationen von OODBS genutzt werden, und die Objekte sind somit durch das Datenbanksystem verwaltbar. In einigen Systemen[92] muß der Programmierer die Methoden zum Laden und Speichern von Objekten ausimplementieren. Diese Methoden müssen teilweise explizit aufgerufen werden (load-/store-Prinzip), wodurch die Transparenz der Persistenz eingeschränkt wird. Kommerzielle Ausprägungen sind u.a. die C++-Schnittstellen der objektorientierten Datenbanksysteme Poet und Ontos. Halb-persistente objektorientierte Programmiersprachen widersprechen dem Grundsatz des Manifestos ([Atkinson et al. 90]), orthogonale, d.h. typ-unabhängige, Persistenz bereitzustellen.

Persistente objektorientierte Programmiersprachen genügen der orthogonalen Persistenz. Alle Objekte gleich welchen Typs können persistent gehalten werden, ohne daß von einer persistenz-unterstützenden Klasse geerbt werden muß. Somit sind generische Anfrage- und Update-Operationen auf die Exemplare beliebiger Klassen, Mengen- und "Collection"-Typen uneingeschränkt anwendbar. Die bekannteste kommerzielle Ausprägung ist die C++-Schnittstelle des objektorientierten Datenbanksystems ObjectStore.

5.5.3 Transparenz der Persistenz

Die Abstraktion von der vertikalen Speichertrennung, die auch als vertikale Ortstransparenz bzw. Transparenz der Persistenz bezeichnet wird, hat Anfang der neunziger Jahre ihren Durchbruch gehabt. Sie stellt für [Wai 89] ein fundamentales Konzept zur Bereitstellung der Persistenz dar. Die Transparenz der Persistenz basiert auf dem virtuellen Speichermanagement, dessen Aufgabe es ist, die aktuell benötigten Objekte im transienten Speicher zu halten. Wann die Objekte in den persistenten Speicher zurückgeschrieben und aus diesem wieder ausgelesen werden, bleibt dem Programmierer verborgen. E wird in der Literatur als die erste persistente Programmiersprache bezeichnet, die transparente Persistenz unterstützt.

[90] Mixin-Klasse ([Booch 94])
[91] direkt oder indirekt
[92] vgl. Arjuna

Begriff 39 : *Der virtuelle Speicher*
Ein virtueller Speicher ist ein logischer Speicherraum, der physikalisch aus den Komponenten Primärspeicher (Hauptspeicher) und Sekundärspeicher (Festspeicher: Dateien oder Datenbanken) besteht. Der Speicherraum ist in Speicherseiten gleicher Größe aufgeteilt, die mit der Methode des Paging zwischen dem Primär- und dem Sekundärspeicher transparent ausgetauscht werden.

Bei der Umsetzung einer auf virtuellem Speicher realisierten Architektur gibt es zwei unterschiedliche Ansätze: Ein Ansatz nutzt den transienten Speicher als Objektcache und stellt eine Hash-Tabelle[93] zur Lokalisierung der Objekte bereit.

Begriff 40 : *Cache (nach [Giloi 93])*
Ein Cache ist einem Speicher, dessen Zugriffszeit wesentlich höher und dessen Speicherkapazität wesentlich größer als die des Caches ist, zugeordnet. Der Cache enthält Kopien von Teilen dieses Speichers und muß für den Programmierer transparent sein, d.h., daß der Programmierer weder von der Existenz des Caches wissen noch sich um dessen Kontrolle kümmern muß.

Bei einem anderen Ansatz werden immer die Speicherseiten des virtuellen Speichers, auf denen sich die aktuell bearbeiteten Objekte befinden, im transienten Speicher gehalten. Beide Ansätze unterscheiden sich hinsichtlich der Granularität des Transfers zwischen Primär- und Sekundärspeicher. Während beim ersten Ansatz das Objekt das Granulat ist, stellt beim zweiten Ansatz eine Seite das Granulat dar. Für eine ausführliche Diskussion über die Granulatgrößen und deren Auswirkungen auf Transferkosten, Zugriffsperren und Transaktionslängen sei auf [Brammer, Göhmann 96] verwiesen[94].

Bei einer auf virtuellem Speichermanagement basierenden Architektur können die Referenzen zwischen Objekten nicht auf Basis der transienten Hauptspeicheradressen der Objekte ausgedrückt werden[95]. Statt dessen wird sich der Adressen des virtuellen Speichers bedient, wodurch eine eindeutige Referenzierung von Objekten im virtuellen Speicher gewährleistet werden kann. Die Transformation der Referenzen in gültige Hauptspeicheradressen findet entweder beim Transfer der Objekte zwischen dem Sekundär- und dem Primärspeicher oder erst zum Zeit-

[93] Jede Form einer Tabelle oder Liste ist prinzipiell möglich. Eine Hash-Tabelle gewährleistet jedoch einen performanten Zugriff.

[94] An dieser Stelle sei auch auf den Unterabschnitt 4.2.2 und den Abschnitt 5.6 verwiesen.

[95] Die transienten Hauptspeicheradressen sind nur gültig, solange sich die Objekte im Hauptspeicher befinden und dort nicht verschoben werden. Weder im persistenten Speicher noch nach einem erneuten Laden in den transienten Speicher ist über sie eine korrekte und eindeutige Referenzierung möglich.

punkt der Dereferenzierung einer Referenz statt. Dieser Vorgang wird auch als *pointer swizzling*[96] bezeichnet.

Begriff 41 : *pointer swizzling*
Unter *pointer swizzling* wird die Konvertierung von einem Zeigerformat in ein anderes verstanden.

Persistente Programmiersprachen, die auf dem virtuellen Speichermanagement beruhen[97], skalieren schlecht in Bezug auf Verteilung. Das Konzept des virtuellen Speichermanagements stellt keine Unterstützung bei der Entwicklung von Anwendungen, die auf verteilten Systemen basieren, bereit. PANDA erweitert den virtuellen Speicher zu einem *Distributed Shared Memory*, der den gemeinsamen Zugriff auf Objekte auch über das Ausführungsende eines Programms hinaus erlaubt. Dabei werden jedoch die ausgereiften Konzepte der Datenbanksysteme weder genutzt noch dem Programmierer bereitgestellt. Im nächsten Abschnitt werden wir deshalb die Zusammenführbarkeit der transparenten Persistenz des virtuellen Speichers und der transparenten Mobilität des *Distributed Shared Memory* darlegen.

5.6 Das Konzept des Distributed Shared Memory

Im letzten Abschnitt sind wir bereits ausführlich auf die transparente Persistenz und damit auf die Aufhebung der vertikalen Speichertrennung in persistenten Programmiersprachen eingegangen. Das Konzept des virtuellen Speichers erweist sich als leistungsstark ([Singhal et al. 92], [Wilson, Kakkad 92]); es beschränkt sich jedoch ausschließlich auf Einzelplatzsysteme, weil der virtuelle Speicher keine Konzepte für einen gemeinsamen Zugriff und dessen Koordination liefert. Da Informationssysteme verteilte Systeme sind und wir an der Idee der vertikalen Speicherabstraktion festhalten wollen, bedarf es eines leistungsfähigen Konzepts zur Verknüpfung von Rechnerknoten.

[96] swizzling engl. (Umrühren alkoholischer Mixgetränke)
[97] zum Beispiel Texas

5.6.1 Einordnung

> ,,A paradigm that is gaining popularity, and one that you may be seeing in a lot of products in the future, is *virtual shared memory*. It would be nice to treat the distributed memory parallel machine as if every processor was tapped into a single, large pool of memory, and let some portion of the environment - software or hardware - back up the illusion by shuffling the data around transparently. Underneath there would still be seperate memories, but the programmer wouldn't have to worry about it - at least not *too* much. As you can imagine, data placement and locality of reference would still be important, though the details of moving data around would be hidden.
> Providing the illusion of a simple integrated memory system to a distributed memory computer is a challenging problem."
> (aus [Mairandres 96] von Kevin Dowd).

Verfolgen wir die herkömmliche Idee, die Rechnerknoten explizit zu verbinden, dann sieht der Entwickler das Informationssystem als ein ,,Universum", das aus einer Vielzahl von Speichern - auch als Regionen bezeichnet - besteht. Diese Regionen sind sowohl auf Sprachebene als auch Anwendungsebene sichtbar. [Daerle et al. 91] spricht bei einer solchen Sicht auf ein Informationssystem von einem *federated-stable-store*. Das zugrundeliegende Konzept ist das der heterogenen, partitionierten Speicherstruktur, die sich automatisch aus der zugrundeliegenden Netzwerkstruktur (zum Beispiel Client/Server-Struktur) ergibt. Den Vorteilen der Einflußnahme des Programmierers auf die Lokation von Objekten, der expliziten physischen Clusterung logisch zusammenhängender Daten sowie der unkomplizierten Fehlerbehandlung durch den Programmierer stehen die Nachteile der fehlenden technischen Transparenz und des explizit zu programmierenden Austauschs von Daten zwischen den Speichern gegenüber.

Da wir als eine Anforderung an die Entwicklung von Informationssystemen die technische Transparenz gestellt haben[98], verfolgen wir die Idee des *Distributed Shared Memory*, den [Daerle et al. 91] auch als Realisierung des *one-world-model* bezeichnet. Hinter diesem Konzept verbirgt sich die Vorstellung, die verteilte Struktur eines Informationssystems (zum Beispiel eine Client/Server-Struktur) zu verstecken und somit die technische Ortstransparenz bereitzustellen. Der Programmierer sieht das ,,Universum" als einen einzigen großen Adreßraum, in dem jedes Objekt eine systemweit eindeutige Identität besitzt. Um die Lokation von Objekten auf unterschiedliche Rechnerknoten braucht sich der Programmierer damit nicht zu kümmern. Im Gegensatz dazu sollte ein Benutzer Wissen über fachliche Orte (zum Beispiel Arbeitsumgebungen) besitzen. Dieses stellt zum Konzept des *Distributed Shared Memory* keinen Widerspruch dar, weil die fach-

[98] vgl. Unterabschnitt 2.3.4

liche Transparenz ein höheres Konzept ist und auf dem des DSM auf einfache Weise aufbauen kann[99].

> ,,A more recent and increasingly popular model for supporting distribution is Distributed Shared Memory (DSM). The idea is essentially an extension of virtual memory to encompass a network. Each object is located on a particular node via a unique (network-wide) virtual address and all objects may potentially be addressed from any node. The system transparently copies remote objects on the first access and maintains a coherent view of the data at all machines." ([Daerle et al. 91])

Der *Distributed Shared Memory* (*DSM*) ist seit Anfang der achtziger Jahre ein Forschungsgebiet. Die ersten Ergebnisse mündeten im *Apollo Domain file system*, das Speicherseiten zwischen Dateien und physikalischem Speicher auf unterschiedlichen Rechnerknoten austauscht. ,,Mit IVY [100] wurde zum erstenmal die Realisierung von virtuell gemeinsamen Speicher [101] als Implementierung auf Ebene des Betriebssystems nachgewiesen." ([Mairandres 96])

Heute hat das Konzept des DSM große Bedeutung bei Multiprozessorsystemen mit Parallelverarbeitung errungen ([Coulouris et al. 94]). Dennoch ist die Entwicklung noch nicht weit fortgeschritten, ,,und viele der Systeme und Konzepte sind momentan noch Gegenstand der Forschung. Es ist nicht auszuschließen, daß zukünftige neue Ideen prägenden Einfluß auf die Arbeit haben werden. Welche der Formen von verteiltem gemeinsamen Speicher sich schließlich durchsetzen werden, wird die Zukunft zeigen." ([Mairandres 96])

> **Begriff 42 :** *Distributed Shared Memory (DSM)*
> Der Distributed Shared Memory (verteilter gemeinsamer Speicher) abstrahiert von physikalisch horizontal verteilten Speichern. Er stellt einen logisch gemeinsamen Speicher bereit, der sich wie ein gewöhnlicher Speicher verhält. Durch diese Abstraktion stellt der DSM technische Transparenz bezüglich der Verteilung der Rechnerknoten und der gemeinsamen Nutzung von Objekten zur Verfügung. Die Objekte sind über netzwerkweit eindeutige, virtuelle Adressen adressierbar. Üblicherweise stellt ein DSM keinen Transaktionsmechanismus und keine Anfragesprache bereit.

Das Haupteinsatzgebiet ist in der gemeinsamen Nutzung von Daten zu sehen, die von unterschiedlichen Prozessen auf (unterschiedlichen) Rechnerknoten ohne gemeinsamen physikalischen Speicher genutzt werden. Dieses kann eine parallele

[99] vgl. auch die Ausführungen in Abschnitt 4.3
[100] **I**ntegrated shared **V**irtual memory system designed at **Y**ale
[101] andere Bezeichnung für *Distributed Shared Memory*

oder verteilte Anwendung aber auch eine Gruppe von individuellen Anwendungen oder ein Multiprozessorsystem sein.

Das Konzept des DSM kann auf unterschiedlichen Systemebenen umgesetzt werden:

- *hardware-basiert*
  Bei dieser Herangehensweise sind Prozessoren und Speichermodule über Hochgeschwindigkeits-Netzwerke verbunden. Die Hardware stellt sowohl den Zugriff auf den DSM als auch dessen Kontrolle bereit.
  Beispiele: Multiprozessorarchitekturen DASH, PLUS

- *betriebssystem-basiert* (seiten-basiert)
  Der betriebssystem-basierte Ansatz beruht auf dem Prinzip des virtuellen Speichermanagements. Der Kernel des Betriebssystems sichert die Konsistenz über *page fault handling*[102] zu.
  Beispiele: Mach, Chorus, IVY, COOL, Clouds

- *bibliotheks-basiert* (programmiersprachen-basiert)
  Bei dem bibliotheks-basierten Ansatz wird das Konzept des DSM auf Programmiersprachenebene umgesetzt. Dazu bedarf es der Einbindung einer Run-Time-Bibliothek in die jeweilige Anwendung, die einen normalen Speicherzugriff bereitstellt. Die Bibliothek stellt Mechanismen zur Konsistenzsicherung zur Verfügung.
  Beispiele: Orca, Linda, eingeschränkt PANDA

Die Vorteile beim Einsatz eines Distributed Shared Memory liegen unabhängig von der Herangehensweise klar auf der Hand: Der Entwickler von Anwendungen kann das vertraute und wohldefinierte Programmiermodell mit den bereitgestellten Datenstrukturen und Datenzugriffen verwenden, ohne sich über Gesichtspunkte der Beweglichkeit Gedanken machen zu müssen (technische Transparenz). Dem Programmierer bleibt deshalb gegenüber dem *federated-stable-store*, der auf dem Nachrichtenaustausch zwischen Rechnerknoten basiert, das *marshalling*[103] und *unmarshalling*[104] verborgen und somit erspart. Außerdem kann der bereits vorhandene Code von Anwendungen leicht portiert werden. Insgesamt ergibt sich deshalb eine deutliche Reduzierung der Entwicklungszeit von Informationssystemen.

Dem steht das Performanzproblem bei der Skalierung des verteilten Systems

[102] Nähere Ausführungen finden sich in [Brammer, Göhmann 96] bei der Darstellung der Programmiersprachenerweiterung Texas.

[103] Unter *marshalling* wird die Konvertierung von einer Datenrepräsentation in eine andere verstanden.

[104] Umkehrvorgang zu einem bestimmten *marshalling*-Vorgang

gegenüber. Der Kommunikationsaufwand insbesondere zur Konsistenzsicherung steigt mit der Anzahl der Rechnerknoten überproportional[105] [106]. Außerdem besteht ein relativ hoher aber einmaliger Implementierungsaufwand zur Umsetzung der Mechanismen des DSM. Ferner bleibt das Wissen über den physikalischen Aufenthaltsort eines Objekts im verteilten System verborgen.

5.6.2 Charakteristika

Der Distributed Shared Memory läßt sich nach [Coulouris et al. 94] durch folgende sechs Merkmale charakterisieren, die unterschiedlich ausgeprägt sein können:

- *Struktur* (Struktur der Daten, die im DSM liegen)
  Ein Anwendungsprogrammierer hat auf den Inhalt der Daten im DSM entweder die Sicht auf Bytes, eine Sammlung von sprachspezifischen Datenstrukturen, eine Sammlung von gemeinsam genutzten Objekten oder eine Sammlung unveränderbarer Daten. Bei einer byte-orientierten Struktur erfolgt der Zugriff wie auf einen normalen virtuellen Speicher. D.h., daß die Daten eine beliebige Struktur einnehmen können. Der Ansatz der sprachspezifischen Sammlung von Datenstrukturen paßt sich dem Programmierparadigma einer Programmiersprache an. In unserem Fall sind die sprachspezifischen Datenstrukturen Objekte. Beim Zugriff auf Objekte in einer Sammlung von gemeinsam genutzten Objekten erfolgt dieser indirekt über die Sammlung. Die Sammlung stellt die Synchronisation sicher, die direkt auf der Operationenebene der Objekte ausgeführt wird. D.h., daß die Operationen auf einem Objekt serialisiert werden. Auch bei der Sammlung unveränderbarer Daten wird indirekt über die Sammlung auf die Daten zugegriffen. Im Gegensatz zur vorhergehenden Sammlung darf der Zugriff nur lesend erfolgen. Modifikationen ergeben neue Versionen der Daten, die wiederum in der Sammlung abgelegt werden können.

- *Synchronisationsmodell* (Bereitstellung der Konsistenz für die Anwendung)
  Ein Synchronisationsmodell stellt durch die atomare Ausführung von mehreren Operationen, die sich in einem kritischen Abschnitt befinden, die Konsistenz der Daten im DSM sicher. Prinzipiell sind die fachlichen von den technischen Synchronisationsmodellen zu trennen. Weil auf fachliche Verfahren bereits ausführlich in Abschnitt 4.2 eingegangen worden ist, sei an

[105] Dieses Problem kann sich in Abhängigkeit der Mechanismen zur Konsistenzerhaltung (siehe unten) in der Tragweite verändern. Zudem wird nur ein Bruchteil aller Daten während einer gewissen Zeit wirklich gemeinsam genutzt.

[106] Beim hardware-basierten Ansatz, bei dem die Homogenität der Prozessoren vorausgesetzt wird, bedarf es eines Verbindungsnetzes mit ,,vernünftiger" Geschwindigkeit, damit eine hohe Performanz erreicht wird. Ferner findet eine Leistungssteigerung durch eine Speicherverwaltungseinheit mit Zugriffsmechanismen auf Seitenebene statt.

dieser Stelle ein technisches kurz diskutiert[107].

Durch die Bereitstellung von Locks oder Semaphore wird der wechselseitige Ausschluß auf gemeinsam genutzte Daten zugesichert. Ein Semaphor ist eine Variable, auf die nach ihrer Initialisierung nur durch die zwei Anweisungen *Setzen* und *Entfernen* zugegriffen werden kann. Wenn ein Semaphor von einem Prozeß gesetzt ist, dann kann nur dieser auf das Datum, das dem Semaphor zugeordnet ist, lesend und/oder schreibend zugreifen. Alle anderen Prozesse sind bis zur Freigabe des Semaphors blockiert. Setzen mehrere Prozesse unabhängig voneinander mehrere Semaphore bzw. Locks, dann kann es zu *Verklemmungen* oder *deadlock*-Situationen kommen[108].

- *Konsistenzmodell* (Mechanismen zur Konsistenzerhaltung)
  Die zentrale Frage, die sich bei einer Untersuchung von Konsistenzmodellen stellt, ist: Welches Ergebnis erhält ein lesender Zugriff, wenn gleichzeitig schreibende Zugriffe stattfinden? In Abhängigkeit von der Antwort lassen sich die Modelle der schwachen und der starken Konsistenz unterscheiden. Während bei der schwachen Konsistenz der Lesezugriff irgendein Schreibergebnis zurückgibt, wird der Lesezugriff bei der starken Konsistenz mit dem jüngsten Schreibergebnis beantwortet.

  Ein starkes Konsistenzmodell ist das *sequential consistency* Modell, bei dem lesende und schreibende Zugriffe eines Prozesses in der vom Programmierer festgelegten Reihenfolge ausgeführt und beantwortet werden. Lesende und schreibende Zugriffe mehrerer Prozesse werden serialisiert. Den Vorteilen der Konsistenz zu jedem Zeitpunkt und der Verborgenheit der Mechanismen zur Konsistenzsicherung für den Programmierer stehen die Nachteile einer ineffizienten Implementation und der Trend der Algorithmen zum *thrashing* (siehe unten) gegenüber.

  Das *release consistency* Modell stellt dem Entwickler Synchronisationsobjekte (Locks) zur Konsistenzwahrung zur Verfügung, die neben den normalen Speicherzugriffen Verwendung finden. Die Funktionalität, die die Synchronisationsobjekte nach [Coulouris et al. 94] bereitstellen sollten, sind *acquireLock*, *releaseLock* und *waitAtBarrier*[109]. In Abhängigkeit der Update-Option (*write-update* oder *write-invalidate*) werden die Veränderungen an den Objekten anderen Prozessen nach der Aufhebung der Sperre kundge-

[107] Zur fachlichen Synchronisation sei noch ergänzt, daß in der Regel die Update-Transfers reduziert werden und die Daten im DSM sich nicht unbedingt in einem konsistenten Zustand befinden müssen.

[108] [Jessen, Valk 87] geht ausführlich auf diese Problematiken ein.

[109] Angemeldete Prozesse werden solange geblockt, bis die Sperre aufgehoben wird und alle Prozesse fortfahren können.

tan. Dem Nachteil, daß der Programmierer die leicht veränderte Semantik des Speicherzugriffs antizipieren muß[110], stehen die Vorteile einer effizienten Implementation und die Anpassung der Sperrsemantik an die Anwendungsdomäne gegenüber. [Coulouris et al. 94] kommt zu dem Schluß, daß das *release consistency* Modell für den DSM am besten geeignet ist[111].

- *Update-Optionen* (Form der Propagierung von Update-Operationen)
  Das Propagieren von Veränderungen an Datenbeständen, mit denen mehrere Prozesse arbeiten, sind von elementarer Bedeutung. Wenn zum Beispiel A Berechnungen über einem Datenbestand vornimmt und B gleichzeitig Teile dieses Bestands aktualisiert, dann sind für A die Veränderungen, die durch B durchgeführt worden sind, durchaus von Interesse. In Abhängigkeit der Update-Optionen *write-update* oder *write-invalidate* findet ein unterschiedliches Propagieren statt.

  Bei *write-update* werden nach der Veränderung eines Datums alle Prozesse, die auch eine lokale Kopie des Datums besitzen, benachrichtigt. Dabei wird ihnen eine aktuelle Kopie zugeschickt. Vollzieht ein Prozeß lediglich einen lesenden Zugriff, kann dieser ohne Kommunikation durchgeführt werden. Diese Art der Update-Option ist auch als *multiple-reader-multiple-writer sharing* bekannt. Als Nachteil ist vor allem der ständige, große Netzwerkverkehr zu nennen.

  Die *write-invalidate* Update-Option, die auch als *multiple-reader-single-writer* bezeichnet wird, unterscheidet zwischen dem *read-only* und dem *read-write* Modus eines Datums. Wenn ein Prozeß schreiben will, so propagiert er seinen Wunsch an alle Prozesse, die eine lokale Kopie von dem Datum besitzen. Die lokalen Kopien werden daraufhin invalidiert. Erst wenn alle Prozesse ihre Invalidierung bestätigt haben, darf der Prozeß das Datum in den *read-write* Modus überführen[112] und verändern. Lesende Zugriffe auf Daten im *read-write* Modus werden unterbunden. Dem Nachteil, daß vor einem Wechsel in den *read-write* Modus alle Kopienbesitzer informiert werden müssen, steht der Vorteil gegenüber, daß nur die letzte Veränderung eines Datums im *read-write* Modus propagiert werden muß.

- *Granularität* (Granularität der gemeinsam genutzten Daten im DSM)
  Die Daten im DSM werden in einer bestimmten Granulatgröße verwaltet. Die Granulatgröße kann die Ausprägung einer Seite, einer Datenstruktur

[110] „... but it has reasonable semantics that are tractable to programmers." ([Coulouris et al. 94])

[111] Es sei noch angemerkt, daß dieses Modell ein schwaches Konsistenzmodell ist. Nur wenn die Synchronisationsobjekte (korrekt) eingesetzt werden, kann von starker Konsistenz gesprochen werden.

[112] Damit ist der Prozeß Besitzer des Datums.

(zum Beispiel ein Objekt) oder eines Attributs einnehmen. In Abhängigkeit des Verwendungszwecks und der Systemebene, auf der der DSM realisiert wird, bieten die Granulatgrößen unterschiedliche Leistungen. Zur genauen Abwägung der Granulatgröße bedarf es der sorgfältigen Untersuchung der Transferkosten zwischen Rechnerknoten, der Verwaltungskosten der Zugriffssperren und der verwendeten Transaktionslängen.

Während bei der Attributgranularität nur die jeweils benötigten Daten gesperrt und transferiert werden, entstehen höhere Kosten bei der Verwaltung der Sperren und höhere Kosten beim Transfer als bei Seitengranularität. Demgegenüber ist der Umgang mit Seiten dann ineffizient, wenn nur Bruchteile einer Seite bearbeitet werden. Ein weiteres Problem entsteht bei der gemeinsamen Bearbeitung von unterschiedlichen Daten, die sich auf einer Seite befinden: Eine Seite muß zwischen den Prozessen ständig ausgetauscht werden. Dieses Phänomen wird in der Literatur als *false sharing* bezeichnet. Mindestens zwei Prozesse nutzen gleichzeitig eine Seite, wobei die Prozesse paarweise unterschiedliche Abschnitte der Seite nutzen.

- *Thrashing* (zeitliches Verhältnis zwischen produktiver Arbeit und dem dazugehörigen Netzwerkverkehr)
  Das Wort *thrashing* wird in der Literatur mit *Seitenflattern* übersetzt. Kennzeichnend ist der wiederholte Transport einzelner Speicherseiten zwischen Hauptspeicher und Hintergrundspeicher. Die Ursachen sind vielfältiger Natur: Zum einen können mehrere Prozesse gleichzeitig auf dieselben Daten oder auf unterschiedliche Daten einer Seite zugreifen (*false sharing*). Zum anderen können unmittelbar zuvor aus dem lokalen Hauptspeicher verdrängte Seiten wieder benötigt werden. Obwohl das Phänomen des *thrashing* auf alle Granulatgrößen übertragbar ist, tritt es insbesondere bei der Seitengranularität auf.

5.6.3 Management

Zur Realisierung des Konzepts des Distributed Shared Memory bedarf es effizienter Managementkonzepte, die die Trennung der transienten Speicher eines verteilten Systems überwinden. Manager müssen die Objekte zwischen den Rechnerknoten austauschen und die Verwaltung der Besitzverhältnisse und der Replikate regeln.

Ein Objekt befindet sich immer im Besitz eines Prozesses und damit im Besitz der Anwendungen, die dem Prozeß zugeordnet sind. Wollen mehrere Prozesse auf genau ein Objekt nacheinander oder auch parallel zugreifen, so muß das Objekt zwischen den transienten Speichern ausgetauscht werden. Hierbei sind prinzipiell zwei Möglichkeiten zu differenzieren: Entweder bleibt ein Prozeß Besitzer

des Objekts, erlaubt dessen Replizierung und koordiniert die an dem Objekt vorgenommenen Veränderungen oder das Objekt wechselt seinen Besitzer. Beim Besitzerwechsel werden keine Replikate von Objekten erstellt; somit muß das Objekt seinen physikalischen Standort wechseln[113]. Nur in diesem Fall kann von der Mobilität der Objekte gesprochen werden, weil im Vergleich zur Objektreplizierung die Objekte tatsächlich zwischen den Speichern verschoben werden. Bei der Umsetzung eines Distributed Shared Memory werden meist beide Verfahren angewandt. Ein Objekt kann also seinen Besitzer und damit den physikalischen Ort wechseln und dupliziert werden. Sowohl die Verwaltung der Besitzverhältnisse als auch die Replikatverwaltung sind Aufgaben eines Managers.

Der Zugriff auf Objekte im Distributed Shared Memory erfolgt wie der Zugriff auf den lokalen transienten Speicher. Dem Programmierer verborgen wird geprüft, ob das Objekt sich im lokalen Speicher befindet, ob es einen aktuellen, gültigen Zustand besitzt und ob die benötigte Berechtigung für den lesenden oder schreibenden Zugriff vorliegt. Zur Prüfung dieses Sachverhalts wird sich eines Managers bedient, der - falls erforderlich - die Voraussetzungen für einen Zugriff schafft und das Objekt in den lokalen Speicher holt.

Zur Umsetzung der genannten Aufgaben des Managements des Distributed Shared Memory nennt [Coulouris et al. 94] drei Konzepte:

1. *Fixed distributed management, centralized manager*
   Der DSM wird durch eine kleine, feste Anzahl von Managern[114] organisiert. Jedes Objekt im DSM ist genau einem dieser Manager zugeordnet und wird von ihm verwaltet. Dabei ist entweder ein ausgezeichneter Manager selbst Besitzer aller Objekte oder jeder Manager verwaltet in einer Tabelle die Besitzverhältnisse der ihm zugeordneten Objekte.

   Bei der Verwaltung von Objektreplikaten können wiederum zwei Verfahren unterschieden werden: Entweder weiß ein Manager, welche Prozesse Kopien der ihm zugeordneten Objekte besitzen, oder der besitzende Prozeß verwaltet die Menge aller Kopienbesitzer. Obwohl im zweiten Fall die Managementaufgabe dezentralisiert zu sein scheint, hält immer nur genau ein Prozeß das Wissen über die Menge aller Kopienbesitzer. Somit enthält das Konzept des *fixed distributed management* immer einen *bottleneck*, der gleichbedeutend mit geringer Performanz ist. Dies wirkt sich insbesondere auf die Skalierbarkeit des verteilten Systems aus. Hinzu kommt eine schlechte Lastverteilung, weil nur wenige Rechnerknoten mit dem Management des DSM beauftragt sind.

[113] Ein Objekt befindet sich immer in dem physikalischen Speicher, der seinem besitzenden Prozeß zugeordnet ist.
[114] häufig nur ein zentraler

2. *Multicast-based distributed management, broadcast distributed manager*
   Bei dem Ansatz des *multicast-based distributed management* hat jeder Prozeß seinen eigenen Manager. Die Objekte sind keinem bestimmten Manager zugeordnet, so daß sich die Manager die Last der Koordination aufteilen können. Dies bedingt jedoch auch, daß bei einem Zugriff auf ein Objekt alle Prozesse, d.h. alle Manager, informiert werden müssen (*broadcasting*). Besondere Probleme dieses Ansatzes sind das ,,zeitgleiche Broadcasting" zweier Prozesse und die gleichbleibend hohe Netzwerk- und Rechnerknotenbelastung durch ständiges ,,Broadcasting" bzw. dadurch hervorgerufene Unterbrechungen des Ablaufs auf den Rechnerknoten.

3. *Dynamic distributed management, dynamic distributed manager*
   Der Ansatz des *dynamic distributed management* ist mit dem des *multicast-based distributed management* vergleichbar. Jeder Prozeß hat seinen eigenen Manager; die Objekte sind keinem bestimmten Manager zugeordnet. Durch einen effizienten Algorithmus können jedoch die Probleme des *multicast-based distributed management* vermieden werden. Jeder Manager hält für jedes Objekt einen Hinweis auf dessen Besitzer. Durch Verfolgung dieser Hinweise (Kette von Hinweisen) kann ein Besitzer ermittelt werden. Damit die Verfolgung der Hinweise effizient ist, müssen diese ständig aktualisiert werden:

   - Bei der Übertragung des Besitzes von A nach B aktualisiert A seinen Hinweis.
   - Muß A sein Replikat aufgrund einer Nachricht von B invalidieren, so aktualisiert A seinen Hinweis (auf B).
   - Wird der lesende Zugriffswunsch von Prozeß A durch B erfüllt, so aktualisiert A seinen Hinweis (auf B).
   - Erhält A eine Anfrage von B nach einem Objekt, das er nicht besitzt, aktualisiert A seinen Hinweis auf B.
   - Periodisch werden die aktuellen Besitzer propagiert. Damit wird die Länge aller Hinweisketten auf eins reduziert.

   Als besonderer Nachteil dieses Ansatzes ist die erforderliche Verwaltung der Hinweise für jedes Objekt auf jedem Rechnerknoten zu erwähnen.

Welches dieser Management-Verfahren in einem verteilten System zum Einsatz kommen sollte, ist von den speziellen Anforderungen einer Anwendungsdomäne abhängig. Bei der Vernachlässigung von Performanzgesichtspunkten kann prinzipiell jeder Ansatz verwendet werden. Ein Vorteil des *fixed distributed management* liegt darin, daß der Ansatz garantiert, den Besitzer einer Seite durch Versenden von maximal zwei Nachrichten aufzufinden. Das *dynamic distributed*

management hingegen kann zwischen einer und n-1 (n ist die Anzahl der Prozesse) Nachrichten benötigen. Damit ist das zeitliche Verhalten weniger präzise vorhersagbar. Andererseits gibt es viele Anwendungen, bei denen auf ein Objekt nur von wenigen - oft nur von zwei - Prozessen zugegriffen wird. In diesem Fall ist das *dynamic distributed management* vorzuziehen. Abschließend sei auf den hohen Speicherbedarf beim *dynamic distributed management* für die Verwaltung der Besitzer und beim *fixed distributed management* für die Verwaltung der Menge der Kopienbesitzer hingewiesen. Weitere Ausführungen können [Coulouris et al. 94] und [Mairandres 96] entnommen werden.

5.6.4 Der Persistent Distributed Shared Memory

> ,,It was observed that the locality transparency principle of persistence can be generalized in a natural way in a distributed environment. ... The manipulation of remote objects is both syntactically and semantically the same as if they were local objects. Distributed programming is no more difficult than conventional persistent programming." ([Wai 89])

In den letzten Unterabschnitten haben wir sowohl die Leistungsstärken des Konzepts des virtuellen Speichers (Transparenz der Persistenz) als auch die des Distributed Shared Memory (Transparenz der Mobilität) diskutiert. Beiden Abstraktionen ist gemein, ,,daß sie auf sehr ähnliche Weise sehr einfache Programmiermodelle ermöglichen" ([Mathiske 96b]):

- Das Konzept der Transparenz der Mobilität (auch Verteilungs- oder Ortstransparenz genannt) abstrahiert horizontal von der Lokalität von Objekten. Ein Programmierer braucht sich nicht darum zu kümmern, ob sich bestimmte Objekte auf dem jeweils lokalen oder einem entfernten Rechnerknoten befinden.

- In Analogie wird bei der Transparenz der Lebensdauer vertikal von der Lokalität von Objekten abstrahiert. Ein Programmierer braucht sich nicht darum zu kümmern, ob sich bestimmte Objekte im Primär- oder Sekundärspeicher befinden.

Ferner liegen beiden Konzepten sehr ähnliche Anforderungen zugrunde:

- Reduzierung der Komplexität der Implementation
- Höhere Wiederverwendbarkeit der Implementation
- Unabhängige Anpaßbarkeit und Optimierbarkeit der Anwendungen
- Vergleichbarkeit des Programmieraufwands mit Einzelplatz- bzw. zentralisierten Anwendungen, die keine Persistenz unterstützen[115]

[115] vgl. [Parrington 92]

Darüber hinaus stellt [Mathiske 96b] fest, daß es analoger Maßnahmen zur Unterstützung von Persistenz, sofern die Paradigmen zwischen Programmiersprache und Datenbanksystem unterschiedlich sind, und von Mobilität bedarf:

- Entwicklung externer Datenrepräsentationen
- Entwicklung von Algorithmen zur Umwandlung zwischen internen Objekten und deren externen Repräsentationen
- Einbindung der Algorithmen in übergeordnete Mechanismen (zum Beispiel Kommunikationsprimitive (u.a. RPC[116]))

Aus diesen Analogien resultiert eine Synergie, die durch die Zusammenführung des virtuellen Speichers und des Distributed Shared Memory erreichbar ist. Unseres Wissens nach existieren bis heute auf der Programmiersprachenebene - abgesehen von DPS-Algol, PANDA und Distributed Texas[117] - keine ernst zunehmenden Versuche, beide Konzepte in einem zu vereinen[118]. Vielmehr werden auf der einen Seite persistente und auf der anderen Seite mobilität-unterstützende Programmiersprachen entwickelt, denen es an Akzeptanz und Verbreitung mangelt. Dieser Trend wird noch dadurch verstärkt, daß meist nur das Dateisystem oder unbekannte und nicht ausgereifte Datenbanksysteme als persistente Speicher angebunden werden.

Das Konzept, das sich hinter der Zusammenführung des virtuellen Speichers und des Distributed Shared Memory unter ausschließlicher Nutzung von Datenbanksystemen als persistente Speicher verbirgt, wollen wir als *Persistent Distributed Shared Memory* (*PDSM*) bezeichnen.

> **Begriff 43 :** *Persistent Distributed Shared Memory (PDSM)*
> Der Persistent Distributed Shared Memory (persistenter verteilter gemeinsamer Speicher) besteht aus den Primärspeichern der beteiligten Rechnerknoten und mindestens einem Datenbanksystem. Er abstrahiert von diesen physikalisch vertikal und horizontal verteilten Speichern und stellt einen logisch gemeinsamen Speicher bereit, der sich wie ein gewöhnlicher transienter Speicher verhält. Auf Objekte im PDSM kann direkt und netzwerkweit eindeutig zugegriffen werden. Dadurch stellt der PDSM Transparenz bezüglich der Persistenz und der Mobilität der Objekte zur Verfügung.

Als technisches Pendant zum Konzept des Materialraums schlagen wir einen Persistent Distributed Shared Memory vor, der wie folgt charakterisiert ist:

---

[116] **r**emote **p**rocedure **c**all

[117] vgl. ([Haines 95])

[118] Bisher gibt es nur wenige Studien, wie mobilität-unterstützende und persistente Programmiersprachen kombiniert werden können ([Mathiske 96b]).

- *Struktur/Granularität*:
  Exemplar einer Materialklasse (Objekt)

- *Synchronisationsmodell*:
  fachliche Verfahren

- *Konsistenzmodell*:
  release consistency Modell

- *Update-Option*:
  write-invalidate

- *Thrashing*:
  Es kommt zum *thrashing*, wenn keine (fachlichen) Sperrverfahren angewandt werden oder der lokale Speicher unterdimensioniert ist[119].

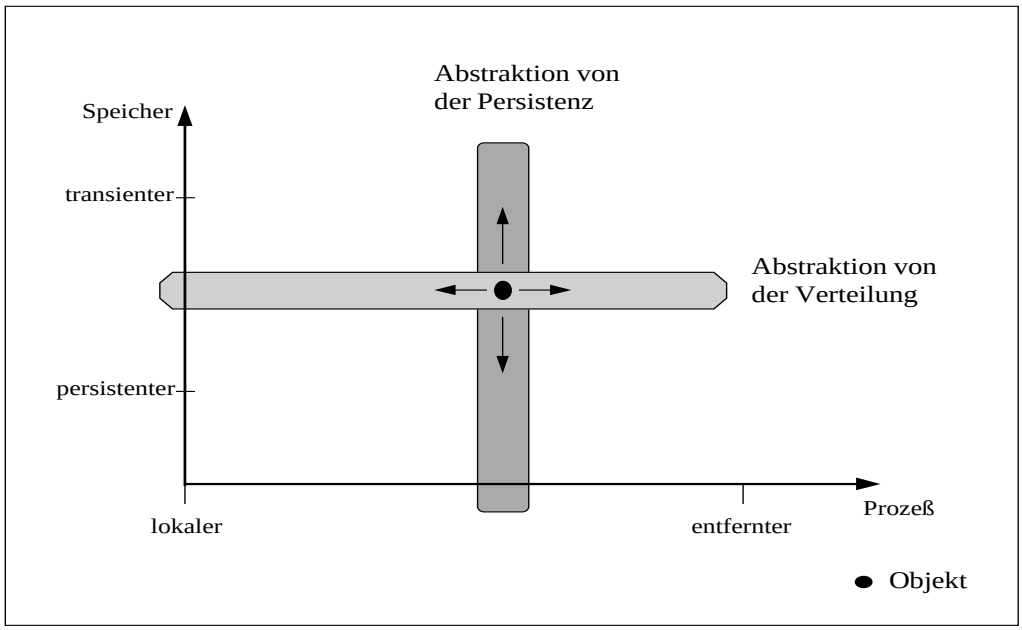


Abbildung 5.11: Abstraktion von der vertikalen und der horizontalen Speichertrennung

Abbildung 5.11 veranschaulicht die Abstraktion von der vertikalen und der horizontalen Speichertrennung. Ein Objekt kann sich an einem beliebigen Ort befinden und ist dennoch von überall in gleicher Weise zugreifbar.

[119] Nicht alle aktuell bearbeiteten Objekte können gleichzeitig im lokalen Speicher gehalten werden.

Für eine erste prototypische Umsetzung des Persistent Distributed Shared Memory schlagen wir als Management das *fixed distributed management* vor, weil es sich im Vergleich zu den anderen Managementansätzen leicht umsetzen läßt. Es sollte durch genau einen Manager realisiert werden, dem die Aufgabe zukommt, die Besitzverhältnisse und die Replikate zu verwalten. Ferner sollte nur genau ein Datenbanksystem angeschlossen werden[120], das darüber hinaus eine ODMG-93-konforme Schnittstelle aufweist. Somit kann die Funktionalität des PDSM unabhängig vom persistenten Speicher realisiert und Datenbanksysteme sowohl unterschiedlicher Hersteller als auch unterschiedlicher Paradigmen angeschlossen werden.

Alle Objekte, die sich im PDSM befinden, sollten in der Datenbank abgelegt sein. Objekte, die ein Anwendungsprozeß aktuell bearbeitet, liegen dann nur als Replikat im lokalen, transienten Speicher. Dieser übernimmt damit die Funktion eines Objektcaches.

Der Persistent Distributed Shared Memory kann prinzipiell durch beliebige Datenbankfunktionalität erweitert werden. Insbesondere sollte die Einführung einer Anfragesprache angedacht werden. Im nächsten Kapitel wird eine prototypische Umsetzung des PDSM vorgestellt, die sich jedoch auf eine Einzelplatzlösung beschränkt. Bis auf die Konzepte des Managers werden alle grundlegenden Elemente anhand einer Spezifikation diskutiert.

[120] „Bisher wurde noch keine allgemeine Antwort auf die Frage gefunden, wie mehrere persistente Objektspeicher, die aufgrund transparent entstandener entfernter Objektreferenzen in undurchschaubarer Weise voneinander abhängen, konsistent gehalten, bzw. wie Fehler und Ausfälle in solchen Systemen (transparent) überwunden werden können [Dearle et al. 91; Mira da Silva 95a; Atkinson, Morrison 95]." ([Mathiske 96b])

Kapitel 6

Zusammenführung von Materialraum und Persistent Distributed Shared Memory

Gegenstand dieses Kapitels ist das Framework *dinx*, das die in Kapitel 4 beschriebenen fachlich generellen Konzepte und damit insbesondere den Materialraum prototypisch und basierend auf einem Persistent Distributed Shared Memory (beschränkt auf eine Einzelplatzlösung) realisiert. Zunächst wird das Framework *dinx* in den Kontext der 3-Schicht-Architektur eingeordnet. Anschließend erfolgt die Beschreibung der (programmier-)technischen Realisierung der in Kapitel 4 erarbeiteten fachlich generellen Konzepte (Materialidentität, Materialverknüpfung, Materialkomposition und Materialzustandsgleichheit sowie das Konzept der Umgangsarten). Dabei identifizieren wir die Komponenten des Frameworks und legen ihre Schnittstelle fest. Nachdem wir auf diese Weise die statische Sicht auf das Framework formuliert haben, werden wir die dynamische Sicht anhand ausgewählter repräsentativer Szenarien darstellen und mit Hilfe von Interaktionsdiagrammen visualisieren. Abschließend werden wir im Zuge der Beschreibung der Parametrisierung auf die Initialisierung und die Terminierung des Materialraums eingehen.

6.1 Überblick

Bevor wir die Struktur und das Verhalten des Frameworks *dinx* detailliert beschreiben, wollen wir in diesem Abschnitt zunächst auf die Charakteristika von Frameworks kurz eingehen, um anschließend eine Einordnung des Frameworks *dinx* in den Kontext der von uns favorisierten Schichtenarchitektur von Informationssystemen vorzunehmen.

Begriff 44 : *Framework*
Ein Framework besteht aus einer „Menge kooperierender Klassen, welche die Elemente eines wiederverwendbaren Entwurfs für eine bestimmte Art von Software darstellen. Ein Framework bietet eine Architekturhilfe beim Aufteilen des Entwurfs in abstrakte Klassen und beim Definieren ihrer Zuständigkeiten und Interaktionen." ([Gamma et al. 95])

Eine entscheidende Rolle kommt der Parametrisierung eines Frameworks zu. Dabei sind prinzipiell drei unterschiedliche Herangehensweisen bei der Parametrisierung von Frameworks zu unterscheiden:

1. *Parametrisierung durch Vererbung*:
   Der Benutzer eines Frameworks spezialisiert ein Framework für eine bestimmte fachliche Anwendung, indem er konkrete anwendungsspezifische Unterklassen abstrakter Framework-Klassen bildet ([Gamma et al. 95]) und Exemplare dieser Klassen instantiiert. Da für die Unterklassenbildung, die Schnittstelle der zu konkretisierenden Methoden nach außen bekannt gemacht werden muß, wird bei dieser Art von Parametrisierung auch von einer *White-Box-Parametrisierung* gesprochen.

2. *Parametrisierung durch Objektinstantiierung*:
   Der Benutzer eines Frameworks instantiiert ein Exemplar einer Framework-Klasse und macht es dem Framework bekannt. Da er dabei kein Wissen über die Schnittstelle des übergebenen Objekts benötigt, wird bei dieser Art von Parametrisierung auch von einer *Black-Box-Parametrisierung* gesprochen.

3. *Parametrisierung durch Klasseninstantiierung*:
   Der Benutzer eines Frameworks spezialisiert ein Framework für eine bestimmte fachliche Anwendung, indem er eine generische Framework-Klasse mit einem Parameter instantiiert und auf diese Weise eine neue Klasse bildet, von der Exemplare erzeugt werden können. Bei dieser Art von Parametrisierung sprechen wir von einer *generischen Parametrisierung*.

Im Gegensatz zu Klassenbibliotheken nutzen nicht die Klassen und Objekte, mit denen das Framework parametrisiert worden ist, das Framework, sondern die Exemplare der Parameterklassen werden in der Umkehrung durch das Framework aufgerufen. Die Nutzung eines Frameworks ist damit durch einen invertierten Kontrollfluß gekennzeichnet, der in der Literatur[1] auch als „Hollywood-Prinzip"[2] bezeichnet wird. Auf die Parametrisierung des Frameworks *dinx* gehen wir im Anschluß an die Darstellung des Frameworks im Unterabschnitt 6.6.2 näher ein.

[1] zum Beispiel [Gamma et al. 95]
[2] Don't call us, we'll call you!

Wie wir bereits in den Abschnitten 2.4 und 3.5 motiviert haben, favorisieren wir eine Partitionierung der Architektur eines Informationssystems in Schichten, deren Schnittstellen höheren Schichten jeweils eine bestimmte Abstraktion und damit eine Reihe ausgewählter Dienste bereitstellen. Die in Abschnitt 2.4 vorgeschlagene 3-Schicht-Architektur besteht aus drei aufeinander aufbauenden Abstraktionsebenen, von denen das Framework *dinx* neben der Adaptionsschicht auch die Abstraktionschicht realisiert.

- Die ressourcenabhängige und anwendungsunabhängige *Abstraktionsschicht* abstrahiert von speziellen persistenten Speichern (zum Beispiel Datenbanksystemen und Dateisystemen) und stellt damit eine plattformunabhängige Schnittstelle zur Bereitstellung von Persistenz zur Verfügung. In Abschnitt 5.3.1 haben wir bereits motiviert, daß die Schnittstelle der Abstraktionsschicht dem ODMG-93-Standard genügen soll. Deshalb ist es die Aufgabe der Abstraktionsschicht, einen konkreten persistenten Speicher beliebigen Paradigmas mit einer ODMG-93-konformen Schnittstelle zu versehen und die Abbildung der Konzepte des ODMG-93-Standards auf die speziellen Konzepte des konkreten persistenten Speichers vorzunehmen.

- Die anwendungsabhängige und ressourcenabhängige *Adaptionsschicht* abstrahiert von der vertikalen und der horizontalen Speichertrennung und stellt damit nach außen eine ressourcenunabhängige Schnittstelle zur Verfügung. Die Adaptionsschicht entkoppelt die fachlichen Anteile eines Informationssystems von den technischen Anteilen und bildet gleichzeitig einen Adapter (*gateway*) zwischen der fachlichen und der technischen Welt. Wie bereits in Kapitel 3 und Kapitel 4 motiviert worden ist, sollte diese Adaptionsschicht, ausschließlich fachlich generelle Konzepte (zum Beispiel einen Materialraum) bereitstellen.

- Die anwendungsabhängige und ressourcenunabhängige *Anwendungsschicht* nutzt die fachlich generellen Basiskonzepte der Adaptionsschicht, um losgelöst von den technischen Details die (programmier-)technische Repräsentation der fachlich spezifischen Welt zu realisieren. Damit besteht die Aufgabe der Anwendungsschicht u.a. darin, den Materialraum beispielsweise durch die Modellierung eines fachlichen Lokalitätsbegriffs mit Hilfe von Arbeitsumgebungen und Archiven zu partitionieren, um auf diese Weise die fachliche Transparenz ([Gryczan 95]) zu modellieren.

Diese Schichtenarchitektur wird zusätzlich um eine Komponente zur Bereitstellung eines Meta-Objekt-Protokolls[3] erweitert (vgl. Abbildung 6.1). Neben der Abstraktions- und der Adaptionsschicht wird auch das Meta-Objekt-Protokoll durch das in dieser Arbeit vorgestellte Framework *dinx* realisiert. Während die

[3] vgl. Abschnitt 6.4

Adaptionsschicht und das Meta-Objekt-Protokoll durch fachlich spezifische Details zu parametrisieren sind, ist der Abstraktionsschicht ein konkreter persistenter Speicher zuzuordnen.

Durch die fachlich generelle Schnittstelle der Adaptionsschicht kann der Softwareentwickler bei der Umsetzung fachlicher Konzepte von Designentscheidungen bezüglich der Datenbanksystemanbindung abstrahieren und sich auf die adäquate Umsetzung der identifizierten fachlich spezifischen Details der zu entwickelnden Anwendung in der Anwendungsschicht konzentrieren. Somit wird die Entwicklung fachlicher Anwendungen, die dem Leitbild kooperativer qualifizierter menschlicher Tätigkeit entsprechen, erheblich vereinfacht. Zudem ermöglicht die lose Kopplung der einzelnen Schichten, zwischen denen keine zyklischen Beziehungen bestehen, einen unkomplizierten Austausch beispielsweise der Abstraktionsschicht und damit des persistenten Speichers. Alle persistenten Speicher, die bereits eine ODMG-93-konforme Schnittstelle aufweisen, stellen vollwertige Abstraktionsschichten dar und können deshalb direkt an die Adaptionsschicht gebunden werden[4].

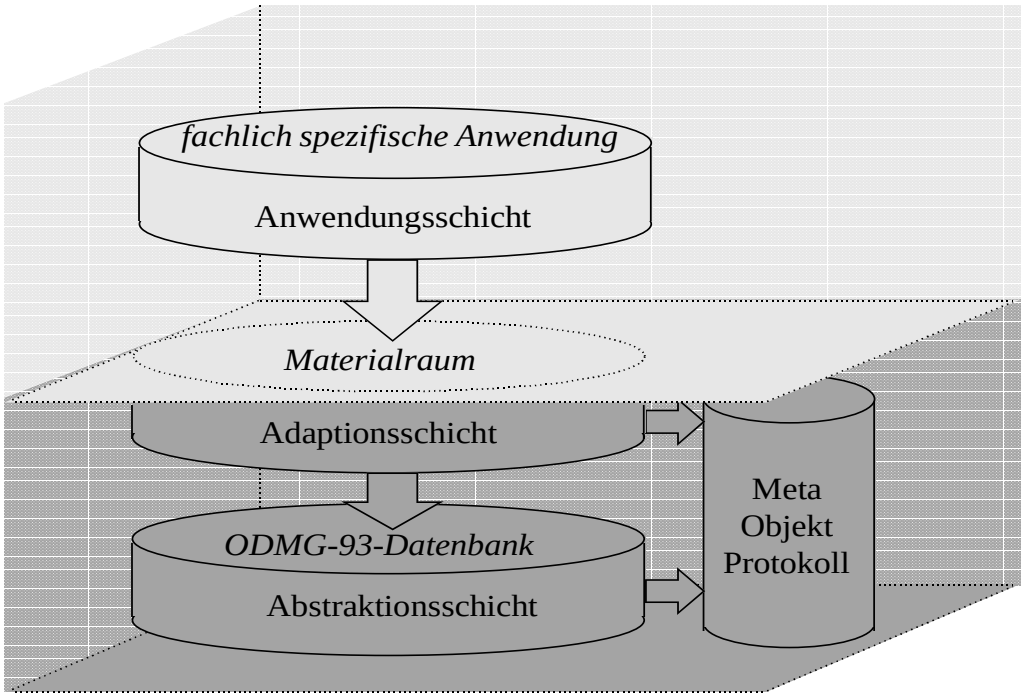


Abbildung 6.1: Überblick über die Schichtenarchitektur eines Informationsystems

Um bei der in den folgenden Kapiteln dargestellten (programmier-)technischen Umsetzung fachlicher Konzepte eine klarere begriffliche Trennung zwischen fach-

[4]vgl. hierzu die Ausführungen in [Hohenstein 96]

lichen Materialien auf der einen Seite und ihrer (programmier-)technischen Umsetzung auf der anderen Seite zu erreichen, wollen wir im folgenden den Begriff des Materialobjekts (Exemplar einer Materialklasse) für die (programmier-)technische Umsetzung fachlicher Materialien verwenden. Während komplexe Materialien durch mehrere Materialobjekte (mindestens ein Kontext- und ein Komponentenmaterialobjekt) realisiert werden, sind nicht-komplexe Materialien als einzelne Materialobjekte realisiert.

6.2 Der Materialraum als fachlich generelle Schnittstelle der Adaptionsschicht

In diesem Abschnitt wollen wir uns mit der (programmier-)technischen Umsetzung der in Kapitel 4 beschriebenen fachlich generellen Konzepte beschäftigen und insbesondere die Realisierung des Materialraums im Kontext des Frameworks *dinx* beleuchten. Dabei wird der Materialraum durch einen Persistent Distributed Shared Memory[5], der auf einem beliebigen ODMG-93-Datenbanksystem basiert, realisiert. Die (programmier-)technisch umgesetzten fachlich generellen Konzepte sind für die Implementation fachlich spezifischer Anwendungen nutzbar. In diesem Rahmen kann das Framework *dinx* als eine Erweiterung der Programmiersprache C++ um Objektpersistenz verstanden werden.

6.2.1 Der Materialraum

Der Materialraum dient zur Verwahrung der Materialobjekte in einem weder vertikal noch horizontal partitionierten Speicher. Ein Exemplar der Klasse *materialraum* kennt die Dienste, die durch die Klassen und Objekte der Adaptionsschicht bereitgestellt werden und delegiert die von außen[6] gestellten Anfragen an die zuständigen technischen Komponenten der Adaptionsschicht. Damit schirmt die Klasse *materialraum* die fachlichen Klienten des Frameworks von den technischen Komponenten ab und fördert auf diese Weise die in Kapitel 3 geforderte lose Kopplung zwischen den fachlichen Klienten auf der einen Seite und den technischen Komponenten auf der anderen. Der Materialraum modelliert damit eine Fassade (vgl. das Entwurfsmuster Fassade[7] ([Gamma et al. 95])) des Frameworks, da ein Exemplar der Klasse *materialraum* über seine bereitgestellte Schnittstelle die implizite Benutzung aller Dienste des komplexen Frameworks ermöglicht.

[5] beschränkt auf eine Einzelplatzlösung

[6] aus der Anwendungsschicht

[7] Zweck des Fassadenmusters: „Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Verwendung des Subsystems vereinfacht." ([Gamma et al. 95])

In den meisten objektorientierten Datenbanksystemen[8] besteht die Möglichkeit, Objekten benutzerdefinierte Namen (Schlüssel) zuzuordnen, anhand derer später auf die Objekte zugegriffen werden kann. Dabei läßt sich die Art der Vergabe anhand der im folgenden aufgezeigten Dimensionen beschreiben:

1. *Anzahl der Namen*
   Zuordnung eines oder mehrerer Namen

2. *Gültigkeitsbereich des Namens*
   Systemweite oder datenbankweite Eindeutigkeit

3. *Zeitpunkt der Vergabe*
   Jederzeit oder ausschließlich zum Zeitpunkt der Objekterzeugung

4. *Änderbarkeit der Zuordnung*
   Zuordnung eines Namens zu einem Objekt ist fix oder kann verändert werden.

Neben dem Zugriff über zuvor vergebene Namen kann der Zugriff auf ein Objekt in einer objektorientierten Datenbank auch über eine entsprechend formulierte Anfrage erfolgen (OQL). Auf der Basis von Datenbankanfragen kann prinzipiell jedes Objekt der Datenbank selektiert und als Einstiegspunkt zur weiteren Objektnavigation genutzt werden. „Idealerweise erfolgt .. in einem OODBMS der Einstieg in die Datenbank sowohl über Anfragen als auch über persistente Namen." ([Hohenstein et al. 96])

Da der Materialraum ähnlich wie objektorientierte Datenbanken zur Versorgung anderer Komponenten mit (Material-)Objekten dient, stellt der Materialraum analog zu der Basisfunktionalität objektorientierter Datenbanksysteme ein rudimentäres Konzept fachlicher Materialnamen zur Verfügung, mit deren Hilfe Referenzen auf Materialobjekte, die sich im Materialraum befinden, hergestellt werden können (*lookup_material()*). Jedem Materialobjekt kann genau ein fachlicher Name zugeordnet werden (*set_material_name()*), der innerhalb des gesamten Materialraums eindeutig ist und jederzeit gesetzt (*set_material_name()*), verändert (*rename_material()*) bzw. gelöscht (*delete_material_name()*) werden kann. Die fachlich spezifische Semantik der Materialnamen findet innerhalb des Frameworks keine Verwendung, so daß die Realisierung der Namen auf einer fachlich generellen Ebene nicht im Widerspruch zu der in Kapitel 4 getroffenen Feststellung, daß Materialnamen fachlich spezifischer Natur sind, steht. Der Umgang mit den Materialnamen erfolgt ausschließlich außerhalb des Frameworks, in einem durch die Anwendungsschicht modellierten fachlich spezifischen Kontext. Darüber hinaus sollte der Materialraum einen Anfragemechanismus bereitstellen, der bestimmte

[8] auch ODMG-93-Standard (vgl. Abschnitt 5.2.3)

Materialobjekte im Materialraum selektierbar macht und als Ergebnis einer Anfrage eine Sammlung von *Materialraumreferenzen*[9] auf die Materialobjekte, die einer Anfrage genügen, zurückgibt[10].

class materialraum

```
public:

void        open                    (abstrakte_fabrik*_my_fabrik,
                                     schema*          _my_schema,
                                     Database*        _my_Database);

void        close                   ();
-------------------------------------------------------------------------------
void        set_material_name       (const mr_ref_any& _material,
                                     int               _name);

void        rename_material         (int _oldname,
                                     int _newname);

mr_ref_any  lookup_material         (int _name);

void        delete_material_name    (const mr_ref_any& _material);
```

Abbildung 6.2: Schnittstelle der Klasse *materialraum*

Der Materialraum wird durch den Aufruf der Operation

- *materialraum::open(abstrakte_fabrik*, schema*, Database*)*

geöffnet. Dabei wird einem Exemplar der Klasse *materialraum*[11] eine ODMG-93-Datenbank (*Database**), eine Fabrik[12] (*abstrakte_fabrik**), die zur Erzeugung von Materialobjekten dient, und ein Klassenschema[13] (*schema**), welches das Meta-Objekt-Wissen bereitstellt, bekannt gemacht. Dadurch wird die Adaptionsschicht (Materialraum) mit einer Abstraktionschicht (ODMG-93-Datenbank) und einem Meta-Objekt-Protokoll (Schema) gekoppelt.

[9] vgl. hierzu Unterabschnitt 6.2.3

[10] Das Konzept fachlicher Namen haben wir im Rahmen der prototypischen Realisierung des Frameworks *dinx* rudimentär auf der Basis von ganzzahligen *int*-Werten umgesetzt. Auf die Umsetzung eines Anfragemechanismus ist aus Komplexitätsgründen verzichtet worden.

[11] vgl. Abbildung 6.2

[12] vgl. hierzu den Abschnitt 6.4

[13] vgl. hierzu den Abschnitt 6.4

Die Kopplung wird erst durch das Schließen des Materialraums durch den Aufruf der Operation

- *materialraum::close()*

wieder aufgehoben.

In den folgenden Unterabschnitten gehen wir auf die (programmier-)technische Umsetzung der in Kapitel 4 diskutierten fachlich generellen Konzepte ein. Dabei werden wir die einzelnen Komponenten des Frameworks identifizieren und ihre Schnittstelle festlegen.

6.2.2 Materialidentität

Da das Identitätskonzept von C++, das die Identität von Objekten über ihre Lokalität im Hauptspeicher definiert, vor allem im Hinblick auf persistente Materialobjekte als unzureichend zu bewerten ist, ist die Einführung eines (programmier-)technischen Konzepts zur Modellierung von Materialidentitäten unumgänglich. Jedem Materialobjekt wird daher zum Zeitpunkt seiner Erzeugung automatisch und für den Programmierer transparent ein sogenannter *Materialidentifikator (MID)* eineindeutig zugeordnet, der über die gesamte Lebensdauer des Materialobjekts nicht veränderbar ist.

> **Begriff 45 :** *Materialidentität*
> technische Interpretation:
> Jedem Materialobjekt ist ein Materialidentifikator eineindeutig zugeordnet. Materialidentifikatoren werden den Materialobjekten zum Zeitpunkt ihrer Erzeugung von einer Instanz (Materialraumidentitor) initial und eineindeutig zugeordnet und sind anschließend nicht modifizierbar.

In Anlehnung an [Kirschke 95] werden die folgenden Bedingungen an eine adäquate technische Identitätsrepräsentation und damit auch an die Materialidentifikatoren gestellt:

1. *Eindeutigkeit*
2. *Unabhängigkeit vom aktuellen Zustand*
3. *Unabhängigkeit vom aktuellen (Aufenthalts-)Ort*
4. *Unabhängigkeit von der Zeit*
5. *Unabhängigkeit vom Typ*

In Anlehnung an [Kirschke 95] sind die folgenden Arten von Materialidentifikatoren zu unterscheiden:

1. *Physikalische Identifikatoren*:
   Die Realisierung des Materialidentifikators auf der Basis von physikalischen Identifikatoren (zum Beispiel Hauptspeicheradressen) ist einfach auf konventionellen Plattformen zu realisieren. Physikalische Identifikatoren sind nur dann zeit- und ortsunabhängig, solange das dazugehörige Materialobjekt nicht aufgrund einer Speicherbedarfsänderung verschoben werden muß. Wenn Materialobjekte im Speicherbereich oder zwischen unterschiedlichen Speichern bewegt oder gelöscht werden, dann müssen alle Referenzen auf sie aktualisiert werden, weil die zugehörigen Verweise nicht länger gültig sind. Da es in persistenz-unterstützenden Systemen ständig zu Objektverschiebungen kommt, ist die Realisierung von Materialidentifikatoren auf der Basis physikalischer Identifikatoren nicht empfehlenswert.

2. *Strukturierte Identifikatoren*:
   Strukturierte Identifikatoren enthalten Informationen über den Typ und den Zustand des zugeordneten Objekts (zum Beispiel *AUTO_BLAU* oder *BUCH_BIBEL*). Damit sind sie zwar ortsunabhängig, aber es besteht weder zeitliche Unabhängigkeit noch eine Unabhängigkeit vom aktuellen Zustand bzw. vom Typ des zugeordneten Objekts. Zudem ist die Sicherstellung der Eindeutigkeit schwierig, weil es nicht ausgeschlossen ist, daß es mehrere Materialobjekte mit derselben Ausprägung einer bestimmten Eigenschaft, die Teil des Identifikators ist, gibt.

3. *Benutzerdefinierte Identifikatoren*:
   Benutzerdefinierte Identifikatoren werden explizit durch den Benutzer eines Materialobjekts vergeben (zum Beispiel *Anlage zum Antrag Hallmackenreuther*) und sind damit zeit- und ortsunabhängig modellierbar. Der Einsatz von benutzerdefinierten Identifikatoren ist vor allem deshalb problematisch, weil der Benutzer selbst die geforderte Eindeutigkeit der Identifikatoren sicherstellen muß. Dies ist insbesondere im Hinblick auf komplexe Informationssysteme schwierig, da hier viele Benutzer sich gegenseitig abstimmen müssen und zudem keiner dieser Benutzer einen umfassenden Überblick über alle bereits vergebenen Identifikatoren haben kann. Benutzerdefinierte Identifikatoren sind gleichzusetzen mit fachlichen Materialnamen, die aufgrund ihrer fachlich spezifischen Natur a priori für die Repräsentation von Materialidentitäten innerhalb eines technischen Kontexts ungeeignet sind. Es käme zur Korrelation der fachlichen und der technischen Welt, die im Hinblick auf die geforderte strikte Trennung beider Welten nicht hingenommen werden kann (vgl. Unterabschnitt 4.1.2).

4. *Tupel-Surrogat*:
   Tupel-Bezeichner oder Tupel-Surrogate stammen aus dem Bereich der relationalen Datenbanken und dienen dort zur Referenzierung von Tupeln. Die Repräsentation von Identität mittels Schlüsseln ist nicht ortsunabhängig, weil Schlüssel nur innerhalb einer Relation eindeutig sind. Nicht zuletzt deshalb ist ihr Einsatz innerhalb eines objektorientierten Kontexts nicht zweckmäßig (vgl. [Kirschke 95]).

5. *Surrogat*:
   Surrogate[14] sind systemgenerierte logische Identifikatoren[15], die unabhängig vom Ort, von der Zeit, vom Typ und vom Zustand sind. Neben einer Instanz zur Generierung von Surrogaten und zur eineindeutigen Vergabe dieser ist eine weitere Komponente erforderlich, die zur Abbildung der logischen Surrogate auf physische Speicheradressen dient.

Da Surrogate völlig unabhängig vom Ort, von der Zeit sowie dem Zustand und dem Typ des zugehörigen Materialobjekts sind, eignen sie sich sehr gut als Identitätsrepräsentationen für persistente Materialobjekte ([Kirschke 95]). Aus diesem Grund wollen wir die Materialidentifikatoren (*MID*) auf der Basis von Surrogaten realisieren.

Jedem Materialtyp bzw. jeder Materialklasse ist ein sogenannter Materialtypidentifikator (*CID*[16] [17]) eineindeutig zugeordnet[18]. Damit ist es möglich, einem Materialobjekt zum Zeitpunkt seiner Erzeugung den Materialtypidentifikator (*CID*) der Klasse, von der es Exemplar ist, zuzuordnen. Der Materialtypidentifikator (*CID*) erfährt wie der Materialidentifikator (*MID*) im Vergleich zu anderen Attributen eine besondere[19] Behandlung.

Beide ausgezeichneten Materialeigenschaften (Materialtypidentifikator (*CID*) und Materialidentifikator (*MID*)) modellieren wir in einer abstrakten Basisklasse *abstraktes_material*[20], von der alle Materialklassen der fachlichen Welt Unterklassen sind. Da die bereitgestellten Methoden (*set_CID()*, *get_CID()*, *set_MID()* und *get_MID()*) in den Unterklassen der Klasse *abstraktes_material* nicht konkretisiert werden müssen, gelingt es, daß jedes Materialobjekt ohne zusätzlichen Implementationsaufwand Wissen über seine Materialidentität und seinen Materialtyp hat

[14] Surrogate kommen zum Beispiel in den Datenbanksystemen GemStone und POSTGRES zum Einsatz.

[15] zum Beispiel auf der Basis von ganzzahligen Werten

[16] **class id**entifier

[17] Im Framework *dinx* werden die Begriffe Typ und Klasse synonym verwendet.

[18] Diese Zuordnung von Materialtypidentifikatoren (*CID*) zu Typnamen/Klassennamen erfolgt durch das Meta-Objekt-Protokoll (vgl. Abschnitt 6.4).

[19] vgl. hierzu die Ausführungen zur Materialzustandsgleichheit und Materialkopie (Unterabschnitt 6.2.5).

[20] vgl. Abbildung 6.3

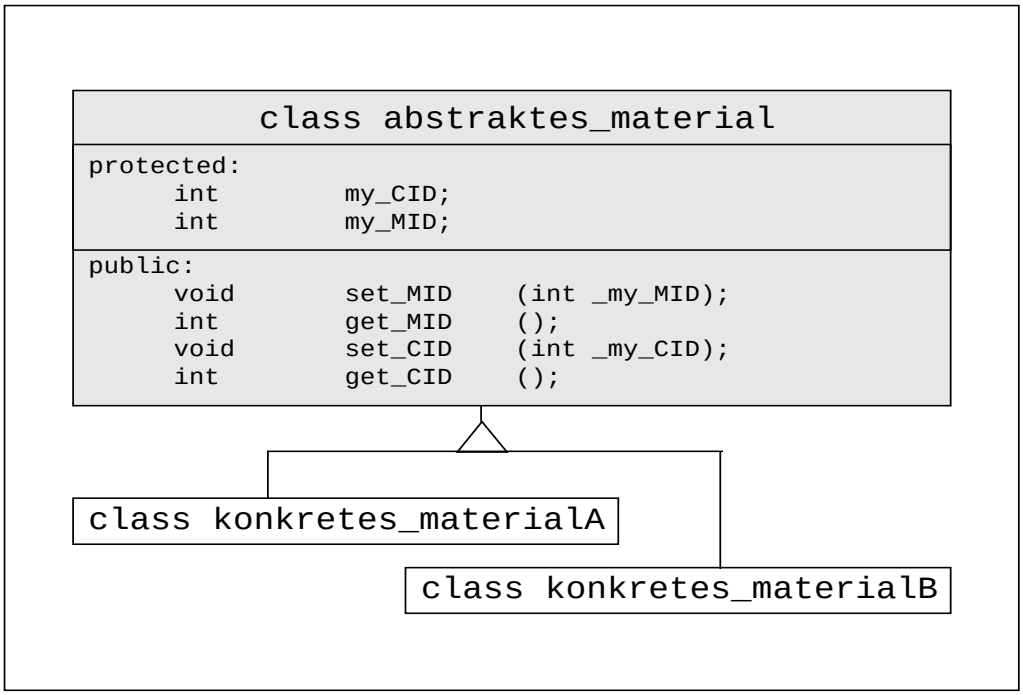


Abbildung 6.3: Schnittstelle der Klasse *abstraktes_material*

und nach diesen befragt werden kann (*get_CID()* und *get_MID()*).

Wie wir gesehen haben, werden für die Vergabe von und den Umgang mit Surrogaten zwei zusätzliche Komponenten benötigt. Zu diesem Zweck führen wir einen *Materialraumidentitor*, der die Vergabe der Materialidentifikatoren regelt, und einen *Materialraumverwalter*[21], der die Abbildung der Materialidentifikatoren (*MID*) auf physische Adressen ermöglicht, ein.

Um die Eindeutigkeit der Materialidentifikatoren (*MID*) sicherzustellen, erfolgt die Zuordnung von Materialidentifikatoren wie die der Materialtypidentifikatoren während der Erzeugung eines Materialobjekts. Ein Exemplar der Klasse *materialraumidentitor* verwaltet einen *MID-Zähler* (*my_MID_counter*), der hochgezählt wird[22], wenn ein neues Materialobjekt erzeugt wird (*get_next_MID()*). Auf diese Weise kann sichergestellt werden, daß keine MID mehr als einmal vergeben wird. Daneben erhält der Materialraumidentitor die Aufgabe, die Vergabe und Koordination fachlicher Materialnamen bereitzustellen.

[21] vgl. Unterabschnitt 6.3.2

[22] Da der MID-Zähler in einem verteilten System global bekannt ist, bildet der Zugriff und die Inkrementierung des MID-Zählers einen kritischen Abschnitt, der in einem verteilten System als solcher zu modellieren ist.

6.2.3 Materialverknüpfung und Materialadressierbarkeit

Die im vorhergehenden Unterabschnitt eingeführten Materialidentifikatoren sind erforderlich, um Referenzen zwischen persistenten Materialobjekten etablieren zu können. Wenn diese Identitätsrepräsentationen fehlen, dann ist es nicht möglich, Referenzen zwischen persistenten Materialobjekten zu bilden. Diese trivial wirkende Feststellung hat für diese Arbeit eine entscheidende Bedeutung: Die Modellierung von Materialverknüpfungen und die Adressierung von Materialobjekten im Materialraum ist nicht auf der Basis von C++-Referenzen realisierbar, weil C++-Referenzen Bezüge zwischen Objekten nicht über Surrogate (zum Beispiel Materialidentifikatoren) herstellen, sondern Verweise lediglich auf der Basis physikalischer Identifikatoren (transiente Hauptspeicheradressen) modellieren. Damit sind C++-Referenzen ausschließlich im transienten Kontext, solange die referenzierten Materialobjekte im transienten Speicher nicht verschoben werden, einsetzbar.

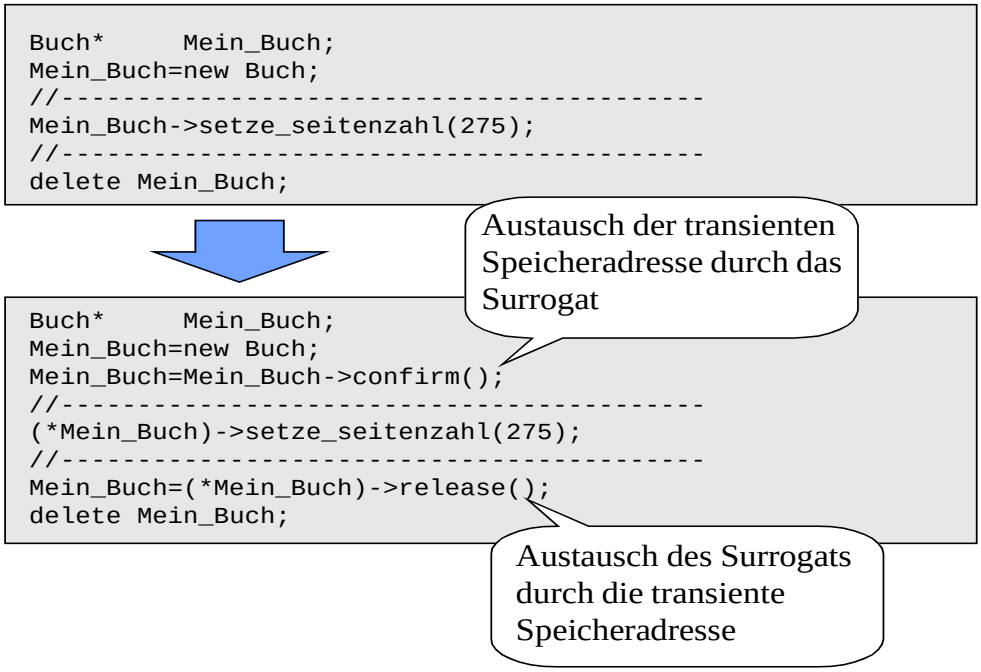


Abbildung 6.4: Verfälschung der C++-Syntax beim Austausch der von den C++-Referenzen verwalteten physikalischen Speicheradressen durch logische Identifikatoren

Die naheliegende Idee, die in C++-Referenzen repräsentierten physikalischen Hauptspeicheradressen durch Surrogate zu überschreiben, erweist sich vor allem im Hinblick auf den in C++ realisierten Mechanismus zur Typkonvertierung als schwierig. Bei einer Typkonvertierung einer C++-Referenz, die auf ein Exemplar einer virtuellen Klasse verweist, kommt es zu einer Veränderung der von ihr

referenzierten Hauptspeicheradresse. Nach der Adreßverschiebung zeigt die C++-Referenz auf den virtuell geerbten Bestandteil der Objektrepräsentation im transienten Speicher (vgl. hierzu den Unterabschnitt 6.4.1). Zudem kann der Dereferenzierungsoperator -> nicht verwendet werden, da er ausschließlich auf der Basis physikalischer Hauptspeicheradressen einsetzbar ist[23]. Der Versuch diese Probleme durch Überladen der Operatoren *new*, *delete*, * und -> in jeder Materialklasse zu umgehen, führt durch eine erforderliche doppelte Dereferenzierung[24] zu einer erheblichen Verfälschung der C++-Syntax, die im Hinblick auf eine leichte Erlernbarkeit des Umgangs mit dem in dieser Arbeit zu entwickelnden Framework nicht hingenommen werden kann (vgl. hierzu die Abbildung 6.4).

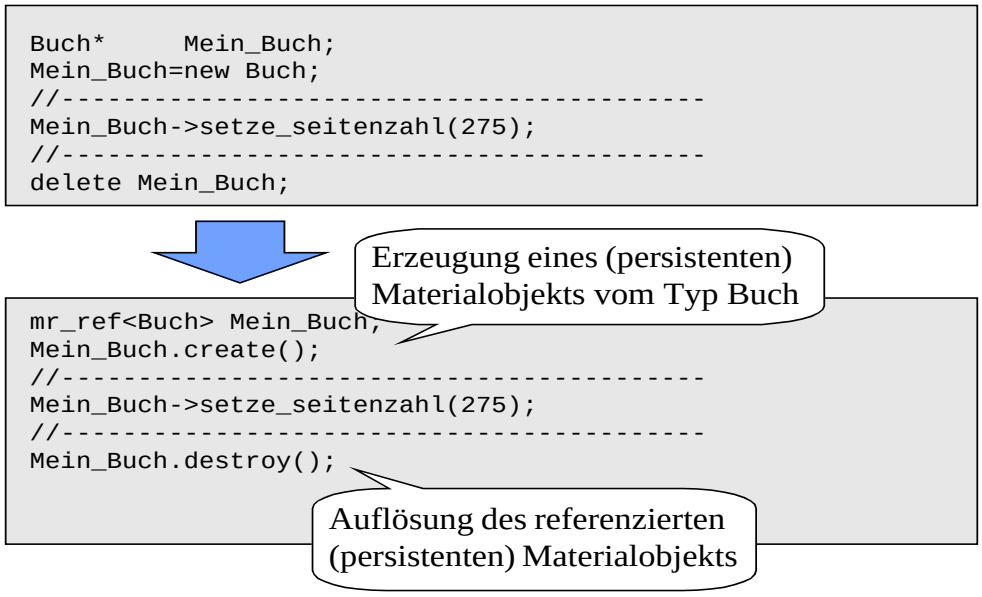


Abbildung 6.5: Umgang mit dem generischen Referenzmechanismus auf der Basis von Materialidentifikatoren

Aus diesen Gründen haben wir uns für einen kompletten Austausch des C++-Referenzmechanismus durch einen generischen Referenzmechanismus auf der Basis von Materialidentifikatoren entschieden (vgl. hierzu die Abbildung 6.5). Dadurch wird es möglich, daß der Benutzer des Frameworks im wesentlichen die

[23]Diese Einschränkung gilt nur für Objekte, die virtuelle Methoden enthalten. Durch den Vorgang des *late binding* erfolgt ein Zugriff auf die transiente Speicherrepräsentation, weil diese sogenannte *versteckte Zeiger* (*hidden pointers*) enthält, die auf die Tabelle der dynamisch gebundenen Methoden (*dispatch table* oder *vtable*) zeigen (vgl. hierzu den Unterabschnitt 6.4.1).

[24]Es muß zunächst die C++-Referenz mit Hilfe des *-Operators dereferenziert werden, um anschließend den überladenen ->-Operator des referenzierten Objekts aufrufen zu können. Da dieser ->-Operator in der üblichen C++-Semantik auch zur Dereferenzierung dient, sprechen wir von einer doppelten Dereferenzierung.

alte C++-Syntax mit erweiterter Semantik nutzen kann, wobei ihm die semantische Erweiterung zum größten Teil verborgen (transparent) bleibt. Lediglich die Deklaration von Referenzen sowie die Erzeugung und Auflösung von Materialobjekten unterscheidet sich von der etablierten C++-Syntax.

Der im Framework *dinx* realisierte generische Referenzmechanismus, den wir im folgenden als Materialraumreferenz-Mechanismus bezeichnen wollen, basiert auf dem in [Gamma et al. 95] beschriebenen Entwurfsmuster Proxy[25], das zur Realisierung von *indirekten Referenzen* (*smart pointer*) herangezogen werden kann. Im Gegensatz zu einer direkten Referenz stellt eine indirekte Referenz (*smart pointer*) nur einen mittelbaren eindeutigen Bezug zu einem Exemplar einer Materialklasse her. Die Dereferenzierung einer indirekten Referenz wird dabei in mehreren Schritten vollzogen, so daß zusätzliche Vorgänge (zum Beispiel der Transfer eines Objekts aus dem persistenten in den transienten Speicher[26]) während der Dereferenzierung angestoßen werden können. Eine indirekte Referenz enthält den Materialidentifikator (*MID*) des referenzierten Materialobjekts, mit dessen Hilfe eine zusätzliche Instanz einen eindeutigen Bezug auf die physische Repräsentation des Materialobjekts herstellen kann. Damit kann eine indirekte Referenz nur in Abhängigkeit von dieser zusätzlichen Instanz genutzt werden.

> **Begriff 46 :** *Materialraumreferenz*
> Eine Materialraumreferenz dient als Verweis auf ein Materialobjekt in einem Materialraum. Eine Materialraumreferenz ist eine indirekte Referenz (*smart pointer*), die eine eindeutige, mittelbare Bezugnahme auf ein Materialobjekt über dessen Materialidentifikator (MID) ermöglicht.

In Anlehnung an [Gamma et al. 95] zeichnet sich eine Materialraumreferenz durch die folgenden Charakteristika aus:

1. Die Materialraumreferenz verwaltet eine Referenz auf ein bestimmtes Materialobjekt auf der Basis eines Materialidentifikators.

2. Die Materialraumreferenz benutzt die Schnittstelle des referenzierten Materialobjekts.

3. Die Materialraumreferenz macht die komplette Schnittstelle des referenzierten Materialobjekts von außen durch andere Objekte zugreifbar. Dieses wird durch die Operatoren * und -> der (getypten) Materialraumreferenz

[25] Zweck des Proxymusters: „Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts." ([Gamma et al. 95])

[26] *Smart pointer* werden typischerweise für den Umgang mit persistenten Objekten eingesetzt und sorgen implizit für das Laden persistenter Objekte in den Hauptspeicher, wenn diese sich nicht bereits dort befinden ([Gamma et al. 95]).

möglich, die das referenzierte Materialobjekt dereferenzieren bzw. eine bestimmte Operation des referenzierten Materialobjekts aufrufen.

4. Die Materialraumreferenz kontrolliert den Zugriff auf das referenzierte Materialobjekt. So wird beispielsweise durch die Materialraumreferenz der Transfer des Materialobjekts aus dem persistenten in den transienten Speicher angestoßen, wenn es sich dort nicht bereits befindet.

Da mit Hilfe der Materialraumreferenzen Verweise auf Materialobjekte, die sich im Materialraum befinden, ausgedrückt werden können, sind die im fachlichen Rahmen identifizierten Materialverknüpfungen durch Materialraumreferenzen auf der (programmier-)technischen Ebene modellierbar. In Anlehnung an den C++-Referenzmechanismus stellt das Framework zwei unterschiedliche Arten von Materialraumreferenzen zur Verfügung. Während die Klasse *mr_ref_any* die ungetypte Referenz[27] modelliert, dient die generische template-Klasse *mr_ref*<*T*> zur Modellierung getypter Referenzen.

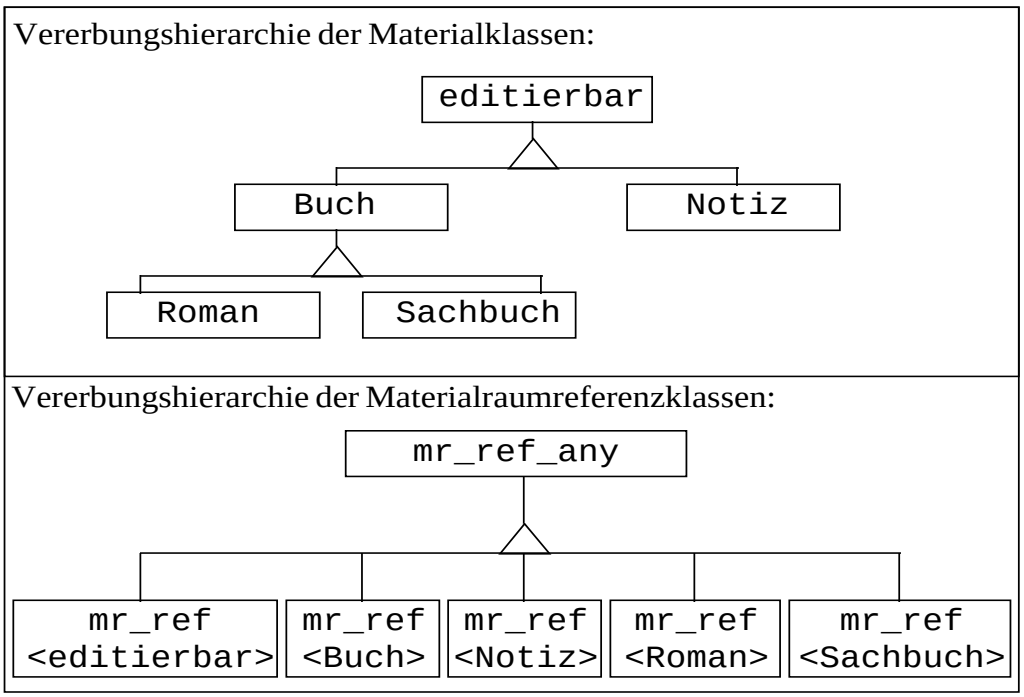


Abbildung 6.6: Vererbungshierarchie von Materialraumreferenzen im Vergleich zu den zugehörigen Materialklassen

Aufgrund der Konstruktion des Materialraumreferenz-Mechanismus (insbesondere durch den Einsatz von Generizität) unterscheidet sich die Vererbungshierarchie der Materialklassen strukturell von der Vererbungshierarchie der korrespondierenden Materialraumreferenzklassen (vgl. Abbildung 6.6). Während die Klasse

[27] vgl. *void**

editierbar Oberklasse der Klasse *Buch* ist, besteht zwischen den beiden Klassen *mr_ref<editierbar>* und *mr_ref<Buch>* keine Vererbungsbeziehung. Dieser Sachverhalt führt zu schwerwiegenden Problemen beim Umgang mit Materialraumreferenzen (vgl. [Edelson 92]). Erwartet zum Beispiel die *set_material()*-Methode eines Editors eine Referenz auf ein editierbares Material (*mr_ref<editierbar>*) und es soll eine Materialraumreferenz auf ein Buch (*mr_ref<Buch>*) übergeben werden, so ergibt der Versuch einer automatischen Typkonvertierung durch C++ einen Fehler, da die beiden Klassen nicht in Vererbungsbeziehung stehen und deshalb nicht ohne weiteres konvertierbar sind. Aus diesem Grund ist jede Klasse *mr_ref<T>* Unterklasse von *mr_ref_any*, so daß stets ein automatisches Casting von einer beliebigen getypten Materialraumreferenz und damit auch von Exemplaren der Klasse *mr_ref<Buch>* nach *mr_ref_any* möglich ist. Da zudem jede Klasse *mr_ref<T>* und somit auch die Klasse *mr_ref<editierbar>* einen Konstruktor bereitstellt, der als Parameter eine ungetypte Materialraumreferenz erwartet[28], kann ein Exemplar der Klasse *mr_ref<editierbar>* erzeugt werden und diesem die zuvor nach *mr_ref_any* konvertierte Materialraumreferenz auf das Buch übergeben werden. In diesem Zusammenhang ist anzumerken, daß das hier beschriebene indirekte Casting nur deshalb möglich ist, weil sowohl ungetypte als auch getypte Materialraumreferenzen den Bezug auf ein Materialobjekt lediglich auf der Basis eines Attributs (*my_MID*) herstellen, so daß es ausreichend ist, die Ausprägung dieses einen Attributs an eine andere Materialraumreferenz zu übertragen[29], damit diese einen eindeutigen Bezug auf ein bestimmtes Materialobjekt herstellen kann.

Neben dieser Möglichkeit der indirekten Typkonvertierung bieten die Klassen *mr_ref_any* und *mr_ref<T>* Methoden[30], um eine Materialraumreferenz als unbesetzt zu markieren (*clear()*) bzw. um zu überprüfen, ob eine Materialraumreferenz unbesetzt ist (*is_NIL()*).

Die Zuweisungs-Operatoren (=) ermöglichen die Kopplung von Materialraumreferenzen an Materialobjekte, die durch bestimmte C++-Referenzen (*_ptr*) oder durch andere Materialraumreferenzen (*_ref*) referenziert werden. Nach einer solchen Zuweisung kann über die Materialraumreferenz auf das bislang nur durch andere Referenzen erreichbare Materialobjekt Bezug genommen werden.

Darüber hinaus erlauben die bereitgestellten Vergleichs-Operatoren == und *!=* die Prüfung, ob das Materialobjekt, das durch die Materialraumreferenz referenziert wird, identisch mit einem Materialobjekt ist, auf das durch eine andere Materialraumreferenz (*refL*, *refR*) oder eine bestimmte C++-Referenz (*ptrL*, *ptrR*) verwiesen wird.

[28] vgl. den Konstruktor *mr_ref<editierbar>(mr_ref_any _ref)*

[29] Eine Anpassung oder Veränderung ist dabei nicht erforderlich.

[30] vgl. Abbildung 6.7 und Abbildung 6.8

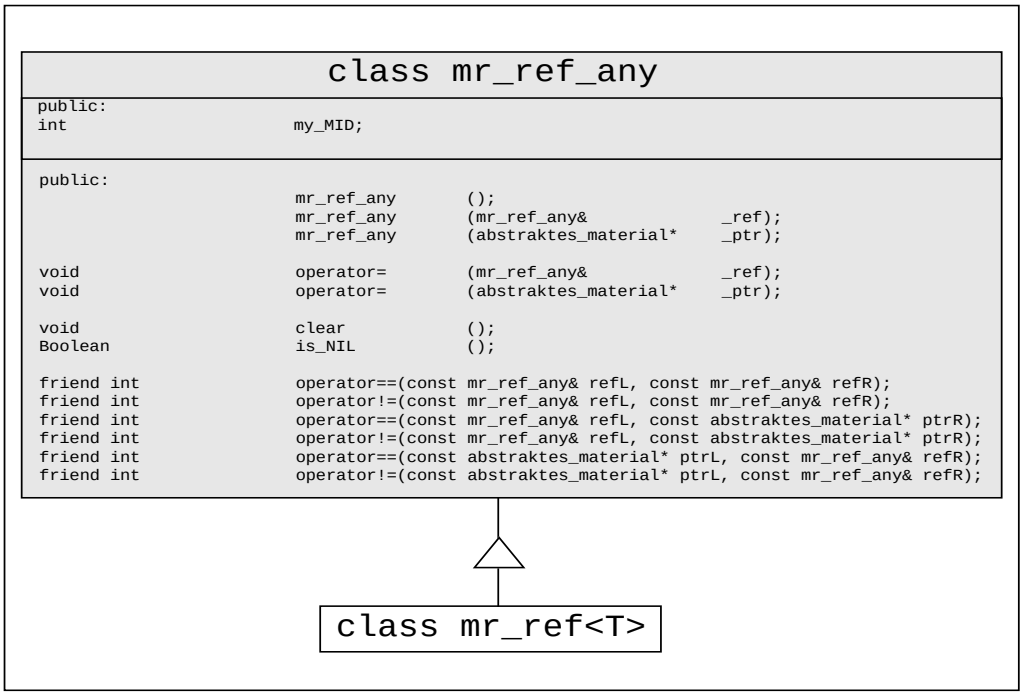


Abbildung 6.7: Schnittstelle der Klasse *mr_ref_any*

Begriff 47 : *Materialidentitätsgleichheit*
technische Interpretation:
Zwei Materialobjekte sind *identisch/identitätsgleich*, wenn sie die gleiche Ausprägung des Materialidentifikators (*MID*) aufweisen. Zwei komplexe Materialien sind genau dann identisch, wenn ihre Kontextmaterialobjekte identisch sind.

Durch den neuen Referenzmechanismus wird das statische Typsystem von C++ überlagert und damit wirkungslos, so daß die Bereitstellung eines neuen Konzepts zur Typprüfung erforderlich ist. Aus diesem Grund stellt das Framework *dinx* eine dynamische Typprüfung zur Verfügung. Da jede Cast-Operation sowohl in Form eines *upcast*[31] als auch eines *downcast*[32] implizit durchgeführt wird, kann auf eine explizite Typkonvertierung im Implementationscode verzichtet werden. Sollte zur Laufzeit ein Typfehler erkannt werden, so wird eine Ausnahmebehandlung (*exception*) angestoßen, die den Konvertierungsversuch abbricht und eine Fehlermeldung ausgibt.

Neben der Prüfung auf Typkonformität muß das Framework einen Mechanismus bereitstellen, der Zugriffe auf nicht mehr existente oder falsche[33] Material-

[31] Konvertierung eines Typs in einen seiner Obertypen
[32] Konvertierung eines Typs in einen seiner Untertypen
[33] Eine Referenz verweist auf ein falsches Objekt, wenn dieses Objekt nicht das Objekt ist, das der Referenz zugeordnet worden ist.

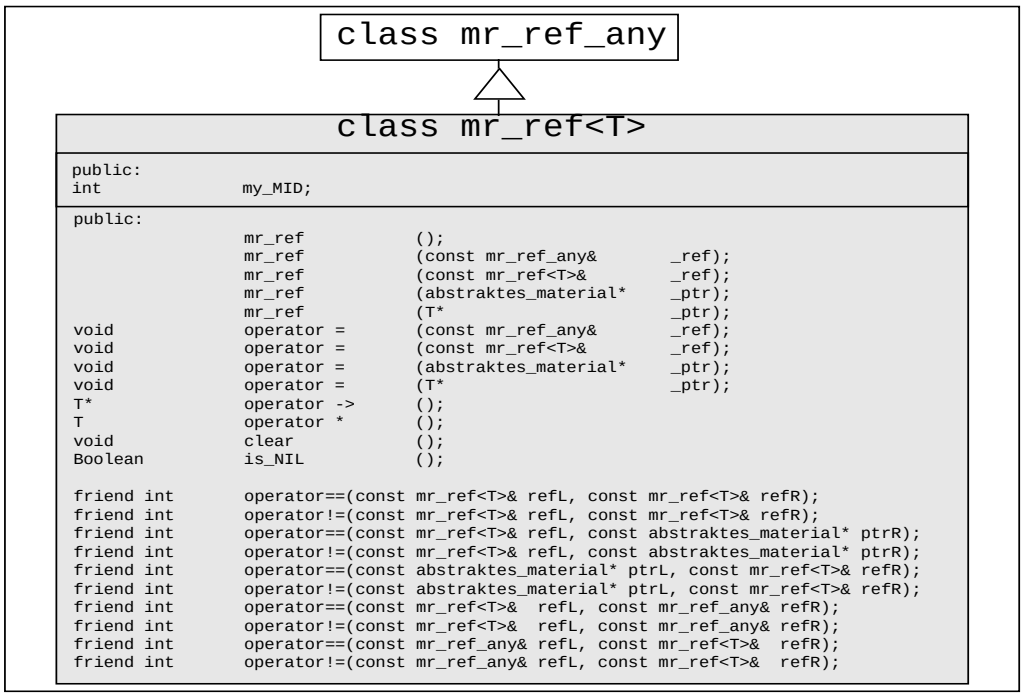


Abbildung 6.8: Schnittstelle der Klasse *mr_ref<T>*

objekte verhindert. Da Materialidentifikatoren von aufgelösten Materialobjekten nicht wiederverwendet werden, ist es a priori nicht möglich, daß eine Materialraumreferenz auf ein falsches Materialobjekt zeigt. Allerdings können Materialraumreferenzen unbesetzt (NIL, NULL) sein oder auf nicht mehr existierende Materialobjekte verweisen (*dangling reference*). In einem System, dessen Referenzen auf der Basis von Materialidentifikatoren modelliert sind, ist daher vor jeder Dereferenzierung die Validität der Referenz zu überprüfen. In Anlehnung an [Cattell 94] gibt es dafür prinzipiell drei verschiedene Möglichkeiten:

1. *Keine Validierung:*
   Der Implementator erhält durch das System keine Unterstützung und muß eigenverantwortlich für die Sicherstellung der referentiellen Integrität sorgen.

2. *Statische Zusicherung:*
   Die Gültigkeit von Materialidentifikatoren wird statisch zugesichert. Materialobjekte können nicht explizit aufgelöst werden. Sie werden durch eine zusätzliche Instanz erst dann aus dem Materialraum entfernt (*garbage collection*[34]), wenn keine Materialraumreferenz mehr auf sie verweist[35].

3. *Dynamische Validierung:*
   Die Gültigkeit von Materialidentifikatoren wird zur Laufzeit geprüft. Sollte

[34] garbage collection engl. (Speicherrückgewinnung)
[35] vgl. O2 und GemStone

eine Referenz, die entweder unbesetzt (NIL, NULL) ist oder die auf ein nicht existentes Materialobjekt verweist, dereferenziert werden, dann kommt es zu einer Ausnahmebehandlung[36].

Die Realisierung der referentiellen Integrität auf der Basis der dynamischen Validierung von Materialidentifikatoren erscheint uns adäquat, weil sie unserer Ansicht nach im Gegensatz zur statischen Validierung, bei der keine explizite Materialauflösung möglich ist, dem intuitiven Umgang mit Materialien der realen Welt entspricht. Damit kommt es auch bei dem Versuch, eine *dangling reference* zu dereferenzieren, zu einer Ausnahmebehandlung, die die Dereferenzierung der Materialraumreferenz abbricht und eine Fehlermeldung ausgibt[37].

6.2.4 Materialkomposition

Im Gegensatz zur Materialverknüpfung, die wie in C++ üblich auf der Basis von Referenzen etabliert werden kann, gestaltet sich die Abbildung der fachlich generellen Materialkomposition in den (programmier-)technischen Kontext schwieriger. Die Modellierung komplexer Materialien in einer einzigen Materialklasse erscheint im allgemeinen Fall nicht adäquat, da ein solches Vorgehen die Eigenständigkeit der Komponentenmaterialien vernachlässigt und damit eine zu starke Bindung zwischen Kontext- und Komponentenmaterialien die Folge wäre (vgl. Abbildung 6.9).

Beispielsweise würde die technische Modellierung des komplexen Materials Automobil in einer einzigen Materialklasse die Komponentenmaterialobjekte (zum Beispiel Rad und Sitz) als Geheimnis des Automobils kapseln[38]. Ein solches Vorgehen würde der in Kapitel 4 definierten Materialkomposition in den folgenden Punkten widersprechen:

- *Kapselung der Komponentenmaterialien*:
  In der fachlichen Welt ist ein direkter Zugriff auf Komponentenmaterialien möglich, so daß die Kapselung den Aspekt des physikalischen Enthaltenseins zu stark betonen würde und damit ein Zugriff auf das Komponentenmaterialobjekt nur indirekt über das Kontextmaterialobjekt möglich wäre.

- *Verlust der Eigenständigkeit der Komponentenmaterialien*:
  Die Eigenständigkeit der Komponentenmaterialien wird bei der Modellierung komplexer Materialien in einer einzigen Materialklasse vernachlässigt.

[36] vgl. ONTOS

[37] Dabei handelt es sich um eine Ausnahmebehandlung, die sich auf die fachlich spezifische Welt durchschlagen darf, da der Versuch unternommen worden ist, auf ein fehlendes Komponentenmaterial oder ein nicht mehr existentes Material zuzugreifen. Es könnten zum Beispiel Meldungen wie ,,Die Radaufhängung des Automobils ist unbesetzt!" oder ,,Das Automobil ist bereits verschrottet worden!" ausgegeben werden.

[38] vgl. hierzu Abbildung 6.9

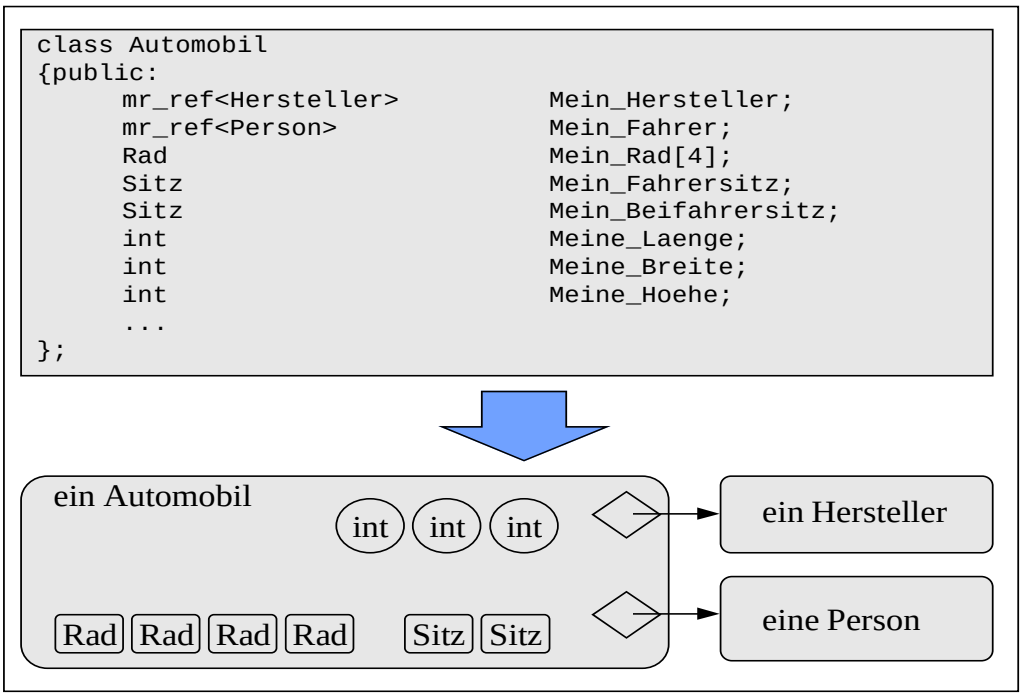


Abbildung 6.9: Modellierung eines Automobils in einer einzigen Materialklasse

Ein Komponentenmaterialobjekt muß erst aus der Repräsentation des komplexen Materials „herausdupliziert" werden, damit es getrennt vom Kontextmaterialobjekt betrachtet werden kann. Ein solches Vorgehen steht im krassen Gegensatz zum fachlichen Modell, weil beim Herauskopieren dem aus einem Auto entnommenen Fahrersitz eine andere (neue) Materialidentität zugeordnet werden würde.

- *Kopplung der Lebensdauern*:
  Das Kontextmaterialobjekt kann ausschließlich zusammen mit seinen Komponentenmaterialobjekten aufgelöst werden, wodurch die Lebensdauer eines Komponentenmaterialobjekts direkt an die Lebensdauer des umgebenden Kontextmaterialobjekts gebunden wird. Das widerspricht der prinzipiellen Unabhängigkeit der Lebensdauer eines Komponentenmaterials von der Lebensdauer des zugehörigen Kontextmaterials in der realen Welt[39].

Wie wir gesehen haben, ist die Modellierung komplexer Materialien innerhalb einer einzigen Materialklasse kein adäquates Vorgehen. Vielmehr ist eine losere Kopplung nötig. Da „in rein objektorientierten Programmiersprachen Zugehörigkeiten immer über Referenzen definiert" werden ([Booch 94]), ist die Idee naheliegend, auch die Komponentenbeziehung auf der Basis von Referenzen zu

[39] Beispielsweise kann die Lebensdauer eines Autositzes von der Lebensdauer des Automobils abhängen, wenn beide Materialien zum gleichen Zeitpunkt verschrottet werden. Es ist aber auch möglich den Sitz vor der Verschrottung aus dem Auto auszubauen.

modellieren. Aus diesem Grund etablieren wir neben der Materialraumreferenz *mr_ref<T>* die sogenannte Komponentenmaterialraumreferenz *mr_cmp<T>*[40], die zur (programmier-)technischen Realisierung der Materialkomposition dient. Wie wir im nächsten Unterabschnitt sehen werden, macht sich der semantische Unterschied der beiden unterschiedlichen Materialraumreferenzen insbesondere bei der Nutzung der fachlich generellen Umgangsformen (zum Beispiel Materialerzeugung und Materialauflösung) bemerkbar.

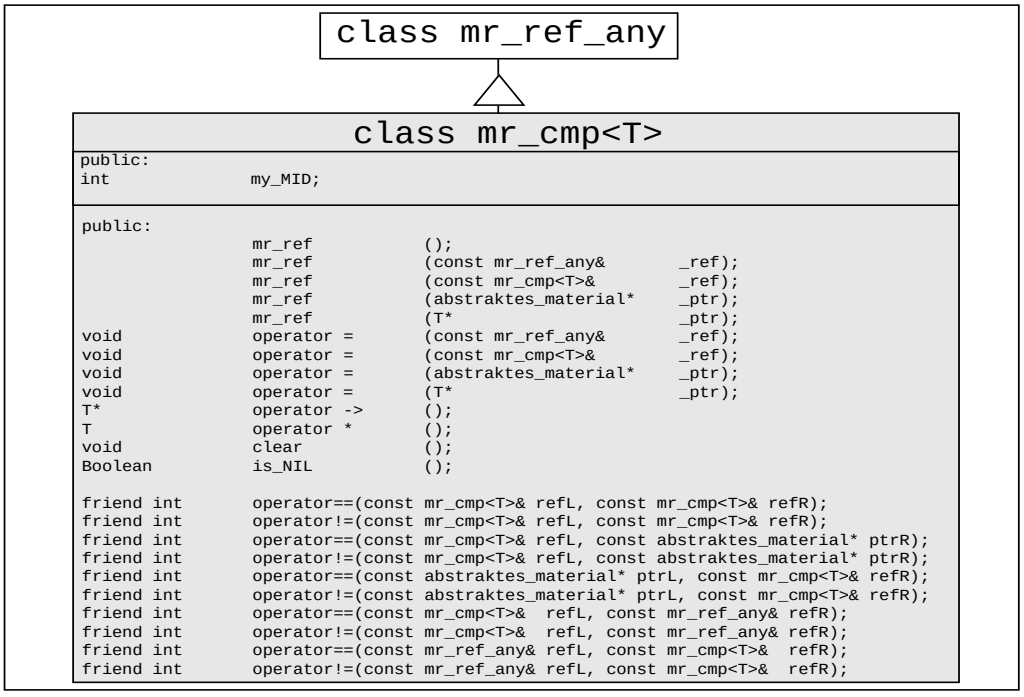


Abbildung 6.10: Schnittstelle der Klasse *mr_cmp<T>*

Durch die Einführung dieser Komponentenmaterialraumreferenz wird es möglich, Materialverknüpfungen und Materialkompositionen auch in der (programmier-) technischen Umsetzung anhand unterschiedlicher Referenzen zu unterscheiden und damit komplexe Materialien ohne Strukturbruch zwischen dem fachlichen und dem (programmier-)technischen Modell in mehr als einer Materialklasse zu modellieren.

Komplexe Materialien können in unterschiedlichen Materialklassen, deren Exemplare über Komponentenmaterialraumreferenzen in Beziehung stehen, modelliert werden. Jede Komponentenmaterialraumreferenz kann beliebig häufig gesetzt und als unbesetzt markiert (NIL) werden. Da es eine ausgezeichnete Ausprägung (NIL) zur Modellierung unbesetzter Komponentenmaterialraumreferenzen gibt,

[40] vgl. Abbildung 6.10

ist es möglich, einem Kontextmaterialobjekt das Fehlen eines zugehörigen Komponentenmaterialobjekts anzusehen (*is_NIL()*).

Die Modellierung des komplexen Materials Automobil in mehreren Materialklassen kann der Abbildung 6.11 entnommen werden. Die einzelnen Objekte vom Typ *Rad* bzw. *Sitz* sind damit nicht mehr mit dem Exemplar der Klasse *Automobil* fest verdrahtet, sondern werden mit dem Automobil über Komponentenmaterialraumreferenzen gekoppelt.

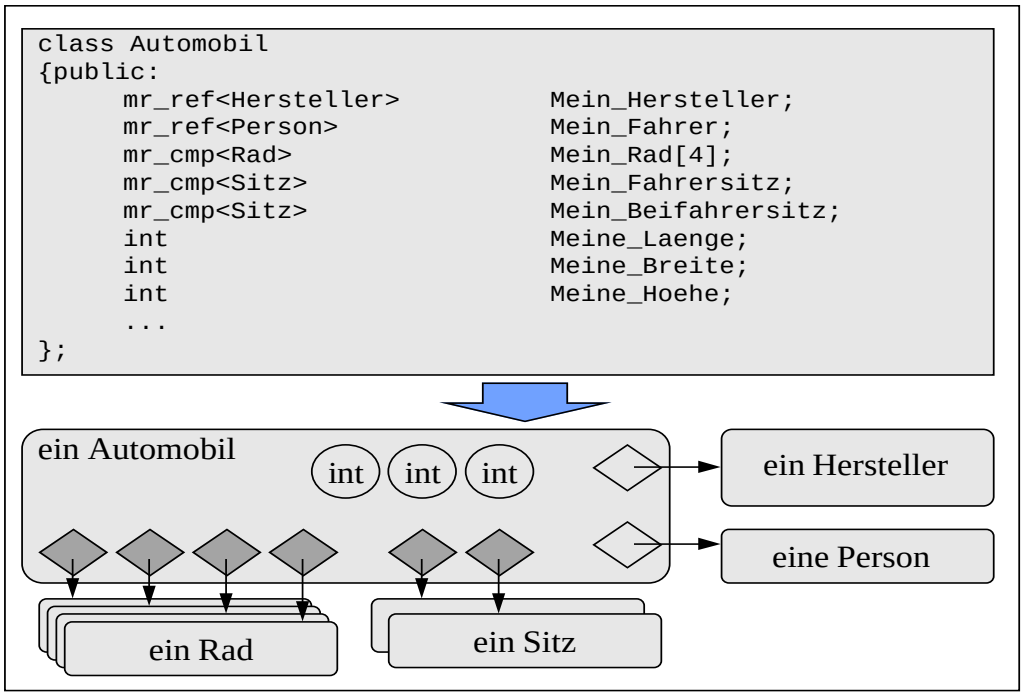


Abbildung 6.11: Modellierung eines Automobils in mehreren Materialklassen

Im Hinblick auf die Lebensdauer von Materialien haben wir in Kapitel 4 festgestellt, daß Materialien (auch wenn sie Teil eines komplexen Materials sind) prinzipiell eine beliebige Lebensdauer haben. Diese Lebensdauer wird losgelöst von technischen Ereignissen (zum Beispiel Herunterfahren einer Anwendung) durch den Zeitpunkt der expliziten Erzeugung sowie den Zeitpunkt der expliziten Auflösung des Materials bestimmt. Im (programmier-)technischen Kontext können wir damit die Lebensdauer nicht auf den transienten Gültigkeitsbereich einschränken, so daß alle Materialobjekte a priori als persistente Objekte zu realisieren sind.

Begriff 48 : *Lebensdauer eines Materials*
technische Interpretation:
Exemplare von Materialklassen sind a priori persistent.

6.2.5 Bereitstellung fachlich genereller Umgangsformen

Durch die Einführung des Materialraumreferenz-Mechanismus lassen sich die fachlich generellen Umgangsformen von Materialien, die in Kapitel 4 ausführlich beschrieben worden sind, im (programmier-)technischen Kontext umsetzen. Die einzelnen fachlich generellen Umgangsformen sind als Methoden der Klassen *mr_ref_any*, *mr_ref<T>* und *mr_cmp<T>* realisiert, so daß sie nicht in jeder Materialklasse implementiert werden müssen. Auf diese Weise gelingt es, den Umfang der zu entwerfenden Materialklassen klein zu halten und die fachlich generellen Umgangsformen von den fachlich spezifischen zu trennen[41].

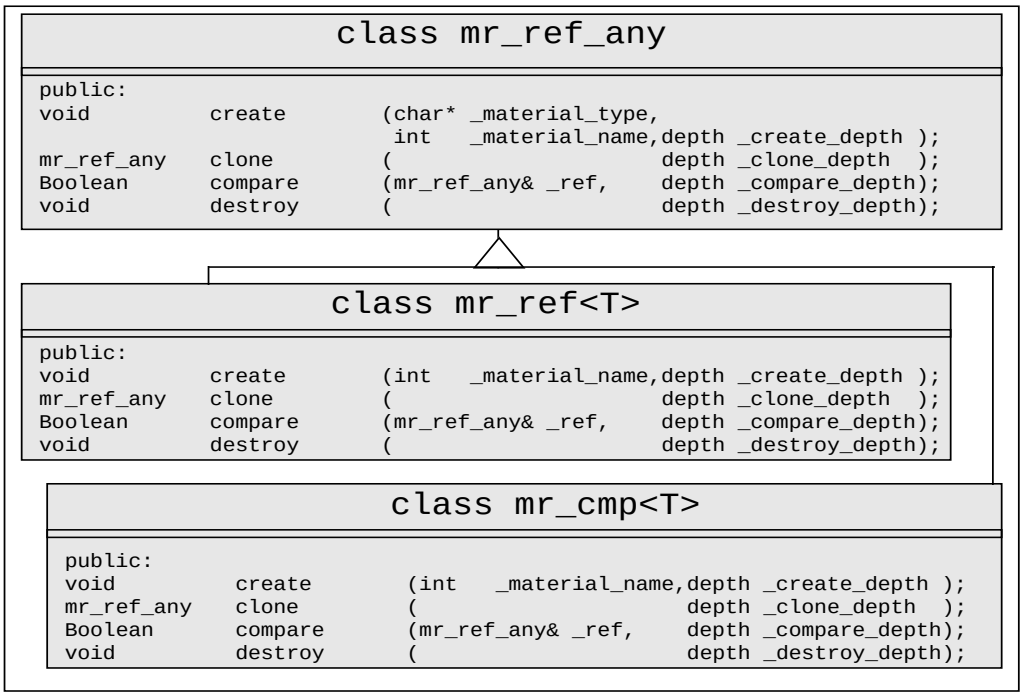


Abbildung 6.12: Erweiterung der Schnittstellen von *mr_ref_any*, *mr_ref<T>* und *mr_cmp<T>*

Alle Materialraumreferenzen stellen Operationen zur Materialerzeugung (*create()*), zur Erstellung von Materialkopien (*clone()*), zur Prüfung von Materialien auf Zustandsgleichheit (*compare()*) und zur Auflösung von Materialien (*destroy()*) bereit (vgl. Abbildung 6.12). Es kann angegeben werden, ob die Operationen flach oder tief ausgeführt werden sollen.

[41] Während die Nutzung einer fachlich spezifischen Umgangsform zur Ausführung einer Operation auf einem dynamisch erzeugten Materialobjekt führt, impliziert die Nutzung fachlich genereller Umgangsformen den Aufruf einer Operation auf einer statisch erzeugten Materialraumreferenz. Aus diesem Grund werden alle Operationen, die fachlich spezifische Umgangsformen widerspiegeln, durch *mr_ref<T>->Operation_X* aufgerufen und alle Operationen, die fachlich generelle Umgangsformen realisieren, durch *mr_ref<T>.Operation_Y* angestoßen.

- *Materialerzeugung (create()):*
  Während bei der Materialerzeugung über eine ungetypte Materialraumreferenz der Typ (*_material_type*), von dem ein Exemplar zu erzeugen ist, spezifiziert werden muß, wird bei den getypten Materialraumreferenzen (*mr_ref<T>* und *mr_cmp<T>*) stets ein Exemplar des zugeordneten Typs (*T*) in der angegebenen Tiefe (*_create_depth*) erzeugt.

  Bei der Materialerzeugung wird ein (Kontext-)Materialobjekt erzeugt, dessen Materialidentifikator (*MID*) und Materialtypidentifikator (*CID*) initial gesetzt und dessen Materialraumreferenzen als unbesetzt (NIL) kenntlich gemacht werden. Während bei der flachen Materialerzeugung auch alle Komponentenmaterialraumreferenzen als unbesetzt (NIL) markiert werden, werden bei der tiefen Materialerzeugung zu allen Komponentenmaterialraumreferenzen des (Kontext-)Materialobjekts Exemplare adäquater Materialklassen tief erzeugt und den Komponentenmaterialraumreferenzen zugeordnet.

  Im Anschluß an die eigentliche Materialerzeugung wird dem erzeugten (Kontext-)Materialobjekt der übergebene fachliche Materialname (*_material_name*) zugeordnet[42] und an die Materialraumreferenz gebunden.

- *Erstellung einer Materialkopie (clone()):*
  Es wird eine Kopie des durch die Materialraumreferenz referenzierten (Kontext-)Materialobjekts in der angegebenen Tiefe (*_clone_depth*) erstellt. Die Operation gibt eine ungetypte Materialraumreferenz auf das neu erzeugte (Kontext-)Materialobjekt zurück.

  Bei der Erstellung einer flachen Materialkopie wird ein neues Exemplar der Materialklasse des (Kontext-)Materialobjekts flach erzeugt (flache Materialerzeugung), so daß alle Materialraumreferenzen und Komponentenmaterialraumreferenzen des neu erzeugten (Kontext-)Materialobjekts als unbesetzt (NIL) gekennzeichnet sind. Anschließend werden die Ausprägungen der einzelnen Attribute des zu kopierenden (Kontext-)Materialobjekts auf das neu erzeugte (Kontext-)Materialobjekt übertragen, wobei eine Übertragung der Ausprägungen ausgezeichneter Attribute (Materialidentifikator (*MID*) und Materialtypidentifikator (*CID*)) nicht erfolgt. Bei der Erstellung einer tiefen Materialkopie werden zusätzlich alle Komponentenmaterialobjekte, die über die Komponentenmaterialraumreferenzen des zu kopierenden (Kontext-)Materialobjekts referenziert werden, tief kopiert. Anschließend erfolgt im Rahmen der Erstellung einer tiefen Materialkopie die Zuordnung der auf diese Weise tief kopierten Komponentenmaterialobjekte

[42] Den Komponentenmaterialobjekten wird kein Materialname zugeordnet. Sie bleiben bis zum Zeitpunkt ihrer expliziten Benennung (*materialraum::set_material_name()*) unbenannt.

zu den Komponentenmaterialraumreferenzen des neu erzeugten (Kontext-) Materialobjekts.

- *Prüfung auf Materialzustandsgleichheit (compare()):*
  Es wird das durch die Materialraumreferenz referenzierte (Kontext-)Materialobjekt mit einem durch eine andere Materialraumreferenz (*_ref*) referenzierten (Kontext-)Materialobjekt auf Zustandsgleichheit in der angegebenen Tiefe (*_compare_depth*) geprüft. Das Ergebnis wird in Form eines *Boolean*-Wertes zurückgegeben.

  Bei der Prüfung von zwei (Kontext-)Materialobjekten auf Materialzustandsgleichheit werden zunächst die Materialtypidentifikatoren (CID) der beiden Objekte auf Gleichheit geprüft. Wenn beide (Kontext-)Materialobjekte den gleichen Typ haben, wird anschließend überprüft, ob die nicht ausgezeichneten Attribute beider (Kontext-)Materialobjekte gleich sind. Ein Vergleich der Ausprägungen der Materialidentifikatoren (*MID*) und der Materialraumreferenzen findet nicht statt. Während bei der Prüfung auf flache Materialzustandsgleichheit auch kein Vergleich der Komponentenmaterialraumreferenzen erfolgt, wird bei der tiefen Materialzustandsgleichheit geprüft, ob die durch die Komponentenmaterialraumreferenzen referenzierten Komponentenmaterialobjekte tief zustandsgleich sind.

- *Materialauflösung (destroy()):*
  Es wird das durch die Materialraumreferenz referenzierte (Kontext-)Materialobjekt in der angegebenen Tiefe (*_destroy_depth*) aufgelöst.

  Bei der flachen Materialauflösung erfolgt ausschließlich die Vernichtung des (Kontext-)Materialobjekts. Im Vergleich zur flachen Materialauflösung, bei der die durch Materialraumreferenzen und Komponentenmaterialraumreferenzen referenzierten Materialobjekte bestehen bleiben, werden bei der tiefen Materialauflösung alle durch Komponentenmaterialraumreferenzen referenzierten Komponentenmaterialobjekte tief aufgelöst und es bleiben lediglich die durch Materialraumreferenzen referenzierten Materialobjekte bestehen.

Durch die Bindung der fachlich generellen Umgangsformen an die Materialraumreferenzen bleibt dem Implementator fachlich spezifischer Materialklassen die Implementation der fachlich generellen Umgangsformen verborgen und er kann diese bei der (programmier-)technischen Umsetzung fachlich spezifischer Konzepte voraussetzen und nutzen[43].

[43] vgl. Abbildung 6.12

6.2.6 Zugriffskontrolle durch das Konzept der Umgangsarten

Neben dem eigentlichen Zugriff auf Materialobjekte über die Dereferenzierung von Materialraumreferenzen dient der Materialraum auch zur Synchronisation nebenläufiger Handlungen an bzw. mit Materialobjekten. Im Gegensatz zu den Ansätzen [Wulf, Weske 94] und [Gryczan 95] favorisieren wir eine Zugriffskontrolle, bei der sich beliebige Komponenten der fachlichen Welt[44] bei dem Materialraum die Umgangsarten Original, Kopie, Sicht, Präsenz auf bestimmte Materialobjekte akquirieren können. Auf diese Weise gelingt es, daß die Entscheidung, auf welchen Ebenen (Arbeitsumgebungen und/oder Arbeitsmitteln) eine fachliche Zugriffskontrolle zu etablieren ist, im jeweiligen fachlich spezifischen Kontext getroffen werden kann. Damit ist im Gegensatz zum Konzept der Materialverwaltung nicht a priori festgelegt, ob eine Zugriffskontrolle innerhalb von Arbeitsumgebungen auf der Ebene von Arbeitsmitteln zu etablieren ist oder nicht. Dies ist insbesondere deshalb interessant, weil [Gryczan 95] im Gegensatz zu [Wulf, Weske 94] und [Riehle 95] auf eine fachliche Zugriffskontrolle innerhalb von Arbeitsumgebungen verzichtet.

Um trotz der fehlenden Zugriffskontrolle eine Synchronisation innerhalb von Arbeitsumgebungen zu erreichen, führt [Gryczan 95] einen Benachrichtigungsmechanismus ein, der Werkzeuge und Automaten benachrichtigt, wenn bestimmte Materialien durch andere Werkzeuge oder Automaten modifiziert worden sind. Um die Sichten von [Wulf, Weske 94], [Riehle 95] und [Gryczan 95] gleichzeitig unterstützen zu können, bietet der Materialraum zwei unterschiedliche Verfahren an: Zum einen sind beliebige Komponenten der fachlichen Welt (zum Beispiel Werkzeuge und Automaten) in der Lage, sich Umgangsarten mit Materialien beim Materialraum zu akquirieren (fachliche Zugriffskontrolle) und zum anderen können sich die fachlichen Komponenten beim Materialraum anmelden, um beim Eintritt bestimmter Ereignisse (zum Beispiel Modifikationen an einem Materialobjekt) benachrichtigt zu werden (Benachrichtigungsmechanismus).

Sowohl das Konzept der Umgangsarten als auch den Benachrichtigungsmechanismus fachlicher Komponenten durch den Materialraum haben wir im Rahmen der prototypischen Realisierung des Frameworks *dinx* nicht umgesetzt, da es das primäre Ziel der prototypischen Realisierung gewesen ist, die Realisierbarkeit von Materialraumreferenzen zu evaluieren und damit auch die Architektur, die zur Bereitstellung der technischen Transparenz dient, mit Hilfe eines Labormusters zu testen und zu optimieren.

[44] nicht nur Arbeitsumgebungen, sondern zum Beispiel auch Werkzeuge und Automaten

Neben der Erweiterung der Schnittstelle der Klasse *materialraum* um eine Methode (*register()*), die die Anmeldung von fachlich spezifischen Komponenten erlaubt (Benachrichtigungsmechanismus), sind die Schnittstellen der Materialraumreferenzklassen *mr_ref_any*, *mr_ref<T>* und *mr_cmp<T>* um die Funktionalität zur Akquisition (*acquire()*) und zur Freigabe von Umgangsarten (*release()*) in einer bestimmten Tiefe (*_acquire_depth* bzw. *_release_depth*) zu erweitern (vgl. Abbildung 6.13):

- *Umgangsartakquisition (acquire()):*
  Bei der flachen Umgangsartakquisition erfolgt ausschließlich die Akquisition einer gewählten Umgangsart mit einem bestimmten (Kontext-)Materialobjekt. Eine Akquisition von Umgangsarten mit Materialobjekten, die durch die Materialraumreferenzen oder Komponentenmaterialraumreferenzen des (Kontext-)Materialobjekts referenziert werden, erfolgt dabei nicht. Im Gegensatz dazu, wird bei der tiefen Umgangsartakquisition nicht nur eine bestimmte Umgangsart mit dem (Kontext-)Materialobjekt akquiriert, sondern es erfolgt gleichzeitig die tiefe Akquisition der gleichen Umgangsart mit den Komponentenmaterialobjekten, die durch die Komponentenmaterialraumreferenzen des (Kontext-)Materialobjekts referenziert werden. Auch bei der tiefen Umgangsartakquisition werden Umgangsarten mit den durch die Materialraumreferenzen des (Kontext-)Materialobjekts referenzierten Materialobjekten nicht akquiriert.

  Der Rückgabewert der *acquire()*-Operation gibt an, ob die Akquisition erfolgreich war oder ob die angeforderte Umgangsart aufgrund der (nicht vorhandenen) Verträglichkeit mit bereits bestehenden Umgangsarten nicht gewährt werden konnte.

- *Umgangsartfreigabe (release()):*
  Bei der Umgangsartfreigabe erfolgt die Freigabe der Umgangsart mit einem bestimmten (Kontext-)Materialobjekt. Im Vergleich zur flachen Umgangsartfreigabe, bei der bestehende Umgangsarten mit Materialobjekten, auf die das (Kontext-)Materialobjekt über Materialraumreferenzen oder Komponentenmaterialraumreferenzen verweist, erhalten bleiben, werden bei der tiefen Umgangsartfreigabe zusätzlich bestehende Umgangsarten mit Materialobjekten, auf die das (Kontext-)Materialobjekt über eine Komponentenmaterialraumreferenz verweist, tief freigegeben. Bestehende Umgangsarten mit Materialobjekten, auf die das (Kontext-)Materialobjekt über Materialraumreferenzen verweist, bleiben erhalten.

Durch diese Realisierungen der Umgangsartakquisition bzw. der Umgangsartfreigabe wird es möglich, daß das Kontextmaterialobjekt und die Komponentenmaterialobjekte eines komplexen Materials in beliebigen (auch unterschiedlichen)

Umgangsarten bearbeitet werden können und damit eine Mischung von Umgangsarten möglich ist.

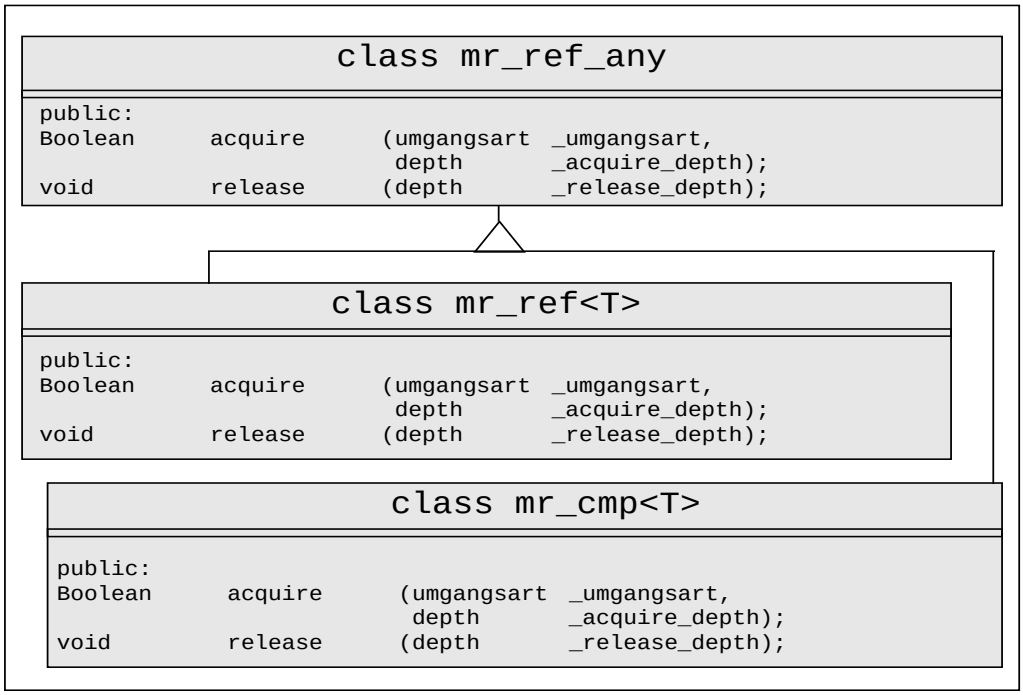


Abbildung 6.13: 2. Erweiterung der Schnittstellen von *mr_ref_any*, *mr_ref*<*T*> und *mr_cmp*<*T*>

Es ist festzuhalten, daß die Akquisition einer Umgangsart mit einem Materialobjekt nicht dessen Transfer aus dem persistenten Speicher in den transienten Speicher impliziert. Im Gegensatz zum Ansatz [Wulf, Weske 94] erfolgt die Akquisition und die Freigabe von Umgangsarten unabhängig von der Aktivierung sowie Passivierung von Materialobjekten. Erst zum Zeitpunkt eines Objektzugriffs wird geprüft, ob sich das zugehörige Materialobjekt im transienten Speicher befindet oder nicht. Sollte es bislang nicht in den transienten Speicher transferiert oder bereits wieder aus diesem verdrängt worden sein, wird es transparent in den transienten Speicher geholt und dort gecacht (vgl. Unterabschnitt 6.3.2).

6.3 Komponenten der Adaptionsschicht

In diesem Abschnitt wollen wir die Komponenten der Adaptionsschicht beschreiben, die die Funktionalität des Materialraums und der Materialraumreferenzen bereitstellen (vgl. Abbildung 6.14). Neben den bereits in Abschnitt 6.2 betrachteten Komponenten (zum Beispiel Materialraum und Materialraumreferenzen)

werden wir in den folgenden Unterabschnitten die Komponenten *Materialraumkoordinator*, *Materialraumverwalter* und *Materialraumversorger* ausführlich betrachten, um einen Einblick zu geben, auf welche Weise im Framework *dinx* die Speichertrennung überwunden und damit die technische Transparenz insbesondere zwischen dem transienten und dem persistenten Speicher erreicht werden kann.

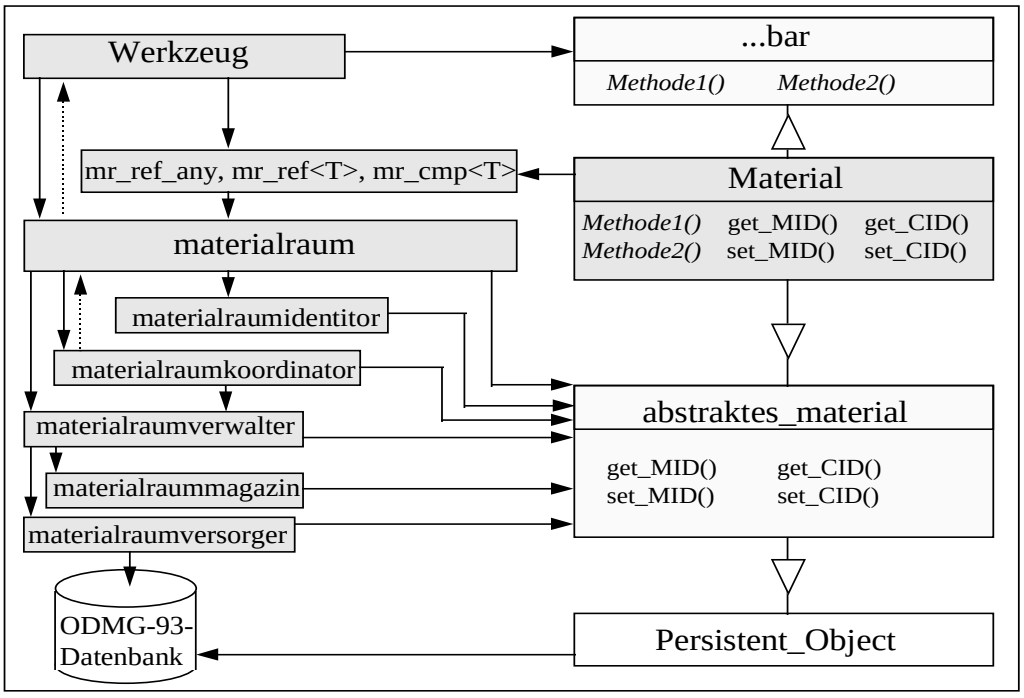


Abbildung 6.14: Klassenentwurf des Materialraums

6.3.1 Der Materialraumkoordinator

Um die fachliche Zugriffskontrolle nach dem Konzept der Umgangsarten realisieren zu können, benötigen wir einen *Materialraumkoordinator*, der primär zur Koordination der fachlichen Umgangsarten dient. Der Materialraumkoordinator sorgt für die adäquate Zuteilung der Umgangsarten gemäß der in Kapitel 4 beschriebenen Verträglichkeiten von angeforderten und bestehenden Umgangsarten. Zu diesem Zweck besitzt der Materialraumkoordinator stets einen umfassenden Überblick über alle bestehenden Umgangsarten mit Materialobjekten innerhalb des Materialraums[45]. Entstehende *deadlock*-Situationen bei der Akquisition von Umgangsarten werden nicht automatisch durch den Materialraumkoordinator

[45]Der Materialraumkoordinator verfügt über ein Verzeichnis aller bestehenden Umgangsarten, die beliebige fachlich spezifische Komponenten mit Materialobjekten zu einem bestimmten Zeitpunkt haben.

aufgelöst, sondern müssen auf der fachlich spezifischen Ebene erkannt und explizit gemäß der ausgehandelten Konventionen zur Koordination innerhalb der Arbeitsdomäne aufgelöst werden.

Da sich die Synchronisation von Handlungen auf den gesamten Materialraum bezieht, wirkt sich die durch den Materialraumkoordinator bereitgestellte Zugriffskontrolle auf das gesamte Informationssystem aus und nicht nur auf einen lokalen Prozeß. Aus diesem Grund wäre es unzureichend, einzelne Materialraumkoordinatoren unabhängig voneinander innerhalb einzelner lokaler Prozesse zu realisieren. Vielmehr sollten hier nur Materialraumkoordinator-Proxies[46] realisiert werden, die untereinander kommunizieren, um in ihrer Gesamtheit die hier beschriebenen Aufgaben eines Materialraumkoordinators bereitzustellen. Dieses kann durch Delegation der Aufträge an einen zentralen Materialraumkoordinator[47], der sternförmig mit allen Proxies verbunden ist, oder auf ausschließlicher Abstimmung der einzelnen Materialraumkoordinator-Proxies untereinander erfolgen. In Unterabschnitt 5.6.3 haben wir bereits eine Reihe unterschiedlicher Management-Konzepte vorgestellt. Ferner sei an dieser Stelle auch auf [Coulouris et al. 94] und [Mairandres 96] verwiesen.

Wie wir bereits im vorhergehenden Abschnitt dargestellt haben, stellt der Materialraum analog zum Ereignisverwalter (vgl. [Gryczan 95]) einen Benachrichtigungsmechanismus zur Verfügung, mit dessen Hilfe der Materialraum durch die Komponenten der fachlich spezifischen Welt beobachtbar (vgl. das Entwurfsmuster Beobachter[48] ([Gamma et al. 95]) und das Entwurfsmuster Ereignismechanismus ([Riehle 95])) ist. Somit können sich die fachlich spezifischen Komponenten (zum Beispiel Werkzeuge) bei dem Materialraum anmelden, um beim Eintritt eines bestimmten Ereignisses (zum Beispiel Akquisition einer bestimmten Umgangsart mit einem bestimmten Materialobjekt) durch den Materialraum benachrichtigt zu werden und anschließend auf das Ereignis reagieren zu können. Der dazu erforderliche Nachrichtenaustausch zwischen den einzelnen Rechnerknoten erfolgt ebenfalls über den Materialraumkoordinator.

Um die Umgangsart Sicht adäquat zu modellieren, ist es erforderlich, neben der Synchronisation von Handlungen eine sogenannte horizontale Änderungspropagierung durchzuführen. Aufgabe dieser Propagierung ist es, dem Sichtinhaber eine stets aktuelle Sicht auf das Materialobjekt bereitzustellen, indem vollzogene Änderungen unmittelbar an der Sicht nachvollzogen werden[49] (vgl. den

[46] vgl. das Entwurfsmuster Proxy ([Gamma et al. 95])

[47] vgl. [Wulf, Weske 94]

[48] Zweck des Beobachtermusters: „Definiere eine 1-zu-n Abhängigkeit zwischen Objekten, so daß die Änderung des Zustands eines Objekts dazu führt, daß alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden." ([Gamma et al. 95])

[49] vgl. hierzu [Rahm 93]

in [Gryczan 95] beschriebenen Ereignisverwalter). Der Materialraumkoordinator übernimmt ebenfalls die Aufgabe der Änderungspropagierung, da er stets über den aktuellen Stand der akquirierten Umgangsarten verfügt und somit auf einfache Art und Weise Mitteilungen über Modifikationen am Original an die Sichtinhaber triggern kann[50].

Das Konzept der Sicht wird auf der technischen Ebene durch ein ,,technisch zustandsgleiches" Materialobjektreplikat realisiert[51], das durch den Sichtinhaber nicht modifizierbar ist. Wenn das Original modifiziert worden ist, wird das zugehörige Materialobjektreplikat beim Sichtinhaber invalidiert und beim nächsten Zugriff automatisch aktualisiert. Die Werkzeuge, die das Materialobjektreplikat aktuell benutzen, sollten sich daher beim Materialraum angemeldet haben, um im Fall einer Änderung am Materialoriginal bzw. dessen Auflösung benachrichtigt zu werden (Benachrichtigungsmechanismus). Kommt es beispielsweise zu einer Modifikation an dem Materialoriginal, so greift das Werkzeug auf das invalidierte Materialobjektreplikat zu, welches automatisch und transparent aktualisiert wird, so daß das Werkzeug den neuen Zustand des Materials sondiert und damit dem Anwender die aktuelle Sicht auf das Materialoriginal bereitstellt. Mit diesem Ansatz verfolgen wir eine sogenannte Broadcast-Strategie ([Rahm 93]). Da zu einem Zeitpunkt maximal ein Anwender im Besitz des Materialoriginals ist und nur dieser das Material modifizieren darf, müssen lediglich die von diesem Anwender durchgeführten Änderungen an alle Sichtinhaber getriggert werden. Auf diese Weise kann der Kommunikationsaufwand gering gehalten werden ([Rahm 93]).

Eine Abbildung und damit Modellierung der fachlichen Umgangsarten auf Basis der von einigen Datenbanksystemen bereitgestellten Sperrverfahren ist nicht möglich, da Komponenten der Adaptionsschicht nicht auf Interna der Abstraktionsschicht zugreifen können. Der Materialraumkoordinator ist damit ausschließlich auf Basis des ODMG-93-Standards zu realisieren, der bislang keine expliziten Sperrmechanismen zur Verfügung stellt. Die Nutzung der Sperrverfahren von Datenbanksystemen ist somit erst dann möglich, wenn die ODMG das Konzept eines Sperrmechanismus in den ODMG-Standard integriert. Bis dahin kann das Konzept der Umgangsart ausschließlich in der Adaptionsschicht unter Nutzung der ODMG-93-Schnittstelle der Abstraktionsschicht umgesetzt werden. Andernfalls wäre die Adaptionsschicht nicht mehr portabel, weil sie von einem bestimmten Datenbanksystemhersteller abhängig werden würde.

[50] In [Rahm 93] wird festgestellt, daß es sinnvoll ist, Mechanismen, die zur Synchronisation oder zur Kohärenzkontrolle dienen, zu bündeln, um den Kommunikationsaufwand zu reduzieren. Aus diesem Grund haben wir uns dafür entschieden, daß eine einzige Komponente (Materialraumkoordinator) sowohl zur Koordination der Umgangsarten (Synchronisation) als auch zur Propagierung von Änderungen (Kohärenzkontrolle) dient.

[51] ,,Technisch zustandsgleich" heißt, daß auch die ausgezeichneten Attribute (zum Beispiel der Materialidentifikator (*MID*)) von Original und Replikat die gleiche Ausprägung besitzen.

6.3.2 Der Materialraumverwalter und der Materialraumversorger

In diesem Unterabschnitt beschäftigen wir uns mit der transparenten Anbindung einer ODMG-93-Datenbank an den Materialraum. Wie wir bereits in Abschnitt 5.5 gesehen haben, besteht durch Einführung eines Caches die Möglichkeit, eine Abstraktion von der vertikalen Speichertrennung zu erreichen.

> „It has recently been pointed out ... that such a uniform memory abstraction in which primary memory acts as a cache of the whole secondary storage is ideally suited for implementing persistent object systems." ([Brössler, Freisleben 89])

Da die *Adaptionsschicht* und damit der Materialraum von der vertikalen Speichertrennung abstrahiert, wollen wir in diesem Unterabschnitt das Konzept eines durch einen *Materialraumverwalter* und einen *Materialraumversorger* bereitgestellten *Materialcaches* beleuchten. Eine direkte Nutzung der Mechanismen eines ODMG-93-Datenbanksystems zur Abstraktion von der vertikalen Speichertrennung ist nicht möglich, da der ODMG-93-Standard

1. neben der Klassenspezifikation kaum Informationen über die dahinterstehenden Konzepte enthält,

2. keine Komponenten-Referenzen bereitstellt,

3. keine Synchronisation nach dem Konzept der Umgangsarten ermöglicht und

4. nicht fachlich generell verwendbar ist und damit nicht direkt durch die Anwendungsschicht genutzt werden kann.

Materialraumverwalter

Der Materialraumverwalter dient wie ein Materialverwalter (vgl. [Wulf, Weske 94] und [Gryczan 95]) zur Beschaffung, Verwahrung und Bereitstellung von Materialobjekten. Seine Aufgabe besteht deshalb in der Verwaltung des Materialcaches, den er mit Hilfe eines *Materialraummagazins* organisiert. Dieses Materialraummagazin ist ähnlich wie das Verwaltermagazin (vgl. [Wulf, Weske 94]) bzw. das Materialmagazin (vgl. [Gryczan 95]) als Behälter modelliert, der alle C++-Referenzen der in der lokalen Umgebung befindlichen Materialobjekte enthält. Der Unterschied zum Verwaltermagazin und dem Materialmagazin besteht darin, daß diese explizit ausgecheckte Materialobjekte enthalten und lediglich eine direkte Abbildung fachlich spezifischer Materialnamen auf C++-Referenzen ermöglichen. Im Gegensatz dazu werden mit Hilfe des Materialraummagazins implizit replizierte Materialrepräsentationen verwaltet. Ausschließlich zum Zweck

des Abgleichs[52] werden Materialobjekte aus dem persistenten Speicher ausgecheckt und anschließend sofort wieder eingecheckt.

Der Materialcache aktualisiert die Materialrepräsentation in der ODMG-93-Datenbank nicht nach jeder Materialmodifikation (*write-through*), sondern erst wenn das Materialobjekt aus dem Materialcache verdrängt wird (*write-back*). Daneben bietet der Materialraumverwalter die Möglichkeit mit Hilfe des Materialraumversorgers, eine von außen angestoßene Aktualisierung eines bestimmten Materialobjekts in der ODMG-93-Datenbank vorzunehmen. Diese Aktualisierung wird explizit durch den Materialraumkoordinator angestoßen, wenn das Materialobjektreplikat, auf das ein Sichtinhaber zugreift, aufgrund von Modifikationen am Materialoriginal zu aktualisieren ist. Dazu wird zunächst der Stand in der ODMG-93-Datenbank aktualisiert (*down-checkpoint*), so daß anschließend durch den Zugriff auf die ODMG-93-Datenbank der Stand des Materialcaches des Sichtinhabers aktualisiert werden kann (*up-checkpoint*).

Das Materialraummagazin enthält zu jedem im Materialcache befindlichen Materialobjekt den zugehörigen Materialidentifikator (*MID*) und die aktuelle physische Adresse innerhalb des Materialcaches (*TID*). Der Materialcache modelliert keine FIFO[53]-Strategie, sondern einen sogenannten LRU[54]-Mechanismus, bei dem stets das Materialobjekt aus dem Materialcache verdrängt wird, dessen letzte Nutzung zeitlich am längsten zurückliegt. Auf diese Weise soll die Wahrscheinlichkeit, daß sich ein Materialobjekt zum Zeitpunkt des Zugriffs bereits im Materialcache befindet, maximiert werden, um auf diese Weise die Zugriffszeit auf Materialobjekte zu minimieren.

Materialraumversorger

Der Materialraumversorger dient zur Beschaffung von Materialobjekten aus und zur Ablage und Auflösung von Materialobjekten in einer angeschlossenen ODMG-93-Datenbank. Angestoßen wird der Materialraumversorger ausschließlich durch den Materialraumverwalter, der sich des Materialraumversorgers bedient, um Materialobjekte, auf die zugegriffen wird und die sich nicht im Materialcache befinden, der zugreifenden Instanz zur Verfügung stellen zu können[55].

---

[52] Es wird entweder der Stand in der Datenbank abgeglichen (*down-checkpoint*) oder der Stand im Materialcache wird aktualisiert (*up-checkpoint*).

[53] **f**irst **i**n - **f**irst **o**ut

[54] **l**east **r**ecently **u**sed

[55] Bei der Spezifikation und prototypischen Realisierung des Frameworks haben wir auf die Betrachtung der Anbindung mehrerer Materialraumversorger verzichtet.

Der Materialraumversorger kann den Materialobjekten „ansehen"[56], welchen statischen Materialtyp sie haben. Dadurch ist es möglich, im Gegensatz zum Materialversorger ([Wulf, Weske 94] und [Gryczan 95]) auf eine Aspektklasse *speicherbar*, die das Protokoll zur Ablage eines Materialobjekts in die Datenbank spezifiziert, zu verzichten. Der Materialraumversorger ist allein durch die Kenntnis des Materialtyps in der Lage, mit Hilfe eines Meta-Objekt-Protokolls alle erforderlichen Daten über das abzulegende, zu beschaffende oder aufzulösende Materialobjekt zu beziehen. Dabei ermöglicht die Modellierung von Materialverknüpfungen und Materialkompositionen auf der Basis von Materialraumreferenzen bzw. Komponentenmaterialraumreferenzen in Kombination mit dem Meta-Objekt-Protokoll den partiellen Umgang mit vernetzten Materialobjektstrukturen. Durch diese Modellierung werden referenzierte Materialobjekte ausschließlich bei Bedarf (*on demand*) geladen[57] und im Gegensatz zu dem in [Wulf, Weske 94] beschriebenen Materialversorgungskonzept nicht automatisch mitgeladen.

Um einen effektiven Umgang mit dem angeschlossenen ODMG-93-Datenbanksystem zu erreichen, verzichten wir bei der Modellierung des Objektzugriffs auf die Nutzung der OQL und realisieren den Zugriff auf Basis des direkten und effizienteren Zugriffs über ODMG-Namen. Zu diesem Zweck bilden wir die Materialidentifikatoren (*MID*) der Materialobjekte direkt auf ODMG-Namen ab.

Im folgenden Abschnitt gehen wir zunächst auf die Realisierung des Meta-Objekt-Protokolls ein. Anschließend werden wir in einem weiteren Abschnitt die Realisierung ausgewählter Klassen der C++-binding des ODMG-93-Standards[58] auf der Basis eines Dateisystems am Beispiel von MS-DOS kurz vorstellen.

6.4 Das Meta-Objekt-Protokoll

Wie wir oben gesehen haben, benötigen wir zur (programmier-)technischen Umsetzung der nachfolgend genannten Dienste der Adaptionsschicht Wissen über die Struktur der Repräsentationen der einzelnen Materialobjekte im transienten Speicher:

- Dynamische Typkonvertierung
- Bereitstellung von Komponentenbeziehungen
- Bereitstellung tiefer fachlich genereller Umgangsformen

[56] *abstraktes_material::get_CID()*

[57] Der Transfer zwischen dem Materialcache und der ODMG-93-Datenbank erfolgt damit objektweise.

[58] eingeschränkt auf die Klassen, die von den Komponenten der Adaptionsschicht genutzt werden.

- Transfer zwischen Materialcache und ODMG-93-Datenbank
- Entkopplung der Materialklassen von technischen Konzepten

Ein Meta-Objekt-Protokoll kann aber auch einen entscheidenden Beitrag zur Realisierung der Abstraktionsschicht leisten. Durch die Abfrage von Meta-Objekt-Wissen kann zum Beispiel die Transformation zwischen einzelnen Paradigmen bzw. Objektrepräsentationen entscheidend vereinfacht werden. Weil C++, anders als einige höhere objektorientierte Programmiersprachen[59], kein Meta-Objekt-Protokoll zur Verfügung stellt, muß das Framework *dinx* ein solches Meta-Objekt-Protokoll zur Verfügung stellen. Dabei ist eine Entscheidung darüber zu treffen, welche Instanz für die Erzeugung bzw. Bereitstellung der Meta-Objekt-Daten zuständig ist und an welchem Ort diese abgelegt werden sollen.

6.4.1 Repräsentation eines C++-Objekts im transienten Speicher

Die Repräsentation von C++-Objekten im transienten Speicher unterliegt keinem in der Literatur festgelegten Standard. Um aber trotzdem Aussagen darüber treffen zu können, wie ein Objekt zur Laufzeit im transienten Speicher repräsentiert ist, haben wir Untersuchungen auf der Basis von Borland C++ (MS-DOS) und dem GNU C++ Compiler (UNIX) durchgeführt. Dieser Unterabschnitt soll die festgestellten Ergebnisse kurz vorstellen.

Ein Objekt, das virtuelle Methoden enthält bzw. virtuell geerbt hat, enthält im transienten Speicher sogenannte versteckte Zeiger (*hidden pointers*), die zum einen auf geerbte Instanzteile (*vbase pointer*) und zum anderen auf die Tabelle der dynamisch gebundenen Methoden (*dispatch table* oder *vtable*) zeigen. Diese versteckten Zeiger sind nur für einen Programmlauf gültig und können deshalb nicht persistent abgelegt werden.

Eingebettet in die Speicherrepräsentation, die sich als ein monolithischer Block im Speicher befindet, gibt es neben den versteckten Zeigern die Repräsentationen der Merkmale des Objekts. Diese Merkmale können zum einen die Werte der Attribute des Objekts oder transiente Speicheradressen referenzierter Objekte sein. Die Abbildung 6.16 zeigt eine solche transiente Speicherrepräsentation anhand eines Exemplars der Klasse D, die Teil der in Abbildung 6.15 dargestellten Klassenhierarchie ist.

[59] Eiffel ist eine höhere objektorientierte Programmiersprache

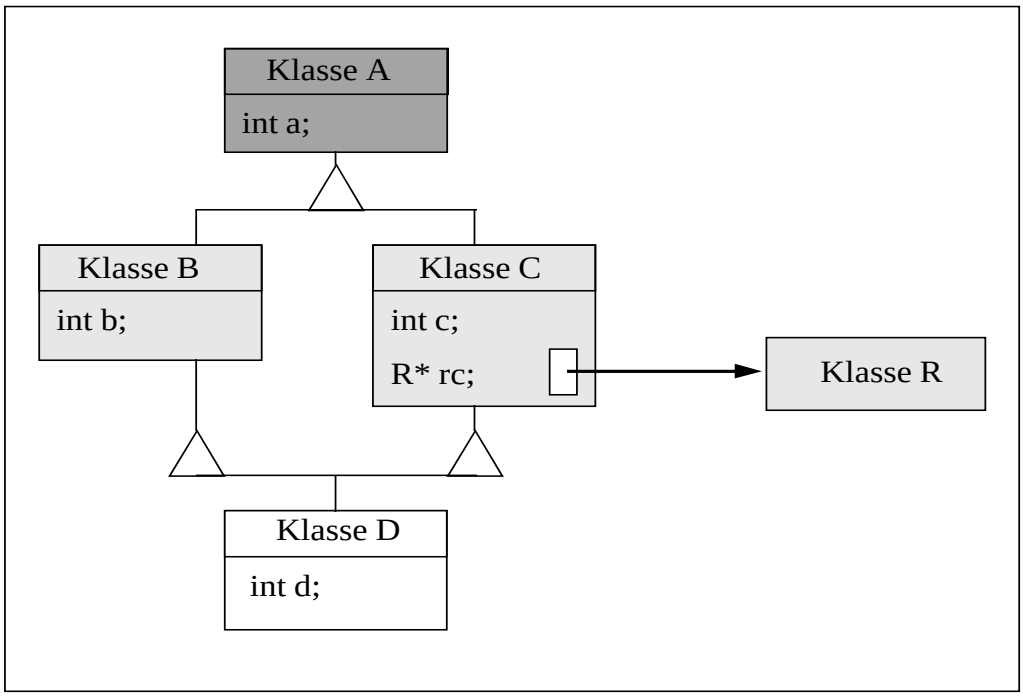


Abbildung 6.15: Klassenhierarchie zur Abbildung 6.16

> ,,Of interest, however, is the handling of object references. Writing them is simple (they just have to be converted into an id), but since they represent objects it must be decided whether to write the full object and not just the reference, and it must be kept track of which object have already be streamed."([Riehle et al. 96])

Durch die Einführung von Materialraumreferenzen, die die C++-Referenzen innerhalb der Speicherrepräsentation ersetzen, wird die Ablage von C++-Objekten in einem persistenten Speicher entscheidend vereinfacht.

Die Abbildung 6.18 zeigt die transiente Speicherrepräsentation eines Exemplars der Klasse D2, die durch Austausch der C++-Referenz *rc* durch eine Materialraumreferenz *rc2* aus der Klasse D hervorgegangen ist (vgl. Abbildung 6.17). Statt des physikalischen Identifikators *A8EF* findet sich nach dem Austausch der Referenzen am Speicherplatz Nr.11 der Materialidentifikator (MID=*56*) des durch die Materialraumreferenz *rc2* referenzierten Materialobjekts. Dieser Materialidentifikator ist unabhängig vom Ort, vom Typ und vom Zustand des referenzierten Materialobjekts.

Durch diese Art der Modellierung ist weder eine Konvertierung von Referenzen in Materialidentifikatoren vorzunehmen noch ist eine Entscheidung darüber zu treffen, wie mit referenzierten Materialobjekten umzugehen ist. Da die Materialraumreferenzen bereits auf der Basis von Materialidentifikatoren gebildet

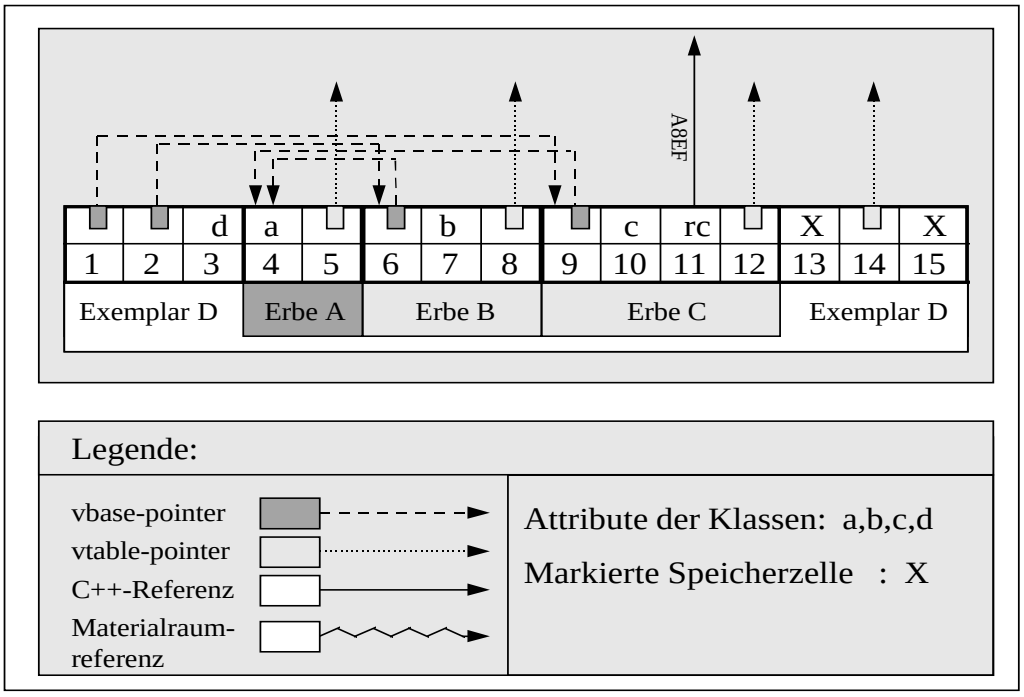


Abbildung 6.16: Speicherpräsentation eines Exemplars der Klasse D

werden und dadurch bei der persistenten Ablage im Gegensatz zu den C++-Referenzen ihre Gültigkeit nicht verlieren, können die Ausprägungen von Materialraumreferenzen eines Materialobjekts wie ,,normale" Attributwerte direkt persistent abgelegt werden. Damit ist auch der partielle Umgang mit vernetzten Materialobjektstrukturen automatisch realisiert.

6.4.2 Erzeugung und Verwahrung des Meta-Objekt-Wissens

Ein Spannungsfeld ergibt sich bei der Frage nach dem Ort der Verwaltung von Meta-Objekt-Daten. Eine Meinung ist, das Wissen nahe an den zugehörigen persistenten Objektklassen zu verwalten, um die Abhängigkeit von diesen zu betonen. Ein anderer Ansatz verlangt das gesamte Wissen gebündelt an einer einzigen Stelle zentral zu organisieren, damit Änderungen am Persistenzmechanismus und dem zugehörigen Protokoll möglichst nur die Veränderung einer zentralen Komponente erfordert und nicht eine Modifikation aller Objektklassen.

Uns erscheint die zweite Sichtweise adäquat, da sie konform zu der von uns vertretenen Argumentation bzgl. der Passivität von Materialien[60] ist. Auch hier sind wir zu dem Schluß gekommen, das teilweise redundant gehaltene Wissen über die Repräsentation eines Materials im persistenten bzw. transienten Speicher aus den

[60] vgl. hierzu die kritische Würdigung der Materialverwaltung in Kapitel 3

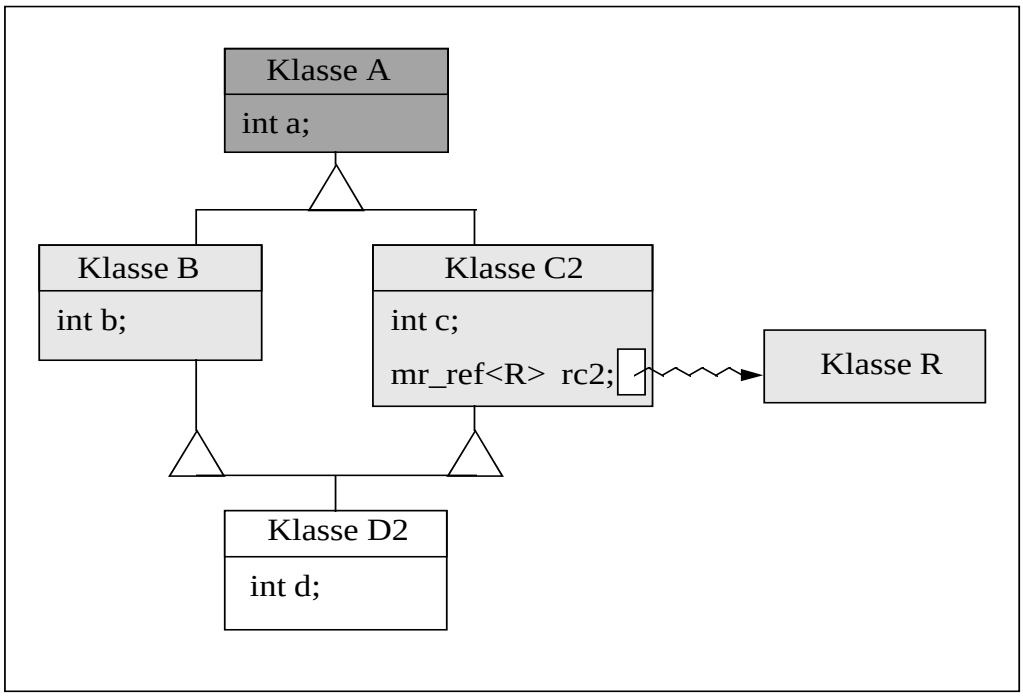


Abbildung 6.17: Klassenhierarchie zur Abbildung 6.18

Materialklassen herauszuziehen und zentral an einer Stelle zu bündeln. Dadurch gelingt es, daß die Schnittstellen der Materialklassen rein fachlich motivierbar sind. Das Meta-Objekt-Wissen wird daher zentral an einer Stelle (Exemplar der Klasse *schema*) organisiert. Zentral heißt in diesem Zusammenhang, daß das Schema orthogonal zu der Abstraktions- und der Adaptionsschicht angeordnet ist, so daß es direkt zur Realisierung dieser herangezogen werden kann.

Eine *Metafabrik* dient zur Erzeugung und zur anschließenden Ablage der Meta-Objekt-Daten im Schema. Das Framework stellt zu diesem Zweck der Anwendungsschicht eine Klasse *abstrakte_metafabrik* zur Verfügung, die durch zusätzliche Methoden für jede fachliche Materialklasse in der Anwendungsschicht zu konkretisieren ist. Da die Metafabrik direkt auf die Struktur der Materialklassen zugreift, widerspricht der zur Konkretisierung erforderliche Implementationscode dem objektorientierten Grundgedanken, die Struktur eines Objekts ausschließlich über bereitgestellte Operationen zu sondieren und zu modifizieren. Aus diesem Grund wäre es sinnvoll, die Metafabrik durch einen Precompiler generieren zu lassen[61], um auf diese Weise eine technisch transparente Erzeugung des Meta-Objekt-Wissens zu erreichen.

[61] Da die Entwicklung eines Precompilers den Rahmen dieser Arbeit sprengen würde, haben wir auf die Realisierung des Precompilers verzichtet.

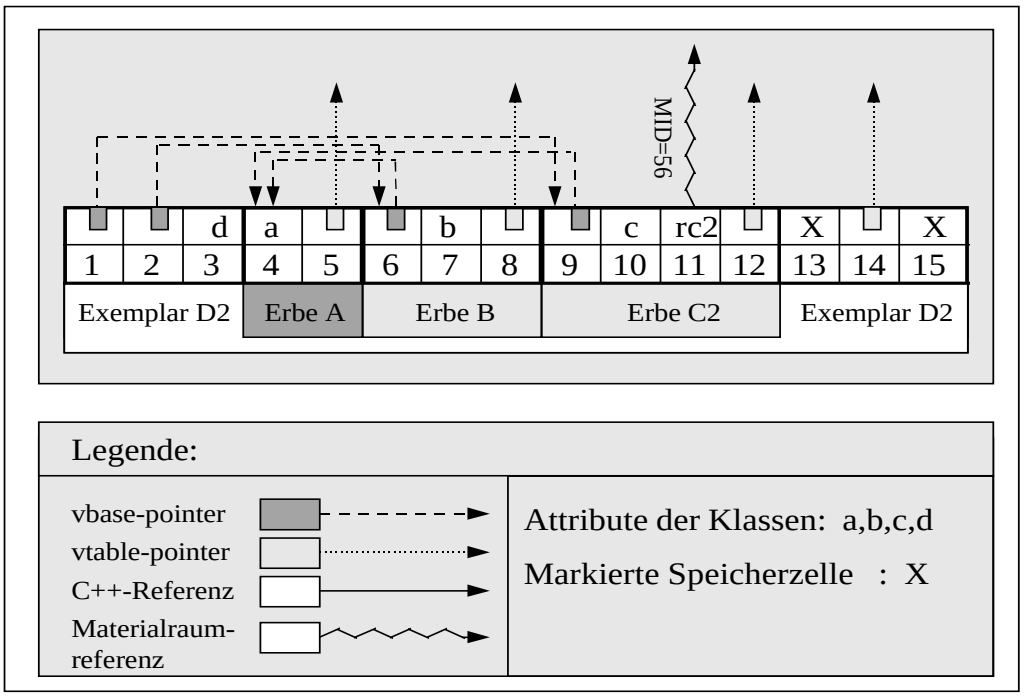


Abbildung 6.18: Speicherrepräsentation eines Exemplars der Klasse D2 unter Nutzung des Frameworks *dinx*

6.4.3 Erzeugung von Materialexemplaren durch eine Fabrik

In Anlehnung an das Entwurfsmuster Abstrakte Fabrik[62] ([Gamma et al. 95]) dient die in der Anwendungsschicht zu konkretisierende abstrakte Fabrik zur Erzeugung von Materialexemplaren eines bestimmten Typs. Die Fabrikklasse *abstrakte_fabrik* stellt zwei Methoden zur Verfügung, die zum einen zur Erzeugung von transienten C++-Objekten (*create(char* _typ)*) und zum anderen zur Erzeugung von ODMG-Objekten (*ODMG_create(char* _typ)*) dienen[63]. Im Gegensatz zum Entwurfsmuster Abstrakte Fabrik vereinen wir die Menge aller Erzeugungsfunktionen spezifischer Typen (*ErzeugeProduktX()*) in einer einzigen Methode (*create(char* _typ)* und *ODMG_create(char* _typ)*). Diese Bündelung erleichtert das Hinzufügen weiterer Typen, weil die Schnittstelle der Fabrik erhalten bleibt und nicht erweitert werden muß. Der Nutzer des Frameworks muß in einer Unterklasse der abstrakten Klasse *abstrakte_fabrik* die beiden Methoden

- *create(char* _typ)* und
- *ODMG_create(char* _typ)*

[62] Zweck des Abstrakte-Fabrik-Musters: „Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen." ([Gamma et al. 95])

[63] Es wird jeweils ein Objekt des übergebenen Typs erzeugt und zurückgegeben.

durch Hinzufügen des Codes zur Erzeugung von Objekten der fachlich spezifischen Typen konkretisieren.

Da die Konkretisierung der Klassen *abstrakte_metafabrik* und *abstrakte_fabrik* von jeder Änderung an der Spezifikation der Klassen der fachlichen Welt abhängig ist, wäre es sinnvoll, die Konkretisierung beider Klassen an einen Precompiler zu deligieren. Dadurch können Fehler bei der prinzipiell redundanten Implementation der Fabriken verhindert und der Implementator von einer stupiden Arbeit befreit werden. Da die Entwicklung eines Precompilers den Rahmen dieser Arbeit sprengen würde, haben wir uns zur Simplifizierung der Programmierung und damit zur Reduzierung des Implementationsaufwands für den Einsatz von *Makros*, die einen Beitrag zur Vereinfachung der Programmierung leisten, als interimistische Lösung entschieden.

6.5 Ein ODMG-93-Datenbanksystem auf Basis eines beliebigen persistenten Speichers

In diesem Abschnitt stellen wir die Erweiterung eines Dateisystems zu einem rudimentären ODMG-93-Datenbanksystem am Beispiel des Dateisystems MS-DOS kurz dar. Die prototypische Realisierung eines ODMG-93-Datenbanksystems auf der Basis des Dateisystems MS-DOS unterstützt die im Rahmen der Erprobung unseres Entwurfs erforderlichen Klassen (*Database*, *Ref_Any*, *Ref<T>* und *Persistent_Object*). Auf eine Implementation weiterer Klassen (zum Beispiel *Bag<T>*) mußte verzichtet werden, da die Entwicklung eines ODMG-93-Datenbanksystems nicht die Aufgabe dieser Arbeit ist.

Im Hinblick auf die ODMG-93-Konformität des prototypisch realisierten ODMG-93-Adapters ist festzuhalten, daß ODMG-Namen in der prototypischen Umsetzung lediglich auf der Basis von ganzzahligen Werten (*int*) realisiert worden sind, obwohl sie nach dem ODMG-93-Standard durch alphanumerische C-Strings umzusetzen sind. Die Realisierbarkeit des hier vorgestellten Entwurfs wird aus unserer Sicht dadurch nicht eingeschränkt, da auch die oben beschriebenen Materialidentifikatoren (*MID*) als ganzzahlige Werte realisiert sind und direkt auf die ODMG-Namen abgebildet werden.

Analog zu einigen Klassen der Adaptionsschicht (zum Beispiel die Klasse *materialraum*) fungieren die ODMG-93-Klassen *Database*, *Ref_Any*, *Ref<T>* und *Persistent_Object* als Fassade des ODMG-93-Datenbanksystems (vgl. Abbildung 6.19). Damit alle Materialklassen automatisch Unterklassen der persistenten Mixin-

Klasse[64] *Persistent_Object* sind, ordnen wir die Klasse in der Vererbungshierarchie über der Klasse *abstraktes_material* an, so daß zum einen die persistente Mixin-Klasse *Persistent_Object* der Anwendungsschicht verborgen bleibt und zum anderen jedes Materialobjekt a priori persistent ist[65].

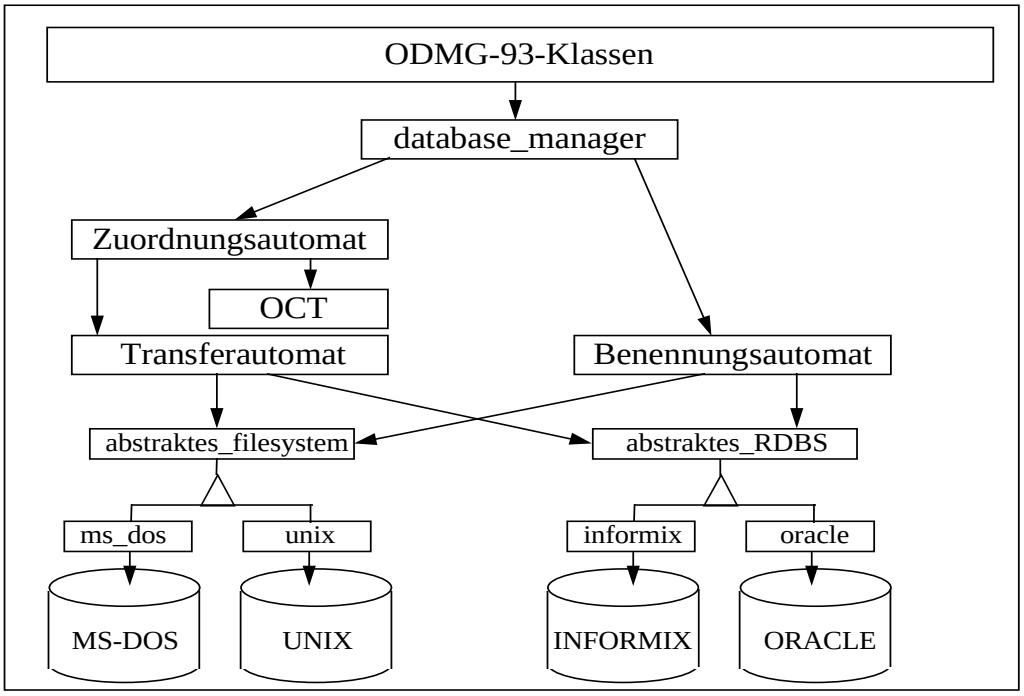


Abbildung 6.19: Klassenentwurf zur Adaptierung eines persistenten Speichers zu einem ODMG-93-Datenbanksystem

Die Exemplare der Klassen *Database*, *Ref_Any*, *Ref<T>* und *Persistent_Object* deligieren die an sie gestellten Anfragen an ein Exemplar der Klasse *database_manager*, das die einzelnen Anfragen mit Hilfe eines *Benennungsautomaten*, der u.a. zur Vergabe der Objektidentifikatoren (OID), sowie eines *Zuordnungsautomaten*, der mit Hilfe eines *Transferautomaten* einen Objektcache realisiert, erledigt.

Die Klassen *abstraktes_filesystem* und *abstraktes_RDBS* dienen als unterer Abschluß des in diesem Kapitel vorgestellten Frameworks. Beide Klassen abstrahieren von den Spezifika eines konkreten persistenten Speichers und stellen ausschließlich die für das Paradigma des persistenten Speichers (Dateisystem oder relationales Datenbanksystem) charakteristische Funktionalität über eine Schnitt-

[64]Nach [Booch 94] ,,ist der gebräuchlichste Ansatz zur Realisierung der Persistenz die Verwendung einer persistenten Mixin-Klasse. Alle Objekte, für die eine persistente Semantik wünschenswert ist, besitzen diese Mixin-Klasse irgendwo in ihrer Vererbungslinie als Oberklasse."

[65]vgl. die technische Interpretation der Lebensdauer eines Materials

stelle bereit (zum Beispiel *create_file()* oder *create_relation()*). Ein konkreter Speicher wird in einem Objekt gekapselt, das Exemplar einer Unterklasse einer dieser abstrakten Klassen ist. Ausschließlich in einer solchen Unterklasse (zum Beispiel *ms_dos* oder *oracle*) wird auf die durch den persistenten Speicher konkret bereitgestellte Funktionalität direkt zugegriffen.

Nachfolgend beschreiben wir die in der Abbildung 6.19 dargestellten Klassen *Benennungsautomat*, *Zuordnungsautomat*, *Transferautomat* sowie die Klassen *abstraktes_RDBS* und *abstraktes_filesystem*.

Die Klasse Benennungsautomat

Analog zum oben beschriebenen Materialidentitor dient der Benennungsautomat zur Vergabe von Objektidentifikatoren (OID). Um die Eindeutigkeit der Objektidentifikatoren (OID) sicherzustellen, erfolgt die Zuordnung von Objektidentifikatoren (OID) während der Erzeugung eines Objekts mit Hilfe eines Exemplars der Klasse *Benennungsautomat*. Dieses Objekt verwaltet einen *OID-Zähler*, der fortlaufend hochgezählt wird. Auch der Zugriff auf diesen OID-Zähler ist analog zum Umgang mit dem MID-Zähler in einem verteilten System als kritischer Abschnitt zu modellieren, der beim Anschluß eines Datenbanksystems mit Hilfe des durch das Datenbanksystem bereitgestellten Sperr- bzw. Transaktionsmechanismus realisiert werden kann[66]. Auf diese Weise kann sichergestellt werden, daß keine Objektidentifikatoren (OID) mehr als einmal vergeben werden.

Daneben erhält der Benennungsautomat die Aufgabe, den ODMG-93-Namenservice bereitzustellen, d.h. die Vergabe von Objektnamen zu koordinieren und vergebene Objektnamen anschließend zu verwalten und persistent abzulegen. Damit auch die Eindeutigkeit dieser Objektnamen sichergestellt werden kann, ist der Umgang mit den in einem verteilten System global bekannten Objektnamen analog zum konkurrierenden Zugriff auf den OID-Zähler adäquat zu modellieren.

Die Klasse Zuordnungsautomat

Der Zuordnungsautomat dient zur Beschaffung, Verwahrung und Bereitstellung von Objekten. Seine primäre Aufgabe besteht deshalb in der Verwaltung eines *Objektcaches*, den er mit Hilfe einer *OCT*[67] organisiert. Dazu enthält die OCT zu jedem im Objektcache befindlichen Objekt den zugehörigen Objektidentifikator (OID) und die aktuelle physikalische Adresse innerhalb des Objektcaches (TID). Der Objektcache ist als ein sogenannter *Rückschreibe-Cache* (*write-back*) organisiert und modelliert wie der oben beschriebene Materialcache einen LRU-Mechanismus, so daß stets das Objekt aus dem Objektcache verdrängt wird, das

[66] Beim Einsatz eines Dateisystems ist die Realisierung schwieriger.

[67] **o**bject **c**ache **t**able

zeitlich am längsten nicht referenziert worden ist (LRU[68]-Objekt).

Damit auf ein Objekt, das sich bislang nicht im Objektcache befindet, zugegriffen werden kann, wird zunächst das LRU-Objekt aus dem Cache verdrängt. Anschließend bedient sich der Zuordnungsautomat eines Transferautomaten, mit dessen Hilfe das gewünschte Objekt aus dem persistenten in den transienten Speicher transferiert wird. Abschließend wird das Objekt in die OCT eingetragen und es kann auf die transiente Repräsentation des Objekts zugegriffen werden.

Die Klasse Transferautomat

Der Transferautomat dient zur Erzeugung, Ablage, Beschaffung und Vernichtung persistenter Objektrepräsentationen in einem angeschlossenen persistenten Speicher und abstrahiert dabei vom Paradigma des angeschlossenen persistenten Speichers. Angestoßen wird der Transferautomat ausschließlich durch den Zuordnungsautomaten.

> „The Object-Converters could in principle access the relational database directly. However, we put a Database Interface Layer in between due to portability; each relational DBS has its own call level interface. Exchanging the underlying relational DBS thus requires few modifications in only this layer." ([Hohenstein 96])

In Anlehnung an [Hohenstein 96] modellieren wir die Klasse *Transferautomat* unabhängig von dem konkreten persistenten Speicher, in den und aus dem der Automat Objekte transferiert. Parametrisiert wird der Transferautomat mit einem Objekt, das die Spezifika eines konkreten persistenten Speichers (zum Beispiel MS-DOS) kapselt, aber die Funktionalität dieses Speichers über eine abstrakte, für das Paradigma des Speichers charakteristische Schnittstelle (zum Beispiel *create_file()*) zur Verfügung stellt[69]. Damit hat der Transferautomat zwar Kenntnis über das Paradigma des angeschlossenen persistenten Speichers, ist aber gleichzeitig unabhängig von den Spezifika des konkreten persistenten Speichers. Je nach Paradigma dieses Speichers bildet der Transferautomat Objekte entweder auf Dateien oder Relationen ab.

Bei der Transformation der Objekte in Relationen oder Dateien benutzt der Transferautomat ein Exemplar der Klasse *schema*, das die nötigen Informationen zur Fragmentierung bereitstellt. Da der Transferautomat den Objekten ansehen[70] kann, welchen statischen Typ sie besitzen, ist er in der Lage, mit Hilfe des Schemas alle erforderlichen Daten über das zu erzeugende, abzulegende oder zu löschende

[68] **l**east **r**ecently **u**sed

[69] Die Klasse *Transferautomat* nutzt nur die Schnittstelle der abstrakten Klassen *abstraktes_filesystem* bzw. *abstraktes_RDBS*.

[70] *Persistent_Object::get_type_ID()*

Objekt zu beziehen, um auf die einzelnen Attribute zugreifen und diese attributweise im persistenten Speicher ablegen oder löschen zu können. Bei der Beschaffung persistent abgelegter Objekte kommt wiederum das Schema zum Einsatz. Zusätzlich nutzt der Transferautomat die Fabrik, mit deren Hilfe er adäquate Objekte erzeugt, um sie anschließend attributweise mit aus dem persistenten Speicher ausgelesenen Werten zu belegen.

Die Klasse abstraktes_RDBS und die Klasse abstraktes_filesystem

Wie Uwe Hohenstein ([Hohenstein 96]) ist auch Grady Booch ([Booch 94]) der Auffassung, daß es vernünftig ist, eine objektorientierte Hülle über ein relationales Datenbanksystem zu legen. Durch die objektorientierte Kapselung der Funktionalität eines persistenten Speichers gelingt die lose Kopplung des Frameworks mit dem Frontend eines konkreten persistenten Speichers. Darüber hinaus können alle Komponenten bis auf das Objekt, das das Datenbanksystem kapselt, frei von datenbankspezifischen Implementationscode gehalten werden.

In Anlehnung an das Entwurfsmuster Brücke[71] ([Gamma et al. 95]) stellt das Framework zwei unterschiedliche abstrakte Klassen *abstraktes_RDBS* und *abstraktes_filesystem* zur Verfügung. Mit Hilfe dieser Klassen gelingt es, die Schnittstelle von der konkreten Implementation zu entkoppeln. Da dadurch die Implementationsdetails vor dem Transferautomaten verborgen werden können, ist dieser unabhängig von dem konkret angeschlossenen persistenten Speicher, so daß das Framework leicht durch andere konkrete persistente Speicher erweitert werden kann.

Für die persistente Repräsentation von Objekten in einem Dateisystem gibt es prinzipiell zwei unterschiedliche Ansätze:

- *Fragmentierte Repräsentation:*
  Der Zustand und damit die Ausprägungen der Attribute des Objekts werden attributweise in einer Datei abgelegt.

- *Nicht fragmentierte Repräsentation*:
  Die Repräsentation im transienten Speicher wird byteweise als BLOB[72] in den persistenten Speicher übertragen.

Wie wir bereits bei der Darstellung des Transferautomaten festgestellt haben, favorisieren wir den attributweisen Datenaustausch mit dem angeschlossenen persistenten Speicher. Zu diesem Zweck bietet die Schnittstelle der Klasse *abstrak-*

[71] Zweck des Brückenmusters: „Entkopple eine Abstraktion von ihrer Implementierung, so daß beide unabhängig voneinander variiert werden können." ([Gamma et al. 95])

[72] **b**inary **l**arge **o**bject

tes_filesystem[73] zu jedem Basistyp (zum Beispiel *int*, *float* und *char*) eine Lese- und eine Schreiboperation (zum Beispiel *read_int()* und *write_int()*).

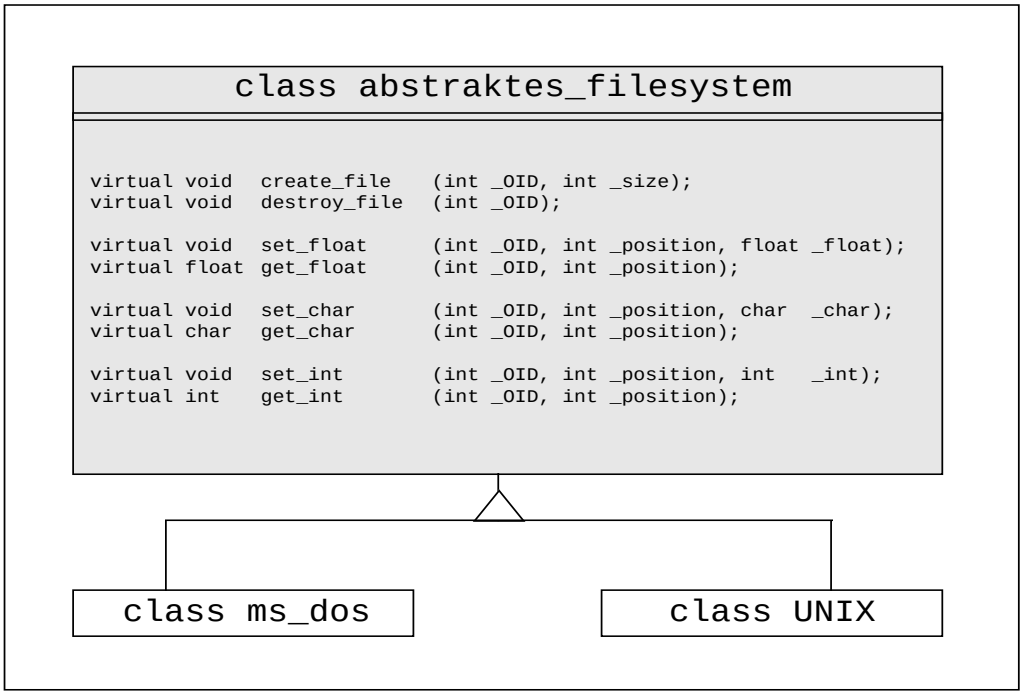


Abbildung 6.20: Schnittstelle der Klasse *abstraktes_filesystem*

6.6 Umgang mit dem Framework

Im Anschluß an die Vorstellung der statischen Sicht auf das Framework soll in diesem Abschnitt das dynamische Verhalten des Frameworks und der Umgang mit dem Framework auf der Basis ausgewählter Szenarien und mit Hilfe von Interaktionsdiagrammen dargelegt und visualisiert werden. Dabei legen wir insbesondere Wert auf die Perspektive eines Implementators, der das Framework zur Programmierung fachlich spezifischer Anwendungen einsetzt. Unser Hauptaugenmerk liegt daher auf dem intuitiven (programmier-)technischen Umgang mit Materialien im Materialraum. In diesem Zusammenhang spielt insbesondere der Materialraumreferenz-Mechanismus, der den transparenten Umgang mit den technischen Komponenten (zum Beispiel dem angeschlossenen persistenten Speicher) ermöglicht, eine wichtige Rolle.

6.6.1 Umgang mit dem Materialraum

In diesem Unterabschnitt wollen wir uns zunächst dem Umgang mit dem Materialraum und den Materialraumreferenzen bei der (programmier-)technischen

[73] vgl. Abbildung 6.20

Umsetzung fachlich spezifischer Konzepte widmen. Die Darstellung erfolgt anhand eines kleinen Beispiels, das aus dem Bereich der auf dem Framework *dinx* basierenden, prototypisch entwickelten Anwendung einer Autowerkstatt stammt.

In unser stark simplifizierten Sicht auf diese Werkstatt gibt es in der zugehörigen fachlichen Domäne Automobile, Personen und Räder. Die Automobile lassen sich in Sportwagen und Lieferwagen unterscheiden, wobei jedes Automobil mit einer Person, die Fahrer des jeweiligen Automobils ist, verknüpft sein kann. Darüber hinaus kann jedes Automobil aus vier Rädern bestehen (Materialkomposition), so daß ein Automobil ein komplexes Material darstellt und damit auch als Ganzes bearbeitet werden kann.

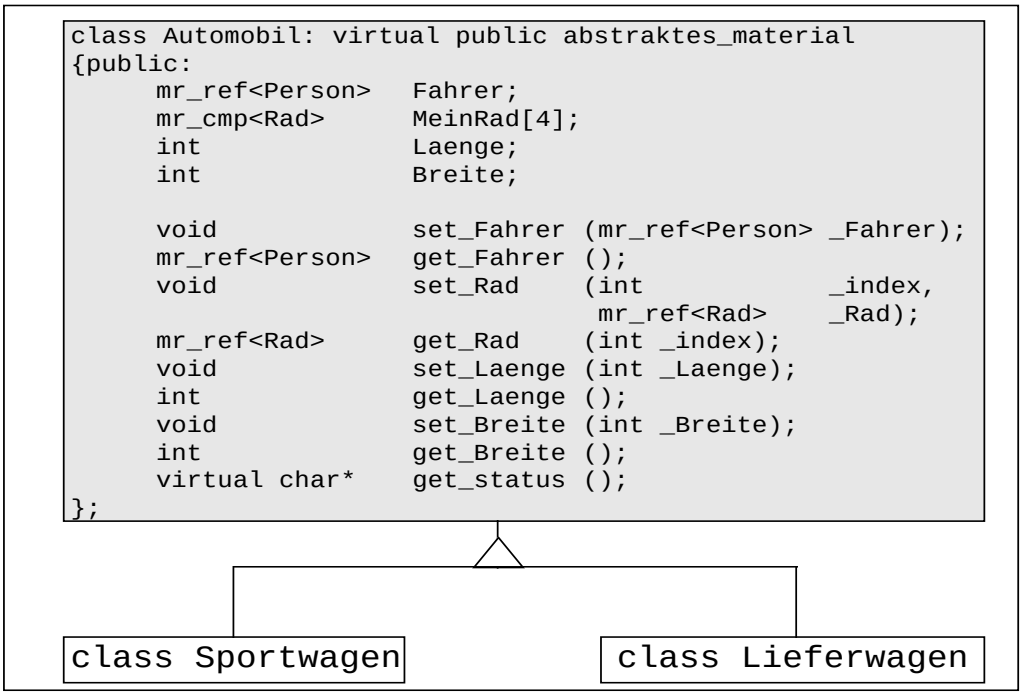


Abbildung 6.21: Materialklasse *Automobil*

Ein Exemplar der Kontextmaterialklasse *Automobil*[74] enthält zwei Attribute, die die Länge (*Laenge*) und die Breite (*Breite*) des Automobils darstellen. Neben der getypten Materialraumreferenz, die die Bezugnahme auf einen Fahrer (*mr_ref<Person> Fahrer*) erlaubt, hat jedes Exemplar dieser Materialklasse vier Komponentenmaterialraumreferenzen (*mr_cmp<Rad> MeinRad[4]*) auf die einzelnen Räder des Automobils.

Ein Materialobjekt vom Typ *Automobil* stellt zu jedem Attribut und jeder (Komponenten-)Materialraumreferenz je eine sondierende und eine modifizierende Operation zur Verfügung (zum Beispiel *get_Fahrer()* und *set_Fahrer()*). Daneben kann

[74] vgl. Abbildung 6.21

mit Hilfe der virtuellen Operation *get_status()*, die charakteristische Qualität des Automobils abgefragt werden (zum Beispiel ist ein Sportwagen *"sehr schnell"* und der Lieferwagen *"kann viel transportieren"*).

Neben der Konkretisierung der virtuellen Operation *get_status()* bietet jede Unterklasse der Klasse *Automobil* ein zusätzliches Attribut (*Geschwindigkeit* bzw. *Ladekapazität*), das durch adäquate Operationen sowohl sondiert als auch modifiziert werden kann[75].

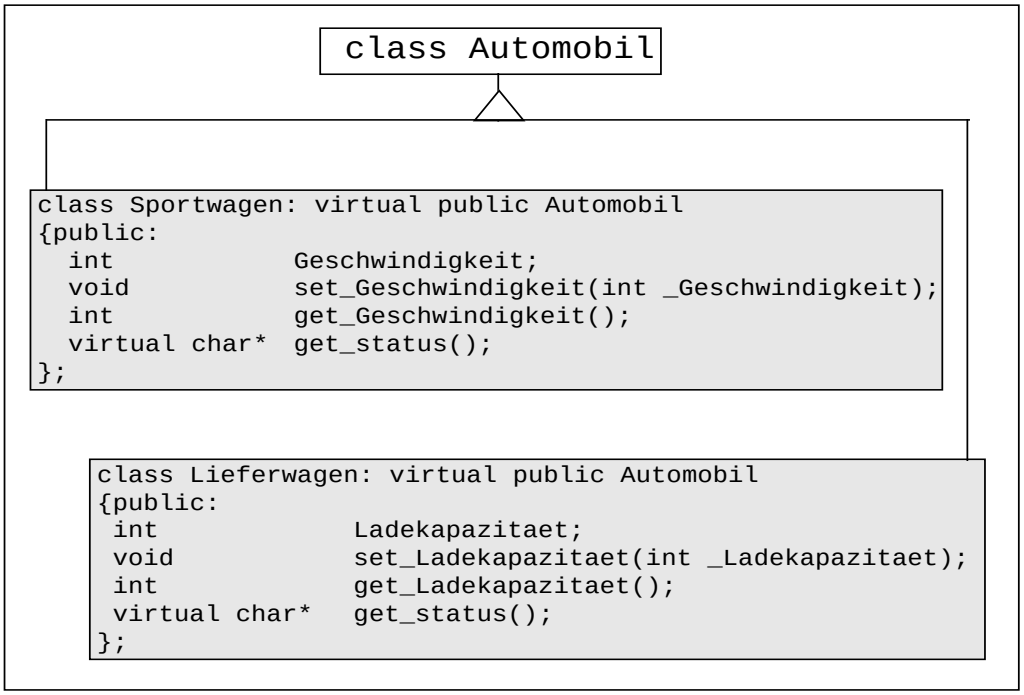


Abbildung 6.22: Materialklassen *Sportwagen* und *Lieferwagen*

Bevor wir das dynamische Verhalten des Frameworks anhand von drei ausgewählten Szenarien (Materialerzeugung, Materialzugriff und Materialauflösung) darstellen, wollen wir anhand eines kleinen Beispielprogramms (vgl. Abbildung 6.23) einen Einblick in den transparenten Umgang mit dem Framework *dinx* geben.

In dem in Abbildung 6.23 dargestellten Codefragment werden vier getypte Materialraumreferenzen genutzt. Während zwei Referenzen (*MeinPorsche* und *MeinFerrari*) auf Sportwagen verweisen, referenziert eine dritte (*Ein_Auto*) ein Automobil und eine vierte (*Eine_Person*) eine Person. Nach der Deklaration dieser Materialraumreferenzen wird ein Sportwagen mit dem Namen *"911 Turbo"* tief, d.h mit Rädern erzeugt und an die Materialraumreferenz *MeinPorsche* gebunden.

[75] vgl. Abbildung 6.22

```
void Beispiel()
{mr_ref<Sportwagen>      MeinPorsche;
 mr_ref<Sportwagen>      MeinFerrari;
 mr_ref<Automobil>       Ein_Auto;
 mr_ref<Person>          Eine_Person;

 MeinPorsche.create("911 Turbo", DEEP);
 MeinFerrari=mein_materialraum->lookup_material("368 GTS");
 Eine_Person=mein_materialraum->lookup_material("Herr_Hoppenstedt");
 //-------------------------------------------------------------
 if   (MeinPorsche!=MeinFerrari)
      {cout << "Der Ferrari und der Porsche sind nicht identisch!";};
 //-------------------------------------------------------------
 Ein_Auto=MeinFerrari.clone(DEEP);
 mein_materialraum.set_material_name(Ein_Auto,"Ein geklonter 368 GTS");
 //-------------------------------------------------------------
 if   (Ein_Auto.compare(MeinFerrari, DEEP)==TRUE)
      {cout << "Der Ferrari und der geklonte Ferrari sind gleich!\n";
      };
 //-------------------------------------------------------------
 cout << Ein_Auto->get_status();
 Ein_Auto->set_Fahrer(Eine_Person);
 //-------------------------------------------------------------
 Ein_Auto.destroy(DEEP);
 MeinFerrari.destroy(SHALLOW);
};
```

Abbildung 6.23: Beispielprogramm

Anschließend wird an die Materialraumreferenz *MeinFerrari* der Sportwagen mit dem Materialnamen *"368 GTS"*, der sich bereits im Materialraum befindet, gebunden. Auf die gleiche Weise wird die Materialraumreferenz *Eine_Person* an die bereits existente Person *"Herr_Hoppenstedt"* gekoppelt.

Darauf folgend werden die durch die beiden Materialraumreferenzen (*MeinPorsche* und *MeinFerrari*) referenzierten Sportwagen auf eine nicht bestehende Materialidentitätsgleichheit geprüft. Da beide Materialien nicht identisch sind, wird der C-String *"Der Ferrari und der Porsche sind nicht identisch!"* auf dem Bildschirm ausgegeben und der Ferrari wird in einem nächsten Schritt tief, d.h. mit seinen Rädern kopiert. Das dabei erzeugte Sportwagen-Duplikat kann anschließend an die Materialraumreferenz *Ein_Auto* gebunden werden, weil die Klasse *Automobil* Oberklasse der Klasse *Sportwagen* und damit die Bindung eines Sportwagens an eine Materialraumreferenz, die auf ein Automobil verweist, möglich ist. Anschließend wird dem geklonten Sportwagen ein im Materialraum eindeutiger Materialname zugeordnet (*"Ein geklonter 368 GTS"*).

Da die Sportwagen mit dem Materialnamen *"368 GTS"* und *"Ein geklonter 368 GTS"* tief zustandsgleich sind, ergibt die Überprüfung dieser Eigenschaft *TRUE* und es wird die Meldung *"Der Ferrari und der geklonte Ferrari sind gleich!"* auf dem Bildschirm ausgegeben. Anschließend kann mit Hilfe der virtuellen Operation *get_status()* die charakteristische Eigenschaft des geklonten Materials (*"ist schnell"*) ermittelt und ausgegeben werden.

Bevor der geklonte Ferrari tief, d.h. mit seinen Rädern (*Ein_Auto.destroy(DEEP)*) und der Ferrari mit dem Materialnamen *"368 GTS"* flach, d.h. ohne seine Räder (*Ein_Ferrari.destroy(SHALLOW)*) aufgelöst wird, ordnen wir dem geklonten Ferrari noch einen Fahrer zu (*Ein_Auto->set_Fahrer(Eine_Person)*), die durch den Vorgang der anschließenden Materialauflösung des geklonten Ferrari nicht betroffen ist und auch nach der Auflösung bestehen bleibt, weil sie mit dem Automobil lediglich verknüpft ist und nicht Teil (Komponentenmaterial) des Automobils ist.

Im folgenden wollen wir anhand von drei Interaktionsdiagrammen die durch die Ausführung der Operationen[76]

- *MeinPorsche.create("911 Turbo", DEEP);*
- *Ein_Auto->get_status();* und
- *Ein_Auto.destroy(DEEP);*

transparent angestoßenen Vorgänge innerhalb des Frameworks[77] visualisieren.

Szenario: Erzeugung eines Materials

Wie alle fachlich generellen Umgangsformen ist auch die Materialerzeugung als Operation der Materialraumreferenzen *mr_ref_any*, *mr_ref<T>* und *mr_cmp<T>* realisiert. Um diese Klassen möglichst kompakt zu halten, delegieren die Materialraumreferenzen die Ausführung der Operationen, die fachlich generelle Umgangsformen widerspiegeln, an den Materialraum. Somit führt der Aufruf der Operation *MeinPorsche.create("911 Turbo", DEEP)* unmittelbar zur Ausführung der Operation *create_material("Sportwagen","911 Turbo",DEEP)* des Materialraums (vgl. Abbildung 6.24). Es wird zunächst mit Hilfe des Schemas der zum Typ bzw. zur Klasse *Sportwagen* zugehörige Materialtypidentifikator (*CID*) ermittelt. Anschließend wird die Operation *get_next_MID()* des Materialraumidentitors angestoßen, die als Rückgabewert einen neuen, bislang nicht vergebenen Materialidentifikator (*MID*) liefert.

Durch Nutzung des Materialraumversorgers kann eine persistente und eine transiente Repräsentation des Materials erzeugt werden (*create_material(MID, CID)*). Beiden Repräsentationen wird sowohl der ermittelte Materialidentifikator (*MID*)

[76] Alle Operationen, die fachlich spezifische Umgangsformen widerspiegeln, werden durch *mr_ref<T>->Operation_X* aufgerufen und alle Operationen, die fachlich generelle Umgangsformen realisieren, werden durch *mr_ref<T>.Operation_Y* angestoßen (vgl. die Ausführungen in Unterabschnitt 6.2.5).

[77] eingeschränkt auf die Adaptionsschicht

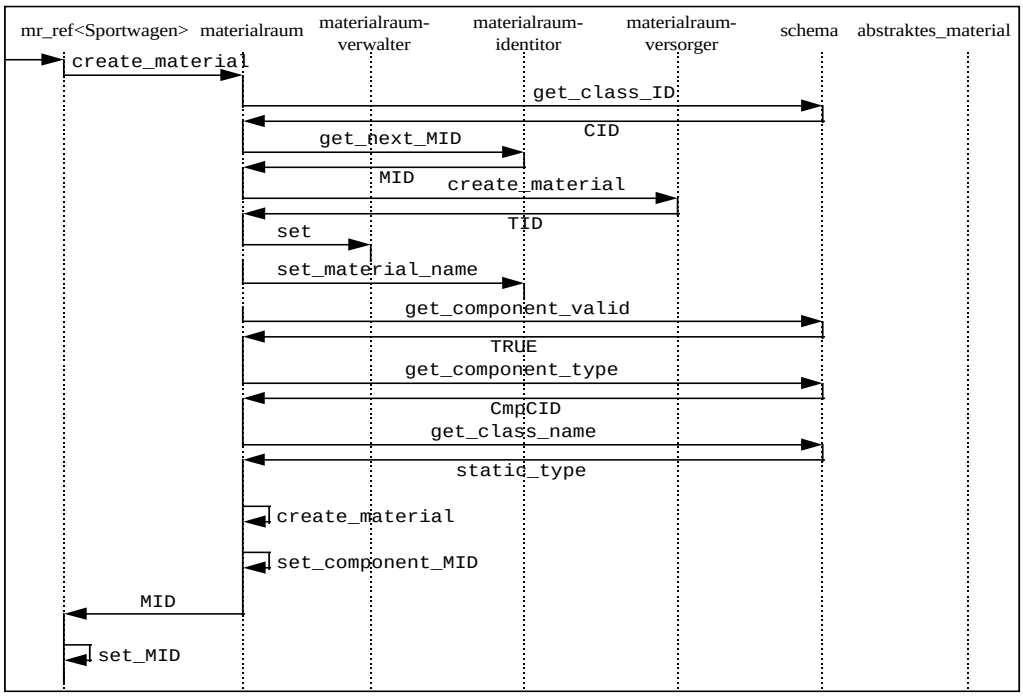


Abbildung 6.24: Interaktionsdiagramm *MeinPorsche.create("911 Turbo",DEEP)*

als auch der zugehörige Materialtypidentifikator (*CID*) zugeordnet. Die Operation *create_material(MID, CID)* des Materialraumversorgers liefert als Rückgabewert eine C++-Referenz auf die transiente Materialobjektrepräsentation (*TID*), so daß diese zusammen mit dem Materialidentifikator (*MID*) anschließend dem Materialraumverwalter bekannt gemacht werden kann (*set(MID,TID)*). Der Materialraumverwalter sorgt für die Eintragung des Materialobjekts in das durch ihn verwaltete Materialraummagazin, so daß bis zum Zeitpunkt der Verdrängung des Materialobjekts aus dem transienten Speicher (Materialcache) auf diese transiente Repräsentation des Materials zugegriffen werden kann.

Nachdem das Materialobjekt im Materialraummagazin eingetragen worden ist, wird dem neu erzeugten Material der Materialname *"911 Turbo"* zugeordnet (*set_material_name(MID,"911 Turbo")*). Da der Sportwagen tief erzeugt werden soll, ist anschließend zu ermitteln, ob die Klasse *Sportwagen* Komponentenmaterialraumreferenzen enthält (*get_component_valid(CID,...)*). Ist dieses der Fall werden nachfolgend die Typen (*get_component_type(CID,...)*) und die Klassennamen (*get_component_class_name(CmpCID)*) der Komponentenmaterialobjekte sukzessiv ermittelt, so daß mit Hilfe der *create_material("Rad","",DEEP)*-Operation des Materialraums adäquate Komponentenmaterialobjekte tief erzeugt werden können, deren Materialidentifikatoren (*CmpMID*) den Komponentenmaterialraumreferenzen des Kontextmaterialobjekts (*Sportwagen::MeinRad[x]*) zu-

geordnet werden[78] (*set_component_MID(MID,CmpMID)*). Als Rückgabewert liefert der Aufruf der Operation *create_material("Sportwagen", "911 Turbo",DEEP)* den Materialidentifikator (*MID*) des erzeugten Sportwagens, der abschließend an die Materialraumreferenz (*MeinPorsche*) gebunden wird (*set_MID(MID)*).

Szenario: Zugriff auf ein Material

Die Ausführung einer Operation, die eine fachlich spezifische Umgangsform realisiert, erfolgt in zwei Phasen. Während in einer ersten Phase geprüft wird, ob sich ein Materialobjekt im transienten Speicher befindet und gegebenenfalls aus dem persistenten Speicher aktiviert wird, dient eine zweite Phase zur Ausführung der eigentlichen Operation auf dem (gecachten) transienten Materialobjekt. Im folgenden Beispiel gehen wir davon aus, daß sich keine Repräsentation des Automobils, dessen charakteristische Eigenschaft wir abfragen wollen (*get_status()*), im transienten Speicher und damit im Materialcache befindet.

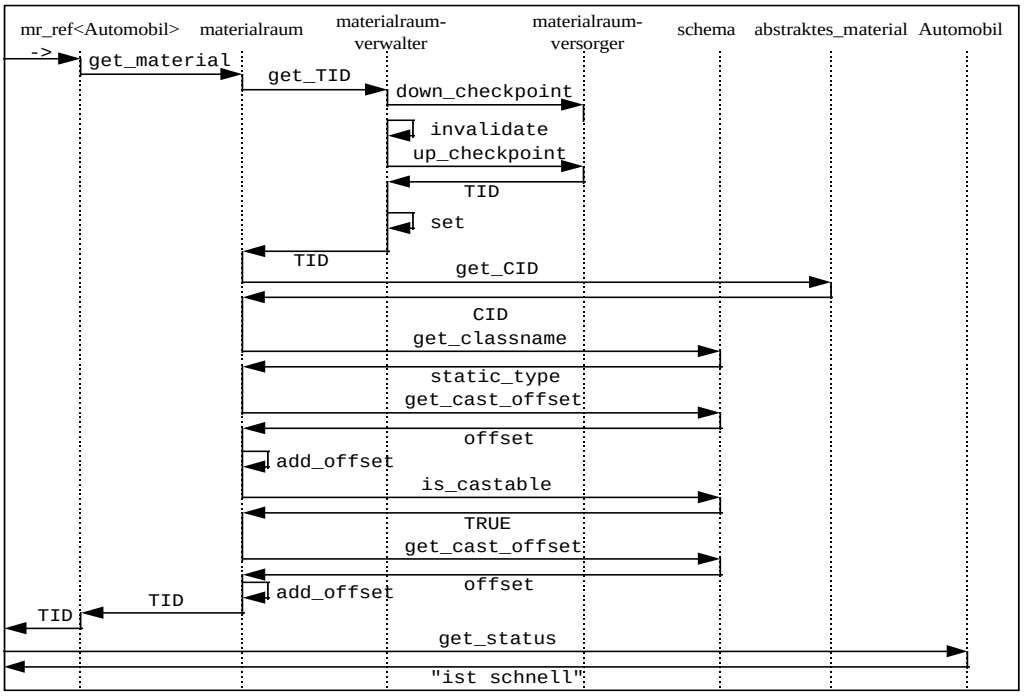


Abbildung 6.25: Interaktionsdiagramm *Ein_Auto->get_status()*

Die Dereferenzierung (->) einer getypten Materialraumreferenz (*Ein_Auto*) führt unmittelbar zum Aufruf der Operation *get_material(MID, "Automobil")* des Materialraums, die zur Ermittlung einer C++-Referenz auf eine transiente Repräsentation eines bestimmten Materials (*MID*) dient. Dabei ist die Referenz bzgl. eines

[78] In dem in Abbildung 6.24 dargestellten Interaktionsdiagramm ist lediglich die Erzeugung eines Rades visualisiert, obwohl insgesamt vier Räder erzeugt und anschließend an die zugehörigen Komponentenmaterialraumreferenzen des Kontextmaterialobjekts gebunden werden.

gewünschten dynamischen Typs - in dem hier betrachteten Fall *Automobil* - bereitzustellen.

Um dieses Ziel zu erreichen wird zunächst mit Hilfe des Materialraumverwalters, der zur Verwaltung des lokalen Materialcaches dient, eine nach *abstraktes_material*-gecastete C++-Referenz auf das gewünschte Materialobjekt ermittelt (*get_TID(MID)*). Da der Materialcache - wie oben festgelegt - das gewünschte Materialobjekt bislang nicht enthält, wird mit Hilfe des Materialraumversorgers das *LRU*-Materialobjekt aus dem Materialcache in die angeschlossene ODMG-93-Datenbank verdrängt (*down_checkpoint(LRU_MID,LRU_TID)* und *invalidate(LRU_MID)*). Anschließend bedient sich der Materialraumverwalter wiederum des Materialraumversorgers, mit dessen Hilfe das gewünschte Materialobjekt aus der ODMG-93-Datenbank in den transienten Speicher transferiert wird (*up_checkpoint(MID)*). Darauf folgend wird das Materialobjekt in das Materialraummagazin eingetragen (*set(MID, TID)*) und es kann anschließend auf die (gecachte) transiente Repräsentation des Materials zugegriffen werden.

Da eine nach *Automobil* gecastete C++-Referenz zu ermitteln ist, wird der statische Typ des Materialobjekts abgefragt (*get_CID(MID)* und *get_classname(CID)*). Dadurch wird es möglich, die C++-Referenz von *abstraktes_material* auf den statischen Typ des Materialobjekts (*Sportwagen*) zu casten (*get_cast_offset("Sportwagen", "abstraktes_material")* und *add_offset(TID, offset)*). Damit erhalten wir in unserem Beispiel eine nach *Sportwagen* gecastete C++-Referenz. Da allerdings eine nach *Automobil* gecastete C++-Referenz zu ermitteln ist, wird zunächst geprüft, ob die Klasse *Sportwagen* eine Unterklasse der Klasse *Automobil* ist oder nicht (*is_castable("Sportwagen","Automobil")*). Da dies der Fall ist, kann anschließend die nach *Sportwagen* gecastete C++-Referenz nach *Automobil* konvertiert werden (*get_cast_offset("Sportwagen", "Automobil")* und *add_offset(TID, offset)*). Die auf diese Weise ermittelte C++-Referenz (*TID*) wird zurückgegeben, so daß die erste Phase, die zur Ermittlung einer C++-Referenz auf eine transiente Repräsentation des Materials dient, abgeschlossen ist.

In der abschließenden zweiten Phase kann die gewünschte Operation auf der transienten Materialrepräsentation ausgeführt werden (*get_status()*). Als Rückgabewert liefert diese in dem hier beschriebenen Beispiel einen C-String (*"ist schnell"*), der anschließend auf dem Bildschirm ausgegeben wird.

Szenario: Auflösung eines Materials

Analog zur Erzeugung von Materialien erfolgt auch die Materialauflösung (vgl. *Ein_Auto.destroy(DEEP)*) eines durch eine Materialraumreferenz (*Ein_Auto*) referenzierten Materialobjekts durch unmittelbaren Aufruf einer Operation des Materialraums (*destroy_material(MID,DEEP)*).

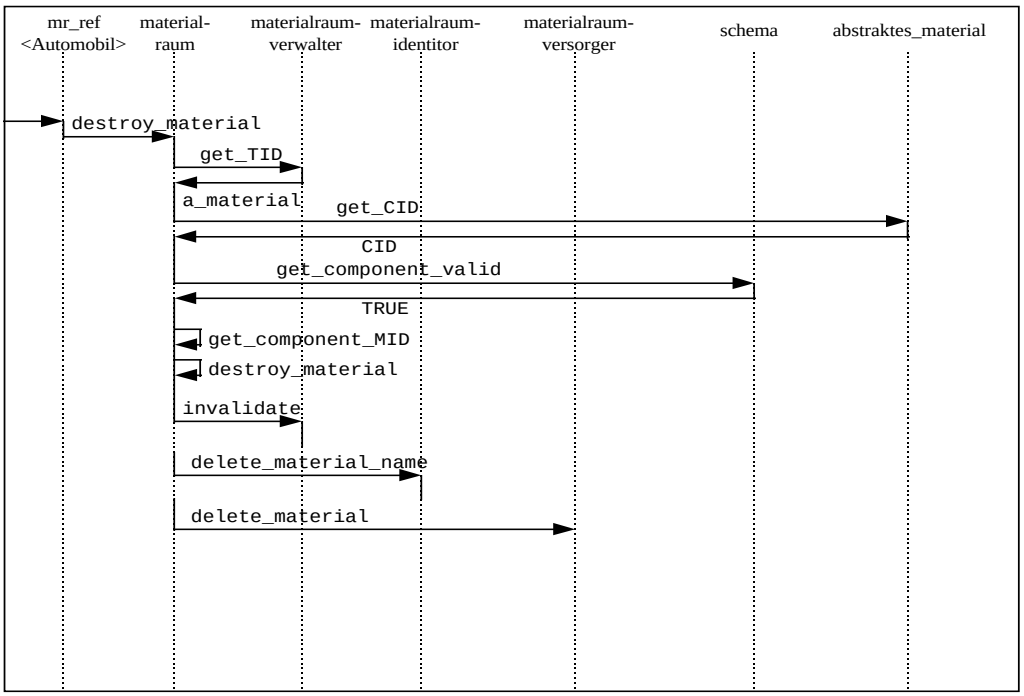


Abbildung 6.26: Interaktionsdiagramm *Ein_Auto.destroy(DEEP)*

Zunächst ermittelt dieser eine nach *abstraktes_material* konvertierte C++-Referenz auf das Materialobjekt und ermittelt darüber den dem Materialobjekt zugeordneten Materialtypidentifikator (*CID*). Damit kann der Materialraum unter zu Hilfenahme des Meta-Objekt-Protokolls (*schema*) die Komponentenmaterialraumreferenzen des Materialobjekts ermitteln (*get_component_valid(CID,...)* und *get_component_MID(CID,...)*) und die diesen Referenzen zugeordneten Materialobjekte ebenfalls tief auflösen (*destroy_material(CmpMID, DEEP)*)[79].

Anschließend wird der zum aufgelösten Materialobjekt zugehörige Eintrag im Materialraummagazin mit Hilfe des Materialraumverwalters invalidiert (*invalidate(MID)*), so daß ein weiterer Zugriff auf das gecachte Materialobjekt ausgeschlossen werden kann. Darüber hinaus wird auch der dem Materialobjekt zugeordnete fachliche Materialname freigegeben (*delete_material_name(MID)*), indem der Bezug zwischen dem Namen und dem Materialidentifikator (*MID*) des aufgelösten Materialobjekts beseitigt wird. Abschließend wird sowohl das transiente Materialobjekt aus dem transienten Speicher als auch das persistente Materialobjekt in der ODMG-93-Datenbank mit Hilfe des Materialraumversorgers beseitigt.

[79] Analog zu dem Interaktionsdiagramm der Materialerzeugung visualisiert auch das in Abbildung 6.26 dargestellte Interaktionsdiagramm lediglich die Auflösung eines Rades, obwohl insgesamt vier Räder aufgelöst werden.

6.6.2 Parametrisierung des Frameworks

Wie bereits oben festgestellt worden ist, kommt der Parametrisierung eines Frameworks eine entscheidende Rolle zu. Das Framework *dinx* nutzt die komplette Palette der verschiedenen Parametrisierungsarten (vgl. Abbildung 6.27[80]):

- *Parametrisierung durch Vererbung*:
  Der Benutzer des Frameworks *dinx* spezialisiert das Framework für eine bestimmte fachliche Anwendung, indem er konkrete anwendungsspezifische Unterklassen der abstrakten Framework-Klassen *abstraktes_material*, *abstrakte_fabrik*, *abstrakte_metafabrik* und gegebenenfalls auch von der Klasse *abstraktes_filesystem* oder *abstraktes_RDBS* bildet und Exemplare dieser Klassen instantiiert.

- *Parametrisierung durch Objektinstantiierung*:
  Der Benutzer des Frameworks *dinx* instantiiert zum Beispiel Exemplare der Klassen *materialraum*, *Database* und *schema* und koppelt anschließend die erzeugten Objekte (siehe unten).

- *Parametrisierung durch Klasseninstantiierung*:
  Der Benutzer des Frameworks *dinx* spezialisiert das Framework für eine bestimmte fachliche Anwendung, indem er eine generische Framework-Klasse (*mr_ref*<*T*> und *mr_cmp*<*T*>) mit einem Klassen-Parameter instantiiert und auf diese Weise eine neue Klasse (zum Beispiel *mr_ref*<*Automobil*>) bildet, von der Exemplare erzeugt werden können.

Auch das in diesem Kapitel vorgestellte Framework stellt mehrere abstrakte Klassen (zum Beispiel *abstraktes_material* und *abstrakte_fabrik*) zur Verfügung, von denen Unterklassen (zum Beispiel fachliche Materialklassen wie *Person* oder *Automobil*) konkretisiert werden können. Die Exemplare dieser Unterklassen werden anschließend durch das Framework genutzt und von diesem nach Bedarf angestoßen. Im Gegensatz dazu stellen zum Beispiel die Klassen *materialraum*, *mr_ref_any*, *mr_ref*<*T*> und *mr_cmp*<*T*> keine typischen Framework-Klassen dar, weil sie im Gegensatz zu den zu konkretisierenden Klassen (zum Beispiel *abstraktes_material*) nicht durch das Framework angestoßen werden (vgl. „Hollywood-Prinzip"), sondern sich stattdessen analog der Klassen einer Klassenbibliothek verhalten. In diesem Zusammenhang kann das in dieser Arbeit vorgestellte Framework *dinx* als eine Mischung aus Framework und Klassenbibliothek verstanden werden.

[80] Die Parametrisierung durch Vererbung wird durch eine Vererbungsbeziehung (Dreieck), die Parametrisierung durch Objektinstantiierung durch eine Benutzt-Beziehung (Pfeil) und die Parametrisierung durch Klasseninstantiierung durch eine generische Parametrisierungsbeziehung (Kreuz) visualisiert.

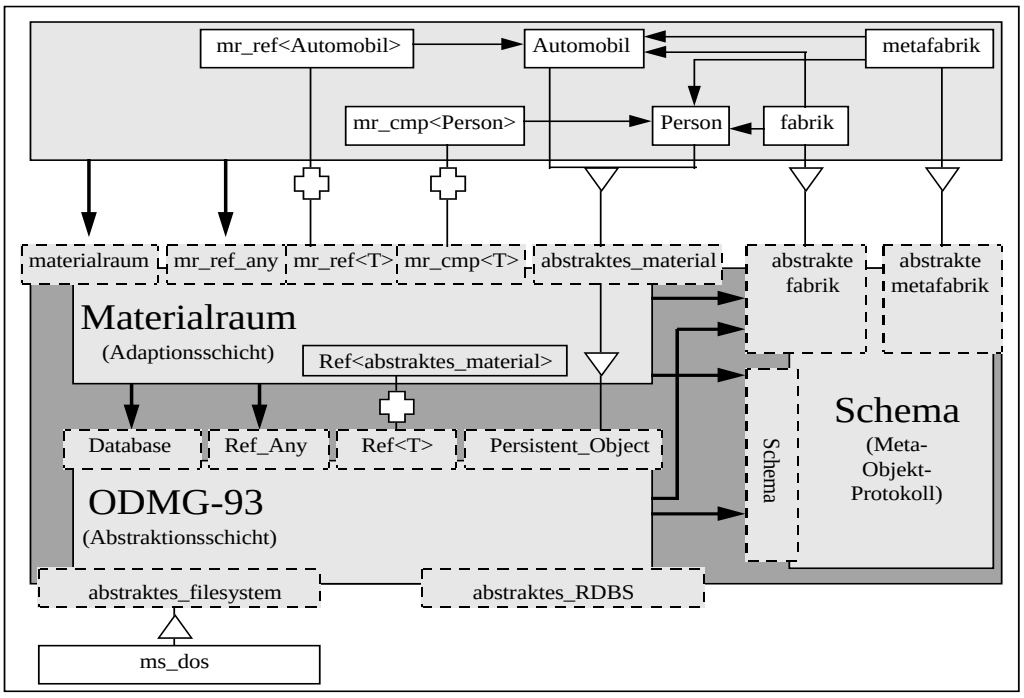


Abbildung 6.27: Parametrisierung des Frameworks *dinx*

Bei der Initialisierung des Frameworks (vgl. Abbildung 6.28, oberer Kasten) genügt es, in einem ersten Schritt Exemplare der Klassen *fabrik*, *metafabrik*, *schema*, *ms_dos*, *Database* und *materialraum* zu erzeugen. In einem zweiten Schritt wird das Schema initialisiert (*my_schema->init()*) und anschließend mit Hilfe der Metafabrik mit Meta-Objekt-Daten parametrisiert (*generate_schema()*). Da die Metafabrik anschließend nicht mehr benötigt wird, kann sie aus dem Speicher gelöscht werden. In einem dritten Schritt wird dann das Exemplar der Klasse *Database* (*my_ODMG*) mit einem konkreten persistenten Speicher (zum Beispiel *my_dos*), dem initialisierten Schema (*my_schema*) und einer Materialfabrik (*my_fabrik*) parametrisiert und damit mit dem Meta-Objekt-Protokoll gekoppelt. Abschließend wird der Materialraum mit dem Meta-Objekt-Protokoll (*my_fabrik* und *my_schema*) sowie mit der ODMG-93-Datenbank (*my_ODMG*) parametrisiert, so daß im Anschluß an diese Initialisierung die Adaptionsschicht, die Abstraktionsschicht und das Meta-Objekt-Protokoll untereinander gekoppelt sind.

Diese Kopplung wird erst zum Zeitpunkt der Prozeßterminierung aufgehoben (vgl. Abbildung 6.28, unterer Kasten). Wenn die Anwendung heruntergefahren wird, dann werden der Materialraum und die ODMG-93-Datenbank geschlossen (*close()*), so daß auch die Kopplung zwischen ihnen aufgehoben wird. Anschließend wird der durch die Exemplare der Klassen *materialraum*, *Database*, *ms_dos*, *schema* und *fabrik* belegte Hauptspeicher freigegeben.

```
my_fabrik        =new fabrik;
my_metafabrik    =new metafabrik;
my_schema        =new schema;
my_dos           =new ms_dos;
my_ODMG          =new Database;
my_raum          =new materialraum;
//-------------------------------------------------------------
my_schema->init();
my_metafabrik->generate_schema(my_schema);
delete my_metafabrik;
//-------------------------------------------------------------
my_ODMG->open("Materialraum", my_dos,my_schema,my_fabrik,
              read_write);
my_raum->open(my_fabrik, my_schema, my_ODMG);
```

```
my_raum->close();
my_ODMG->close();
my_schema->terminate();
//-------------------------------------------------------------
delete my_raum;
delete my_ODMG;
delete my_dos;
delete my_schema;
delete my_fabrik;
```

Abbildung 6.28: Initialisierung und Terminierung eines Materialraums

6.6.3 Zusammenfassung

Das zentrale Element des Frameworks *dinx* ist der Materialraum, der alle Materialien verwahrt und Konzepte zur Materialverknüpfung und Materialkomposition sowie den Materialzugriff bereitstellt. Zu diesem Zweck stellt das Framework drei unterschiedliche Materialraumreferenzen zur Verfügung, die den Zugriff auf Materialobjekte im Materialraum ermöglichen[81]. Darüber hinaus stellt der Materialraum das in Kapitel 4 vorgestellte Konzept der Umgangsarten zur Synchronisation nebenläufiger Handlungen an oder mit Materialobjekten zur Verfügung[82]. Der in diesem Zusammenhang beschriebene Materialraumkoordinator dient zusammen mit dem Materialraumverwalter und dem Materialraumversorger[83] zur Bereitstellung eines Persistent Distributed Shared Memory, mit dessen Hilfe es möglich wird, der Anwendungsschicht sowohl eine horizontale als auch vertikale Speicherabstraktion bereitzustellen. Diese Komponenten bleiben im Gegensatz zu den in [Wulf, Weske 94] und [Gryczan 95] beschriebenen Ansätzen den Nutzern des Materialraums verborgen. Alle Transformationen und Verschiebungen von technischen Materialrepräsentationen werden implizit über die Schnittstelle des Materialraums bzw. durch den Umgang mit Materialraumreferenzen angestoßen und durch das Framework *dinx* technisch transparent durchgeführt.

[81] vgl. die Unterabschnitte 6.2.3 und 6.2.4
[82] vgl. Unterabschnitt 6.2.6
[83] vgl. Unterabschnitt 6.3.2

Im folgenden Kapitel werden wir die Ergebnisse dieser Arbeit zusammenfassend darstellen und kritisch würdigen. In diesem Rahmen werden auch die wesentlichen Unterschiede zwischen dem Konzept des Materialraums und der in [Wulf, Weske 94] und [Gryczan 95] beschriebenen Materialverwaltung dargestellt.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung der Arbeit

Das übergeordnete Ziel dieser Arbeit, eine Grundlage für eine Zusammenführung von technischen und fachlichen Komponenten eines objektorientiert entwickelten Informationssystems zu erarbeiten, wurde erreicht: Der Materialraum bzw. der Persistent Distributed Shared Memory dienen als Gateway-Komponente zwischen der technischen und der fachlichen Sicht. Die Umsetzbarkeit der Konzepte des Materialraums bzw. des Persistent Distributed Shared Memory wurde in einer prototypischen Implementation in Form eines C++-Frameworks für Einzelplatzsysteme gezeigt.

Den in Kapitel 2 aufgestellten Entwicklungszielen der

- Unterstützung des qualifizierten Anwenders,
- Bereitstellung fachlicher Transparenz,
- Bereitstellung technischer Transparenz und
- Softwareintegration

wird unser Ansatz in besonderer Weise gerecht. Dies ist auf die konsequente Umsetzung einer 3-Schicht-Architektur zurückzuführen, durch die fachliche und technische Sichtweisen und Komponenten getrennt und durch eine Adaptionsschicht voneinander entkoppelt werden können:

- Unser konzeptionell entwickeltes Framework, das die Programmiersprache C++ zu einer mobilität-unterstützenden persistenten Programmiersprache erweitert, kapselt in einer Abstraktionsschicht paradigmatisch unterschiedliche persistente Speicher. Während objektorientierte Datenbanksysteme, die dem ODMG-93-Standard genügen, direkt anschließbar sind, konnten

wir unter Anbindung eines Dateisystems aufzeigen, daß beliebige persistente Speicher - auch relationale Datenbanksysteme - über einen ODMG-93-konformen Adapter anschließbar sind. Der geforderten Möglichkeit der Softwareintegration in Bezug auf persistenz-bereitstellende Systeme sind wir damit gerecht geworden.

- Die Konzeption der Adaptionsschicht, die den Schwerpunkt dieser Diplomarbeit bildete, beruht auf dem fachlichen Basiskonzept des Materialraums sowie seinem technischen Pendant (Persistent Distributed Shared Memory). Beide Konzepte gehen von der Grundannahme aus, daß sich alle Materialien in einem Raum bzw. einem logischen Speicher befinden und unabhängig von ihrer physikalisch horizontalen und vertikalen Lokation in einem verteilten System in gleicher Weise zugreifbar sind. Spezifische technische Details werden durch das Konzept des Persistent Distributed Shared Memory verborgen. Somit stellt die Adaptionsschicht die technische Transparenz bereit.

- Das fachliche Basiskonzept des Materialraums, das technisch durch den Persistent Distributed Shared Memory umgesetzt wurde[1], präsentiert sich als fachlich generelle Schnittstelle der Adaptionsschicht. Beliebig vernetzte Materialstrukturen können in diesem Raum erzeugt, verwahrt, kopiert oder aufgelöst werden. Darüber hinaus können Materialien durch die Umgangsarten Original, Kopie, Sicht und Präsenz sequentiell und quasi-gleichzeitig gemeinsam genutzt werden. Im Materialraum sind weder technische Komponenten sichtbar noch stellt dieser technische Funktionalität bereit. Damit bildet der Materialraum ein grundlegendes Konzept zur rein fachlichen Entwicklung von WAM-Anwendungen, die dem Leitbild des qualifizierten Anwenders genügen und die fachliche Transparenz losgelöst von technischen Details umsetzen.

- Die Adaptionsschicht ist sowohl fachlich als auch technisch interpretierbar, weil sie auf der einen Seite technische Transparenz und auf der anderen Seite fachliche Funktionalität bereitstellt. Jedoch führen diese fachlichen und technischen Anteile der Adaptionsschicht zur strikten Entkopplung der fachlichen Komponenten der Anwendungsschicht und der technischen Komponenten der Abstraktionsschicht. Diese strikte Trennung war ein elementares Ziel dieser Diplomarbeit.

Das fachliche Basiskonzept des Materialraums und dessen technisches Pendant konnten wir in den aufeinander aufbauenden Kapiteln dieser Diplomarbeit sukzessiv motivieren und herleiten. In Kapitel 2 zeigten wir anhand des motivierenden Beispiels eines CIM-Systems die Kluft zwischen den fachlichen und technischen Facetten eines Informationssystems und bei der Betrachtung der Komponenten

[1] beschränkt auf eine Einzelplatzlösung

eines Informationssystems die bestehende Kluft zwischen diesen auf. Darauf aufbauend konnten die für diese Arbeit grundlegenden Entwicklungsziele an ein Informationssystem darlegt und ein Architekturvorschlag, bei dem fachliche und technische Komponenten voneinander entkoppelt werden, unterbreitet werden. Auf dieser Grundlage wurde durch die Formulierung der konkreten Aufgabenstellung eine Positionierung dieser Arbeit im Rahmen der Entwicklung von Informationssystemen vorgenommen.

Im Vordergrund von Kapitel 3 stand die Herausarbeitung der Probleme, die bei den bestehenden Konzepten zur Materialverwaltung existieren und insbesondere auf die Korrelation fachlicher und technischer Komponenten zurückzuführen sind (siehe Tabelle 7.1, linke Spalte). Dazu bedurfte es einer Einführung in das Leitbild des Arbeitsplatzes für kooperative qualifizierte menschliche Tätigkeit und der Darstellung der bestehenden unterschiedlichen Materialverwaltungskonzepte. Aus der Vielzahl der genannten Probleme entstanden wichtige Entwurfselemente, die in das Konzept des Materialraums bzw. seines technischen Pendants (Persistent Distributed Shared Memory) einflossen.

In Kapitel 4 wurde das Konzept des Materialraums entwickelt. Ausgehend vom etablierten Begriff des Materials und den erarbeiteten Problemen des Konzepts der Materialverwaltung wurden die Konzepte der Materialkomposition und Materialverknüpfung, der Materialidentität, der Materialgleichheit und der Lebensdauer von Materialien erörtert und eingeführt. Darüber hinaus wurde das Konzept von Original und Kopie überarbeitet und durch die Einführung der Konzepte Sicht und Präsenz zum Konzept der Umgangsarten (Original, Kopie, Sicht und Präsenz) erweitert. Aufbauend auf dem erweiterten Materialbegriff und den rein fachlich geprägten Umgangsarten mit Materialien konnte die fachliche Basisabstraktion eines Materialraums eingeführt werden. Der Materialraum dient als Abstraktion von technischen Komponenten und macht die Materialien direkt über Materialraumreferenzen zugreifbar. Tabelle 7.1 stellt den Problemen der Materialverwaltung die im vierten Kapitel erarbeiteten Lösungsansätze gegenüber.

Den Schwerpunkt des fünften Kapitels bildete die Herausarbeitung einer Zusammenführung paradigmatisch unterschiedlicher Datenbanksysteme und eines Konzepts zur technischen Umsetzung des Materialraums. Zu diesem Zweck wurden die Zusammenführung der Funktionalität von Programmiersprachen und Datenbanksystemen in einer persistenten Programmiersprache und die Anforderungen an persistente Objektsysteme betrachtet. Dabei sind wir sowohl auf die Kluft, die zwischen den Modellen der Programmiersprachen C++, der relationalen Datenbanksysteme und der ODMG-93-konformen Datenbanksysteme besteht, als auch auf die Überbrückung dieser Kluft eingegangen. Damit sollte auch eine Grundlage für weiterführende Arbeiten geschaffen werden. Durch die Zusammenführung der Konzepte des virtuellen Speichers und des Distributed Shared Memory im

Konzept der Materialverwaltung	*Konzept des Materialraums*
Der Zugriff eines Werkzeugs auf ein Material über einen Materialverwalter ist nicht fachlich motivierbar.	Ein Werkzeug greift direkt über eine Materialraumreferenz auf ein Material im Materialraum zu. Der Materialraum substituiert den Materialverwalter.
Das Identitätskonzept von Materialien beruht auf fachlichen Namen. Bei der Veränderung des fachlichen Namens eines Materials wird damit seine Identität verändert. Ein Material ohne einen fachlichen Namen besitzt keine Identität.	Die Identität eines Materials ist unveränderlich, einmalig und vom System vergeben. Sie ist vom Konzept des fachlichen Namens getrennt. Die Materialidentität wird durch einen Materialidentifikator realisiert.
Beim Zugriff auf Materialien werden fachliche Namen mit Hilfe des Materialmagazins direkt auf technische Adressen abgebildet.	Die Korrelation der technischen und fachlichen Anteile des Materialmagazins werden durch die Indirektion des Materialidentifikators aufgehoben.
Der Kopiebegriff ist technisch geprägt und entspricht nicht dem intuitiven Kopiebegriff einer Zweitschrift. Eine Kopie hat keine eigene Identität und ist nicht modifizierbar.	Die Erstellung einer Kopie entspricht der Erstellung einer Zweitschrift, die eine eigene Identität besitzt, zunächst zustandsgleich mit dem Original und modifizierbar ist.
Das Konzept von Original und Kopie schränkt nebenläufige Handlungen unnötig ein.	Die Differenzierung des Originalbegriffs und die Einführung der Konzepte der Präsenz und Sicht erlauben ein sensibleres ,,Sperren" von Materialien und damit die quasi-gleichzeitige Nutzung von Materialien.
Das Material verliert seinen Materialcharakter, weil es Wissen über seine eigene Repräsentation in unterschiedlichen Speichern hält, Operationen zur Beschaffung und persistenten Ablage besitzt und damit mit konkreten persistenten Speichern fest verdrahtet ist (Aspekt speicherbar).	Die Schnittstelle eines Materials ist rein fachlich interpretierbar. Das Zusammenspiel aus Materialraumverwalter und Materialraumversorger unter der Verwendung von Meta-Objekt-Wissen ermöglicht, daß Materialien beschafft sowie zwischen unterschiedlichen Speichern ausgetauscht und zwischen Repräsentationen abgebildet werden können (kein Aspekt speicherbar).
Der Anschluß paradigmatisch unterschiedlicher Datenbanksysteme impliziert für jedes Datenbanksystem und jedes Material die Implementation des Austauschs zwischen den Materialrepräsentationen.	Für jeden Materialtyp wird einmalig Meta-Objekt-Wissen angelegt. Ein spezifischer Datenbanksystemadapter kann selbständig mit Hilfe dieses Meta-Objekt-Wissens zwischen unterschiedlichen Repräsentationen abbilden.
Die vernetzte (auch zyklische) Struktur von Materialien findet keine Berücksichtigung. Beim Transfer eines Materials in den transienten Speicher werden alle referenzierten Materialien mit transferiert. Dieses kann die nebenläufige Nutzbarkeit von Materialien erheblich einschränken.	Jedes Material in einer vernetzten Materialstruktur kann einzeln gehandhabt werden. Durch die Unterstützung von Kompositionsbeziehungen können komplexe Materialien sowohl partiell als auch als Ganzes gehandhabt werden.

Tabelle 7.1: Gegenüberstellung der Probleme der Materialverwaltung und der Lösungsansätze des Materialraums

Konzept des Persistent Distributed Shared Memory, bei dem sowohl von der vertikalen Speichertrennung als auch von der horizontalen Speichertrennung abstrahiert wird, gelang es, ein technisches Pendant zum fachlichen Konzept des Materialraums zu entwickeln.

Um die in Kapitel 4 und 5 aufgestellten Konzepte auf ihre praktische Umsetzbarkeit zu verifizieren, wurde ein Framework für die Programmiersprache C++ entwickelt, das in Kapitel 6 dargestellt wurde. Durch die Einführung eines generischen Zeigermechanismus (Materialraumreferenz-Mechanismus) und die Bereitstellung eines Meta-Objekt-Protokolls gelang es, die technisch transparente Unterstützung beliebiger Lebensdauer unter Anschluß beliebiger paradigmatisch unterschiedlicher Datenbanksysteme und die partielle Handhabung von vernetzten Materialobjektstrukturen zu realisieren ohne den gewohnten Umgang des Entwicklers mit der Programmiersprache C++ stark zu verändern.

Die Transparenz der Persistenz (*persistence independence*) wurde durch einen Cache und die Orthogonalität des Sekundärspeichers (*orthogonal persistent storage*) durch eine ODMG-93-konforme Schnittstelle umgesetzt. Die Anschließbarkeit paradigmatisch unterschiedlicher persistenter Speicher wurde durch die Entwicklung eines ODMG-93-konformen Adapters, mit dem ein Dateisystem angeschlossen wurde, gezeigt. Lediglich die Orthogonalität der Persistenz (*persistent data type orthogonality*) konnten wir nur eingeschränkt umsetzen. Jede Klasse, deren Exemplare eine beliebige Lebensdauer haben können, muß Unterklasse der abstrakten Klasse *Persistent_Object* sein (*persistence by type*). Da diese Mixin-Klasse in der Vererbungshierarchie über der Klasse *abstraktes_material* angeordnet ist, bleibt sie der Anwendungsschicht verborgen und es kann sichergestellt werden, daß alle Materialobjekte (unabhängig von ihrem Materialtyp) persistent ablegbar sind.

Nach den Maßstäben der in Kapitel fünf dargestellten Anforderungen an persistente Objektsysteme erfüllt die Erweiterung der Programmiersprache C++ zu einer persistenten Programmiersprache durch das Framework *dinx* nicht die nachfolgend aufgeführten Eigenschaften:

- Bereitstellung einer (deklarativen) Anfragesprache
- Bereitstellung von Transaktionen
- Benutzerautorisierung und Zugriffsschutz
- Schemaevolution
- Gleichzeitiger Anschluß mehrerer Datenbanksysteme

Die Entwicklung von mehrbenutzerfähigen Anwendungen ist im Konzept des Persistent Distributed Shared Memory berücksichtigt, jedoch im Rahmen der prototypischen Realisierung des Framework *dinx* nicht umgesetzt worden. Da sowohl

der generische Zeigermechanismus (Materialraumreferenz) als auch die Bereitstellung von Meta-Objekt-Wissen die Grundlage zur Bereitstellung transparenter Mobilität bilden, sollte die Umsetzbarkeit des Konzepts des Persistent Distributed Shared Memory und damit des Konzepts des Materialraums in verteilten Systemen möglich sein.

Ein grundsätzliches Problem der prototypischen Implementation besteht in der Außerkraftsetzung des statischen Typsystems von C++. Dies ist auf den generischen Zeigermechanismus zurückzuführen, der eine Ausprägung des *smart pointer* Ansatzes ist. Die Aufweichung des Typsystems stellt keine konzeptionelle Schwäche unseres Ansatzes dar, sondern ist lediglich ein Problem der Umsetzung. Die Beurteilung, inwieweit die Aufweichung für den realen Einsatz eine Einschränkung bedeutet, hängt von einer Abwägung der bereitgestellten, vorteilhaften Merkmale des Framework *dinx* mit dem jeweiligen Einsatzgebiet ab.

Aus den besonderen Eigenschaften des Persistent Distributed Shared Memory, der eine Weiterentwicklung gegenüber anderen Systemen mit virtuellem oder gemeinsamem Speicher darstellt, ist die Erzeugung und Verwaltung von Meta-Objekt-Wissen, die partielle Handhabung von vernetzten Material- bzw. Objektstrukturen und der Umgang mit komplexen Materialien hervorzuheben. Die partielle Handhabung stellt den fachlich adäquaten Umgang mit Materialien sicher und erhöht den Grad nebenläufiger Handlungen. Dies ist auf die Verwendung von *smart pointern* zurückzuführen, durch die über eine einstufige Indirektion auf Objekte zugegriffen wird. Die Verknüpfung der (Material-)Objekte erfolgt direkt über eineindeutig vergebene Materialidentifikatoren.

Eine Prognose zur Zukunft des Programmiermodells des persistenten verteilten gemeinsamen Speichers (Materialraum) ist schwierig. Ein logisch gemeinsamer Adreßraum wird für die Programmierung von Anwendungssystemen als wünschenswert angesehen. Zur Zeit sind Informationssysteme auf der Basis einer horizontalen und vertikalen Speichertrennung realisiert. Eine reine Software-Implementation eines persistenten verteilten gemeinsamen Speichers kann hinsichtlich der Leistung nicht mit Systemen konkurrieren, die über spezielle Hardware- oder Betriebssystem-Unterstützung verfügen. Mit Hilfe des Materialraums können jedoch Erfahrungen gesammelt werden, die nicht nur die Abstraktion von der Persistenz und Mobilität, sondern auch die fachliche Anwendungsentwicklung betreffen. Diese Erfahrungen werden unserer Einschätzung nach auch für andere Entwicklungsansätze von Bedeutung sein.

Die Perspektive, die sich aus dem Materialraum bzw. aus dem Persistent Distributed Shared Memory in Bezug auf Unabhängigkeit, Anpaßbarkeit, Flexibilität und Erweiterbarkeit sowohl für fachliche Anforderungen als auch für spezielle Datenbanksysteme und Dienste zur Unterstützung von Mobilität ergeben, wur-

den dargelegt. Viele Aspekte, die dabei identifiziert und problematisiert wurden, konnten im Rahmen dieser Diplomarbeit nicht bearbeitet werden - zum Teil hätte eine intensive Auseinandersetzung mit einem solchen Thema bereits den Umfang einer einzelnen Diplomarbeit.

7.2 Weiterführende Arbeiten

In diesem letzten Abschnitt der Arbeit wird ein kurzer Überblick über eine Auswahl weiterführender Themen gegeben. Die am Ende von Kapitel 4 dargestellten höheren fachlichen Konzepte (Arbeitsvorgänge und Arbeitsschritte, fachliche Beweglichkeit, Besitz und Eigentum) bilden weitere Themenbereiche, die an dieser Stelle nicht wiederholt werden sollen.

- *Transaktionskonzepte und nebenläufige Handlungen qualifizierter Anwender*
  Es gibt Untersuchungen zu Transaktionskonzepten, die die Nebenläufigkeit weniger stark einschränken als traditionelle Techniken, die eine strikte Serialisierbarkeit fordern. Weil nebenläufige Handlungen mit vernetzten Materialstrukturen weitestgehend ermöglicht werden sollen, ist zu evaluieren, ob und in welchem Umfang die neuen Techniken nutzbar sind. Dazu muß auch untersucht werden, welche fachlichen Anforderungen an Transaktionen gestellt werden, inwieweit Konzepte aus dem *CSCW-Bereich* oder dem *Workflow-Management* antizipiert werden können und welche technischen Konzepte dann direkt als fachlich generelle nutzbar sind.

- *Anfragesprache*
  Ein Zugriff auf Materialobjekte ist bei dem in dieser Arbeit vorgestellten Ansatz per Navigation über Referenzen oder durch fachliche Namen möglich. Weil das Verfolgen von Referenzen in großen Datenbeständen nicht zielgerichtet und problemadäquat sein kann, um Materialobjekte effizient aufzufinden, und ein Überblick über alle fachliche Namen nicht möglich ist, bedarf es einer Anfragesprache. Die Anforderungen, die an eine solche Anfragesprache gestellt werden müssen, entsprechen den Anforderungen an objektorientierte Anfragesprachen: Ad-hoc Formulierung, Deskriptivität, Mengenorientiertheit, Abgeschlossenheit, Adäquatheit, Orthogonalität, Optimierbarkeit, Effizienz, Sicherheit, Eingeschränktheit, Vollständigkeit, Wahrung der Kapselung, Nutzung von Methoden. Die Entwicklung von Anfragesprachen für OODBS ist ein aktuelles Forschungsgebiet. Zur Zeit sind existierende Sprachen zum Teil nur an Sammlungen gebunden (zum Beispiel GemStone, ObjectStore) oder erfüllen nicht die volle Bandbreite der Ergebnisse objektorientierter Anfragen (Ergebnis mit relationaler Semantik, objekterzeugender Semantik, objekterhaltender Semantik).

- *Beweglichkeit von Objekten*
  Die Problematik unterschiedlicher Beweglichkeit von Objekten einer vernetzten Objektstruktur und die Frage nach der Fortpflanzung der Eigenschaft der Beweglichkeit von Objekten in vernetzten Objektstrukturen sind zu untersuchen.

- *Gleichzeitige Nutzung mehrerer Datenbanksysteme*
  Die gleichzeitige Nutzung mehrerer Datenbanksysteme stellt einen interessanten Aspekt da. Datenbanksysteme können an genau einen oder an unterschiedliche Rechnerknoten eines verteilten Systems angeschlossen werden. Dabei ergeben sich wichtige Fragestellungen: Welche Auswirkungen ergeben sich für die Realisierung des Materialraums? Welche Konsequenzen entstehen für Transaktionskonzepte und Anfragesprachen?

- *Performanz*
  Da sich diese Diplomarbeit insbesondere auf die Konzeption eines Materialraums und eines Persistent Distributed Shared Memory konzentrierte, diente die prototypische Umsetzung als Nachweis der Realisierbarkeit der Konzepte. Das Kriterium der Performanz ist aus diesem Grund nicht berücksichtigt worden, obwohl erst nach deren Berücksichtigung die Einsatztauglichkeit der vorgestellten Konzepte verifiziert werden kann.

- *Werkzeug-Automat-Material-Raum (WAM-Raum)*
  In Kapitel 3 haben wir dargelegt, daß die Festlegung, ob ein Gegenstand Material oder Werkzeug ist, objektiv nicht festlegbar und kontextabhängig ist. Deshalb ist zu untersuchen, ob die Erweiterung des Konzepts des Materialraums zu einem WAM-Raum, der auch Werkzeuge sowie Automaten aufnehmen kann, fachlich motivierbar ist.

- *Beziehungen zwischen Objekten*
  Die Objektmodelle der C++-binding des ODMG-93-Standards und der Programmiersprache C++ stellen nur einen Bruchteil der im Anhang A aufgelisteten Beziehungen zwischen Objekten bereit. Mit der Umsetzung der Kompositionsbeziehung konnten wir zeigen, daß das C++-Objektmodell im Rahmen unseres Frameworks um Objektbeziehungen erweitert werden kann. Mit Hilfe des bereitgestellten Meta-Objekt-Wissens sollte es möglich sein, weitere Objektbeziehungen umzusetzen.

Bei der Betrachtung weiterführender Fragestellungen sollte neben den praktischen Erfahrungen aus aktuellen Softwareentwicklungsprojekten auch das bei diesen Projekten gewonnene Fachwissen über unterschiedliche Anwendungsdomänen einfließen. Neben diesem fachlichen Wissen ist auch die Integration der Erkenntnisse technischer Disziplinen, wie die der Betriebssysteme, Programmiersprachen und Datenbanksysteme sowie die der Entwicklung von Informationssystemen,

zur Beantwortung der Fragestellungen heranzuziehen. Auf diese Weise kann der Entstehung von Insel- und Speziallösungen entgegengewirkt und das ständige Neuerfinden bereits bestehender Konzepte minimiert werden.

Anhang A

Eigenschaften von Objekten und Beziehungen zwischen Objekten, Klassen und Typen

Für die in Kapitel 5 vorgenommene Untersuchung der Modelle konnten aus der Literatur ([Cattell et al. 94], [Heuer 92], [Heuer, Saake 95], [Stürner 93] und [Wetzel 94]) die Eigenschaften von Objekten und die Beziehungen zwischen diesen sowie Beziehungen zwischen Klassen und zwischen Typen als auch Beziehungen zwischen Objekten, Klassen und Typen zusammengetragen werden.

Eigenschaften von Objekten

- *Abhängigkeit*:
  Ein Objekt ist abhängig, wenn es von der Existenz eines anderen Objekts abhängt.
  - unabhängig
  - abhängig (von n Objekten)
- *Änderbarkeit*:
  Ein Objekt ist änderbar und hat eine Historie, wenn sein Zustand zeitlich abhängig ist und über die Zeit gespeichert wird.
  - unveränderbar
  - (n-mal) änderbar:
    * ohne Historie
    * mit Historie

- *Bekanntheit*:
  Ein Objekt heißt bekannt, wenn es einem anderen Objekt durch eine Beziehung bekannt ist.
  - unbekannt
  - bekannt:
    * privat, disjunkt
    * (n-fach) gemeinsam, nicht disjunkt
- *Duplizierbarkeit*:
  Ein Objekt ist duplizierbar, wenn neue Objekte vom gleichen Typ und gleichen Zustand erzeugbar sind.
  - nicht duplizierbar (Singelton)
  - (n-fach) duplizierbar
- *Lebensdauer*:
  Ein Objekt ist persistent, wenn es seinen erzeugenden Prozeß überlebt.
  - transienter Gültigkeitsbereich
  - persistenter Gültigkeitsbereich
- *Mobilität*:
  Ein Objekt ist mobil, wenn es seinen Adreßbereich verlassen kann.
  - immobil
  - mobil
  - ubiquitär
- *Rekursivität*:
  Ein Objekt heißt rekursiv, wenn es mit Objekten des eigenen Typs in Beziehung steht.
  - nicht-rekursiv
  - rekursiv
- *Sichtbarkeit*:
  Ein Objekt ist gekapselt, wenn es nur über ein anderes Objekt erreichbar ist.
  - nicht gekapselt
  - gekapselt

Beziehungen zwischen Objekten

- *Kardinalität der Beziehung*:
  1. einwertig: 1:1
  2. mehrwertig: 1:n, n:m
- *Reflexivität der Beziehung*:
  1. nicht reflexiv: uni-direktional
  2. reflexiv: bi-direktional
- *Partnerkardinalität der Beziehung*:
  1. binär: Beziehung zwischen 2 Objekten
  2. ternär: Beziehung zwischen 3 Objekten
  3. n-är: Beziehung zwischen n Objekten
- *Attributiertheit der Beziehung*:
  1. nicht attributiert
  2. attributiert

Beziehungen zwischen Klassen und zwischen Typen

- *Typ-Typ-Beziehung* (Typvererbung)
  1. *(Is-Subtype-Of, Typ-Spezialisierung (IS-A))*:
     - Ein Typ kann nur genau einen Obertyp haben (1:1) (Einfachvererbung).
     - Ein Typ kann mehrere Obertypen haben (n:1) (Mehrfachvererbung).
  2. *(Is-Supertype-Of, Typ-Generalisierung)*:
     - Ein Typ kann nur genau einen Untertyp haben (1:1).
     - Ein Typ kann mehrere Untertypen haben (1:n).
- *Klasse-Klasse-Beziehung* (Klassenvererbung)
  1. *(Is-Subclass-Of)*:
     - Eine Klasse kann nur genau eine Oberklasse haben (1:1) (Einfachvererbung).
     - Eine Klasse kann mehrere Oberklassen haben (n:1) (Mehrfachvererbung).
  2. *(Is-Superclass-Of)*:
     - Eine Klasse kann nur genau eine Unterklasse haben (1:1).
     - Eine Klasse kann mehrere Unterklassen haben (1:n).

Beziehungen zwischen Objekt, Klasse und Typ

- *Objekt-Klasse-Beziehung* (Is-Instance-Of):
  - Ein Objekt ist immer genau einer Klasse zugeordnet (1:1 oder n:1).
- *Klasse-Objekt-Beziehung* (Is-Template-Of):
  - Eine Klasse kann genau einmal instantiiert werden (1:1, Singelton).
  - Eine Klasse kann beliebig oft instantiiert werden (1:n).
- *Klasse-Typ-Beziehung* (Is-Implementation-Of):
  - Eine Klasse ist immer genau einem Typ zugeordnet (1:1 oder n:1).
- *Typ-Klasse-Beziehung* (Is-Specification-Of):
  - Zu einem Typ kann es nur eine Implementation geben (1:1).
  - Zu einem Typ kann es mehrere Implementationen geben (1:n).

Anhang B

Quellcode der prototypischen Realisierung des Frameworks

Literaturverzeichnis

[Aarsten et al. 96] A. Aarsten, D. Brugali, G. Menga:
Patterns for Three-Tier Client/Server Applications in: PLoP '96 Proceedings, Allerton Park, 1996.

[ACM 91] Communications of the ACM:
Next-Generation Database Systems, Special Issue, Communications of the ACM, Vol.34, No.10, 10/1991.

[Agrawal et al. 93] R. Agrawal, S. Dar, N. Gehani:
The O++ Database Programming Language: Implementation and Experience, in: Proceedings Ninth International Conference on Data Engineering, IEEE Computer Society, Wien, 1993.

[Agarwal et al. 95] S. Agarwal, C. Keene, A. M. Keller:
Architecting Object Applications for High Performance with Relational Databases, in: OOPSLA '95 Proceedings, Austin, 1995.

[Allard, Wile 89] Dennis G. Allard, David S. Wile:
Aggregation, Persistence, and Identity in Worlds, in: Persistent Object Systems, John Rosenberg et al., Springer, Heidelberg, 1989.

[Ananthanarayanan et al. 93] R. Ananthanarayanan, V. Gottemukkala, W. Kaefer, T. J. Lehman, H. Pirahesh:
Using the Co-existence Approach to Achieve Combined Functionality of Object-Oriented and Relational Systems, in: ACM SIGMOD, No.5, Seite 109-118, 1993.

[Arens 93] Thomas Arens:
Konsistenz verteilter Softwareproduktionsumgebungen, in: Objekte in verteilten Systemen 9, Seminar SS93, Institut für Telematik, Universität Karlsruhe, 1993.

[Assenmacher et al. 93] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, R. Schwarz:
PANDA - Supporting Distributed Programming in C++, in: ECOOP '93 Proceedings, http://www.uni-kl.de/AG-Nehmer/panda/panda.html, Kaiserslautern, 1993.

[Assenmacher et al. 94] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, H. Peine, R. Schwarz:
Parallel Programming in PANDA, http://www.uni-kl.de/AG-Nehmer/panda/panda.html, 1994.

[Atkinson et al. 83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, R. Morrison:
An Approach to Persistent Programming, in: The Computer Journal, The British Computer Society, Swindon, 1983.

[Atkinson, Morrison 89] Malcolm Atkinson, Ronald Morrison:
Persistent System Architectures, in: Persistent Object Systems, John Rosenberg et al., Springer, Heidelberg, 1989.

[Atkinson et al. 90] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik:
The Object-Oriented Database System Manifesto, aus: Proc. of the 1st. International Conference on Deductive and Object-Oriented Databases, 1990.

[Attia 96] Karim Attia:
Persistenz in objektorientierten Anwendungssystemen - Über die Abbildung von Objektstrukturen auf relationale Strukturen, Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1996.

[Baker 95] Sean Baker:
CORBA and Databases - Do you really need both?, SIGS Publications, New York, 1995.

[Bal 90] Henri E. Bal:
Languages for Parallel Programming, Faculteit der Wiskunde en Informatica, Rapportnr. IR-226, Vrije Universiteit, Amsterdam, 1990.

[Bal 91] Henri E. Bal:
A Comparative Study of Five Parallel Programming Languages, Faculteit der Wiskunde en Informatica, Rapportnr. IR-250, Vrije Universiteit, Amsterdam, 1991.

[Bal, Grune 94] Henri E. Bal, Dick Grune:
Programming Languages Essentials, Addison-Wesley, Reading, 1994.

[Berri 90] Catriel Berri:
A formal approach to object-oriented databases, in: Data&Knowledge Engineering, Vol.5, Seite 353-382, Elsevier Science Publishers B.V., Amsterdam, 1990.

[Biliris et al. 93] A. Biliris, S. Dar, N. H. Gehani:
Making C++ Objects Persistent: the Hidden Pointers, in: Software-Practice and Experience, Vol.23, No.12, Seite 1285-1303 , 12/1993.

[Bischofberger et al. 96] W. Bischofsberger, M. Guttman, D. Riehle:
Global Business Objects: Requirements and Solutions, Ubilab, Zürich, 1996.

[Booch 91] Grady Booch:
Object Oriented Design with Applications, Benjamin/Cummings Publishing Company, Redwood City, 1991.

[Booch 94] Grady Booch:
Objektorientierte Analyse und Design: Mit praktischen Anwendungsbeispielen, 2. Auflage, korrigierter Nachdruck 1996, Addison-Wesley, Bonn, 1994.

[Borghoff, Nast-Kolb 89] Uwe M. Borghoff, Kristof Nast-Kolb:
Distributed Systems: A Comprehensive Survey, Institut für Informatik, Technische Universität München, 1989.

[Brammer, Göhmann 96] Ulrich Brammer, Markus Göhmann:
Objektorientierung und Persistenz: Ein Anforderungskatalog an die Verwaltung von Materialien in Einzelplatz-Arbeitsumgebungen aus fachlicher und technischer Sicht, Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1996.

[Brockhaus 81] *Der große Brockhaus in 12 Bänden*, 18., völlig neubearbeitete Auflage, F. A. Brockhaus, Wiesbaden, 1981.

[Broghammer 89] Claudia Broghammer:
AMBER: Parallele und verteilte objekt-orientierte Programmierung, in: Objekte in verteilten Systemen, Seminar SS 89, Fakultät Informatik, Universität Karlsruhe, 1989.

[Brössler, Freisleben 89] P. Brössler, B. Freisleben:
Transactions on Persistent Objects, in: Persistent Object Systems, John Rosenberg et al., Springer, Heidelberg, 1989.

[Buschmann et al. 92] Frank Buschmann, Konrad Kiefer, Michael Stal:
A RUNTIME TYPE INFORMATION SYSTEM FOR C++, in: TOOLS 7, Seite 265-274, Prentice Hall, New York, 1992.

[Cattell 94] R.G.G. Cattell:
Object Data Management, Object-Oriented and Extended Relational Database Systems, überarbeitete Auflage, Addison-Wesley, Reading, 1994.

[Cattell et al. 94] R.G.G. Cattell et al.:
The Object Database Standard: ODMG-93, Morgan Kaufmann Publishers, San Francisco, 1994.

[Caughey et al. 93] S. J. Caughey, G. D. Parrington, S. K. Shrivastava:
SHADOWS - A Flexible Support System for Objects in Distributed Systems, Department of Computer Science, University of Newcastle upon Tyne, 1993.

[Caughey et al. 95] S. J. Caughey, S. K. Shrivastava:
Architectural Support for Mobile Objects in Large Scale Distributed Systems, Department of Computer Science, University of Newcastle upon Tyne, 1995.

[Cohen 96] Shimon Cohen:
Lightweight Persistence in C++, in: C++ Report, Mai 1996, SIGS Publications, New York, 1996.

[Coulouris et al. 94] George Coulouris, Jean Dollimore, Tim Kindberg:
Distributed Systems - Concepts and Design, second edition, Addison-Wesley, Reading, 1994.

[Dabbene, Damiani 95] D. Dabbene, S. Damiani:
Adding persistence to objects using smart pointers, Alenia Aeronautica, Artificial Intelligence Laboratory and Systems Engineering Dept., in: Journal of object-oriented programming, Vol.8, Nummer 3, Torino, 6/1995.

[Daerle et al. 91] A. Dearle, J. Rosenberg, F. Vaugham:
A Remote Execution Mechanism For Distributed Homogeneous Stable Stores, in: Database Programming Languages, Bulk Types & Persistent Data, The Third Intl. Workshop, Paris, 1991.

[Darwen, Date 95] H. Darwen, C. J. Date:
The Third Manifesto, in: SIGMOD RECORD, Vol. 24, No.1, March 1995.

[Date, White 88] C. J. Date, C. J. White:
A Guide To DB2, 2nd edition, Addison-Wesley, Reading, 1988.

[Dixon 88] G. N. Dixon:
Object Management for Persistence and Recoverbility, Technical Report Series, No. 276, University of Newcastle upon Tyne, 12/1988.

[Dixon et al. 89] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, S. M. Wheater:
The Treatment of Persistent Objects in Arjuna, Technical Report Series, No. 283, University of Newcastle upon Tyne, 6/1989.

[Edelson 92] Daniel R. Edelson:
Smart Pointers: They're Smart, but They're Not Pointers, in: USENIX C++ Conference Proceedings, Portland, 1992.

[Eirich 94] Thomas Eirich:
Fine-Grained Checkpointing in Distributed Object Systems, Technischer Report, Fachbereich Betriebssysteme, Friedrich-Alexander-Universität, Erlangen, Nürnberg, 1994.

[Eisenecker 96a] Ulrich W. Eisenecker:
Templates und Vererbung, in: OBJEKTspektrum, 2/96, Seite 90-91, SIGS, Bergisch Gladbach, 1996.

[Eisenecker 96b] Ulrich W. Eisenecker:
Templates statt Vererbung, in: OBJEKTspektrum, 4/96, Seite 92-95, SIGS, Bergisch Gladbach, 1996.

[Eisenecker 96c] Ulrich W. Eisenecker:
Generatives Programmieren mit C++, in: OBJEKTspektrum, 6/96, Seite 79-84, SIGS, Bergisch Gladbach, 1996.

[Fäustle 92] Michael Fäustle:
Beschreibung der Verteilung in objektorientierten Systemen, Dissertation, Arbeitsbericht des Instituts für mathematische Maschinen und Datenverarbeitung (Informatik), Band 25, Nummer 8, Friedrich-Alexander-Universität, Erlangen, Nürnberg, 1992.

[Fäustle 93] M. Fäustle:
Eine orthogonale Verteilungssprache für uniforme objektorientierte Programmiersprachen, Interner Bericht TR-14-11-93, Institut für Mathematische Maschinen und Datenverarbeitung, Friedrich-Alexander-Universität, Erlangen, Nürnberg, 1993.

[Fekete et al. 89] Alan Fekete, Nancy Lynch, Michael Merritt, William Weihl:
Commutativity-Based Locking for Nested Transactions, in: Persistent Object Systems, John Rosenberg et al., Springer, Heidelberg, 1989.

[Flor 96] Thomas Flor:
OOTIP-Workshop: Von Entwurfsmustern, Komponenten und Frameworks, in: OBJEKTspektrum, 2/96, Seite 93-94, SIGS, Bergisch Gladbach, 1996.

[Floyd 91] C. Floyd:
Einführung in die Softwaretechnik, Skriptum zur gleichnamigen Vorlesung, Fachbereich Informatik, Universität Hamburg, 1991.

[Gamma et al. 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides:
Design Patterns: Elements of Reusable Design, Deutsche Übersetzung von Dirk Riehle, Addison-Wesley, Bonn, 1996.

[Geppert 97] A. Geppert:
Objektorientierte Datenbanksysteme - Ein Praktikum, dpunkt, 1. Auflage, Heidelberg, 1997.

[Gessner 93] Wilfried Gessner:
Oasis - Die Programmiersprache, in: Objekte in verteilten Systemen 9, Seminar SS93, Institut für Telematik, Universität Karlsruhe, 1993.

[Giloi 93] W. K. Giloi:
Rechnerarchitektur, 2. überarbeitete Auflage, Springer, Heidelberg, 1993.

[Gryczan 95] G. Gryczan:
Situierte Koordination computergestützter qualifizierter Tätigkeit über Prozeßmuster, Dissertation, Fachbereich Informatik, Universität Hamburg, 1995.

[Haines 95] Jason Haines:
Persistent Object Store Applications, Batchelor's Thesis (Draft), Department of Computer Science, Australian National University, Canberra, Australia, 1995.

[Herrmann et al. 89] Herrmann et al.:
Sperren disjunkter, nicht-rekursiver komplexer Objekte mittels objekt- und anfragespezifischer Sperrgraphen, Heidelberg, 1989.

[Heuer 92] Andreas Heuer:
Objektorientierte Datenbanken: Konzepte, Modelle, Systeme, Addison-Wesley, Bonn, 1992.

[Heuer, Saake 95] Andreas Heuer, Gunter Saake:
Datenbanken: Konzepte und Sprache, International Thomson Publishing GmbH, Bonn, 1995.

[Heuser, Schill 95] Lutz Heuser, Alexander Schill:
Verteiltes objektorientiertes Programmieren mit C++ - Das Programmsystem DC++, International Thomson Publishing GmbH, Bonn, 1995.

[Hirschfeld 96] Robert Hirschfeld:
Three-Tier Distribution Architecture, in: PLoP '96 Proceedings, Allerton Park, 1996.

[Hoener 91] Alexander Hoener:
Das DEC Trellis/DOWL Projekt für verteilte Systeme, in: Objekte in verteilten Systemen, Seminar SS 91, Fakultät Informatik, Universität Karlsruhe, 1991.

[Hohenstein 96] Uwe Hohenstein:
Bridging the Gap between C++ and Relational Databases, in: ECOOP '96 Proceedings, Pierre Cointe (Hrsg.), Springer, Heidelberg, 1996.

[Hohenstein et al. 96] Uwe Hohenstein, Regina Lauffer, Klaus-Dieter Schmatz, Petra Weikert:
Objektorientierte Datenbanksysteme, ODMG-Standard, Produkte, Systembewertung, Benchmarks, Tuning, Vieweg, Braunschweig, 1996.

[Jessen, Valk 87] Eike Jessen, Rüdiger Valk:
Rechensysteme: Grundlagen der Modellbildung, Studienreihe Informatik, Springer, Heidelberg, 1987.

[Josuttis 94] Nicolai Josuttis:
Objektorientiertes Programmieren in C++: Von der Klasse zur Klassenbibliothek, 1. Auflage, Addison-Wesley, Bonn, 1994.

[Kakkad et al. 92] Sheetal V. Kakkad, Vivek Singhal, Paul R. Wilson:
Texas: An Efficient, Portable Persistent Store, in: Persistent Object Systems, Antonio Albano et al., Springer, Heidelberg, 9/1992.

[Keller, A. M. 94] A. M. Keller:
Penguin: Objects for Programs, Relations for Persistence, Computer Science Department, Standford University, 1994.

[Keller, L. 94] L. Keller et al:
Aktive und mobile Objekte als Modellierungskonzept für dezentrale Ingenieuranwendungen, Universität Karlsruhe, 1994.

[Kemper, Moerkotte 92] A. Kemper, G. Moerkotte:
Grundlagen objektorientierter Datenbanksysteme, Bericht Nr. 92/9, Aachener Informatik-Berichte, Fachgruppe Informatik, Rheinisch-Westphälische Technische Hochschule Aachen, 1992.

[Kilberth et al. 93] K. Kilberth, G. Gryczan, H. Züllighoven:
Objektorientierte Anwendungsentwicklung, Konzepte, Strategien, Erfahrungen, Vieweg, Braunschweig, 1993.

[Kirsche 94] Thomas Kirsche:
Kooperation und Koordination in verteilten Systemen, in: Verteilte Systeme - Grundlagen und zukünftige Entwicklung, Hartmut Wedekind (Hrsg.), BI-Wissenschaftsverlag, Mannheim, 1994.

[Kirschke 95] H. Kirschke:
Persistenz in der objekt-orientierten Programmiersprache CLOS am Beispiel des PLOB! Systems, Bericht Nr. 179, Fachbereich Informatik, Universität Hamburg, 1995.

[Klein 91] Gerhard Klein:
Konfigurationsverwaltung des LOCUS Betriebssystems, in: Objekte in verteilten Systemen, Seminar SS 91, Fakultät Informatik, Universität Karlsruhe, 1991.

[Kleinöder 92] Jürgen Kleinöder:
Objekt- und Speicherverwaltung in einer offenen, objektorientierten Betriebssystemarchitektur, Dissertation, Institut für mathematische Maschinen und Datenverarbeitung, Friedrich-Alexander-Universität, Erlangen, Nürnberg, 1992.

[Kopp 89] Martin Kopp:
Vergleich verschiedener Kommunikationsmechanismen für verteilte Systeme, in: Objekte in verteilten Systemen, Seminar SS 89, Fakultät Informatik, Universität Karlsruhe, 1989.

[Kowalski 93] Alexander Kowalski:
Distributed object management - DOM, in: Objekte in verteilten Systemen 8, Seminar WS92/93, Institut für Telematik, Universität Karlsruhe, 1993.

[Lebastard 95] Franck Lebastard:
Is an object layer on a relational database more attractive than an object database?, CERMICS/INRIA - BP 93, Sophia-Antipolis Cedex, Frankreich, 1995.

[Leikauf 89] Peter Leikauf:
Konsistenzsicherung durch Verwaltung von Konsistenzverletzungen, in: Datenbanksysteme in Büro, Technik und Wissenschaft, Theo Härder (Hrsg.), Springer, Heidelberg, 1989.

[Lenkov et al. 91] Dimitry Lenkov, Michey Mehta, Shankar Unni:
Type Identification in C++, in: USENIX C++ Conference Proceedings, Washington D.C., 1991.

[Listl 94] Andreas Listl:
Using Subpages for Coherency Control in Parallel Database Systems, Department of Computer Science, TU München, 1994.

[Livesey, Allison 92] Mike Livesey, Colin Allison:
Coherence in Distributed Persistent Object Systems, in: Persistent Object Systems, Antonio Albano et al., Springer, Heidelberg, 9/1992.

[Loomis 93a] M. E. S. Loomis:
Object programming + database management - Differences in perspective between the two, in: Journal of object-oriented programming, Vol.5, Nummer 5, Colorado Springs, 1993.

[Loomis 93b] M. E. S. Loomis:
The ODMG object model, in: Journal of object-oriented programming, Vol.5, Nummer 6, Colorado Springs, 1993.

[Loomis 93c] M. E. S. Loomis:
Object programming + database management, in: Journal of object-oriented programming, Vol.5, Nummer 9, Colorado Springs, 1993.

[Loomis 93d] M. E. S. Loomis:
Making objects persistent, in: Journal of object-oriented programming, Vol.5, Nummer 10, Colorado Springs, 1993.

[Loomis 94] M. E. S. Loomis:
Hitting the relational wall, in: Journal of object-oriented programming, Vol.6, Nummer 1, Colorado Springs, 1994.

[Mairandres 96] Martin Mairandres:
Virtuell gemeinsamer Speicher mit integrierter Laufzeitbeobachtung, Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich GmbH, Jülich, 1996.

[Mathiske 96a] Bernd Mathiske:
Mobilität in persistenten Objektsystemen, Folien des Vortrags im Oberseminar DBIS SS96, Universität Hamburg, 1996.

[Mathiske 96b] Bernd Mathiske:
Mobilität in persistenten Objektsystemen, Dissertation, Arbeitsbereich DBIS, Fachbereich Informatik, Universität Hamburg, 1996.

[Matthes 95] Florian Matthes:
Objektorientierte Daten- und Workflowmodellierung, Skriptum zur gleichnamigen Vorlesung, Fachbereich Informatik, Universität Hamburg, 1995.

[Meckenstock et al. 94] Axel Meckenstock, Detlef Zimmer, Rainer Unland:
Ein Konzept für kooperierende Transaktionen in Entwurfsumgebungen, CADLAB-Report 37/1994, C-LAB, Paderborn, 1994.

[Mertens et al. 92] Peter Mertens, Freimut Bodendorf, Wolfgang König, Arnold Picot, Matthias Schumann:
Grundzüge der Wirtschaftsinformatik, 2. Auflage, Springer, Heidelberg, 1992.

[Meyer 90] Bertrand Meyer:
Objektorientierte Softwareentwicklung, Prentice Hall, London, 1990.

[Mösching 96] Andreas Mösching:
Das Speichern von Objekten in Tabellen, in: OBJEKTspektrum, 5/96, Seite 72-79, SIGS, Bergisch Gladbach, 1996.

[O'Toole, Shrira 94] James O'Toole, Liuba Shrira:
Hybrid Caching for Large-Scale Object Systems, in: Persistent Object Systems, Malcolm Atkinson et al., Springer, Heidelberg, 1994.

[Oberquelle 91] Horst Oberquelle:
Kooperative Arbeit und menschengerechte Groupware als Herausforderung für die Software-Ergonomie, in: Kooperative Arbeit und Computerunterstützung (Arbeit und Technik, Praxisorientierte Beiträge aus Psychologie und Informatik, Band 1), Verlag für angewandte Psychologie, Göttingen, Stuttgart, 1991.

[Parrington 92] Graham D. Parrington:
Programming Distributed Applications Transparently in C++: Myth or Reality?, Computing Laboratory, University of Newcastle upon Tyne, 1992.

[Porter 89] Harry H. Porter, III:
Persistence in a Distributed Object Server, in: Persistent Object Systems, John Rosenberg et al., Springer, Heidelberg, 1989.

[Rahayu, Chang 93] Wenny J. Rahayu, Elizabeth Chang:
A Methodology for Transforming an Object-Oriented Data Model to a Relational Database, in: TOOLS 12, 1993.

[Rahm 93] E. Rahm:
Kohärenzkontrolle in verteilten Systemen, Fachbereich Informatik, Universität Kaiserslautern, 1993.

[Raisch 91] Jürgen Raisch:
Untersuchung der Entwurfsmethode OMT, in: Objekte in verteilten Systemen, Seminar SS 91, Fakultät Informatik, Universität Karlsruhe, 1991.

[Richardson, Carey 89] J. E. Richardson, M. J. Carey:
Persistence in the E Language: Issues and Implementation, in: Software-Practice and Experience, Vol. 19(12), Seite 1115-1150, 12/1989.

[Richardson, Carey 93] J. E. Richardson, M. J. Carey:
The Design of the E Programming Language, in: ACM Transactions on Programming Languages and Systems, Vol.15, No.3, Seite 494-534, 7/1993.

[Riehle 95] Dirk Riehle:
Muster am Beispiel der Werkzeug und Material Metapher, Ubilab, Zürich, 1995.

[Riehle et al. 96] D. Riehle, W. Siberski, D. Bäumer, D. Megert, H. Züllighoven:
The Atomizer - Efficently Streaming Object Structures, in: PLoP '96 Proceedings, Allerton Park, 1996.

[Robie, Bartels 95] J. Robie, D. Bartels:
Why Use an ODBMS? A Comparison Between Relational and Object Oriented Databases for Object Oriented Application Development, Poet Software, Hamburg, 1995.

[Rumbaugh 93] James Rumbaugh:
Objektorientiertes Modellieren und Entwerfen, Deutsche Übersetzung von Doris Märtin, Prentice-Hall, München, 1993.

[Schmidt et al. 94] J. Schmidt, R. Müller:
Objektspeichersysteme, Skriptum zur gleichnamigen Vorlesung, Fachbereich Informatik, Universität Hamburg, 1994.

[Schneider 91] Hans-Jochen Schneider (Hrsg.):
Lexikon der Informatik und Datenverarbeitung, 3. aktualisierte und wesentlich erweiterte Auflage, Oldenbourg-Verlag, München, Wien, 1991.

[Schröder 96] Gerald Schröder:
Kooperierende persistente Objektsysteme, Folien des Vortrags im Oberseminar DBIS SS96, Universität Hamburg, 1996.

[Shrivastava et al. 88] S. K. Shrivastava, G. N. Dixon et al.:
A technical overview of Arjuna: a system for reliable distributed computing, Technical Report Series, No. 262, University of Newcastle upon Tyne, 7/1988.

[Shrivastava 95] Santosh K. Shrivastava :
Lessons Learned from Building and Using the Arjuna Distributed Programming System, Department of Computer Science, University of Newcastle upon Tyne, 1995.

[Sigmund 93] Ulrich Sigmund:
Fragmentierte Objekte, in: Objekte in verteilten Systemen 8, Seminar WS92/93, Institut für Telematik, Universität Karlsruhe, 1993.

[Silberschatz et al. 96] A. Silberschatz, M. Stonebraker, J. D. Ullman:
Database Research: Achievements and Opportunities into the 21st Century, Department of Computer Science, Stanford University, 1996.

[Simmel, Godard 92] Sergiu S. Simmel, Ivan Godard:
Objects of Substance, BYTE, 12/92, Peterborough, 1992.

[Singhal et al. 92] Vivek Singhal, Sheetal V. Kakkad, Paul R. Wilson:
Texas: Good, Fast, Cheap Persistence in C++, in: ACM SIGPLAN OOPS, Vol.4, No.2, Seite 145-147, 1993.

[Sörgaard 88] Paal Sörgaard:
Object Oriented Programming and Computerised Shared Material, in: ECOOP '88 Proceedings, K. Nygaard et al., Springer, Heidelberg, 1988.

[Sousa et al. 96] P. Sousa, A. R. Silva, J. A. Marques:
Naming and Identification in Distributed Systems: A pattern for Naming Policies, in: PLoP '96 Proceedings, Allerton Park, 1996.

[Stürner 93] Günther Stürner:
Oracle7 - Die verteilte semantische Datenbank, dbms Publishing, Weissach, 1993.

[Surber 95] D. Surber:
Unnatural Acts - Object Applications talking to Relational Databases, White Paper, Roth Well Intl., 1995.

[Suzuki et al. 94] Shinji Suzuki, Masaru Kitsuregawa, Mikio Takagi:
An Efficient Pointer Swizzling Method for Navigation Intensive Applications, in: Persistent Object Systems, Malcolm Atkinson et al., Springer, Heidelberg, 1995.

[Turner, Keller 95] P. Turner, A. M. Keller:
Reflections on Object-Relational Applications, in: OOPSLA '95 Proceedings, Austin, 1995.

[Universal 79] *Grosses Universal Lexikon in Farbe - Illustriertes Wissen in 10 Bänden*, Edition Thomas, Herrsching, 1979.

[Universal 82] *Grosses Universal Lexikon*, Honos Verlagsgesellschaft, Zug, 1982.

[Vaughan, Dearle 92] Francis Vaughan, Alan Dearle:
Supporting Large Persistent Stores using Conventional Hardware, in: Persistent Object Systems, Antonio Albano et al., Springer, Heidelberg, 9/1992.

[Wai 89] Francis Wai:
Distributed PS-algol, in: Persistent Object Systems, John Rosenberg et al., Springer, Heidelberg, 1989.

[Weiser 89] Udo Weiser:
Ein objekt-orientiertes Modell für verteilte Anwendungen, in: Objekte in verteilten Systemen, Seminar SS 89, Fakultät Informatik, Universität Karlsruhe, 1989.

[Wetzel 94] Ingrid Wetzel:
Programmieren mit STYLE: Über die systematische Entwicklung von Programmierumgebungen, Dissertation, Fachbereich Informatik, Universität Hamburg, 1994.

[Wilson, Kakkad 92] Paul R. Wilson, Sheetal V. Kakkad:
Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware, in: IEEE Press., Workshop on Object Orientation in Operating Systems, Seite 364-377, 1992.

[Wulf, Weske 94] M. Wulf und D. Weske:
Konzepte zur Materialversorgung verteilter Werkzeugumgebungen am Beispiel einer objektorientierten Datenbank, Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1994.

[X3J16 97] X3 Technical Committee J16:
X3J16 Annual Report, Vorveröffentlichung des C++-Standards von 1997, http://www.x3.org/tc_home/tc_annreps/x3j16ar.htm, 1997.

[Züllighoven 94] Heinz Züllighoven:
Objektorientierte Systementwicklung, Teil A: Entwurf, Skriptum zur gleichnamigen Vorlesung, Fachbereich Informatik, Universität Hamburg, 1994.