Studienarbeit

Persistente softwaretechnische Archive zur kooperationsunterstützenden Verwaltung behälterartiger Materialien

vorgelegt von

Michael Otto
Töpfertwiete 36
21029 Hamburg
3otto@informatik.uni-hamburg.de

Norbert Schuler
Alter Teichweg 91
22049 Hamburg
3schuler@informatik.uni-hamburg.de

Oktober 1998

Betreuung: Dr. Guido Gryczan

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

„Blessed shall be thy basket and thy store.“

Genesis, xxviii. 5.

Inhalt:

1 EINLEITUNG 5

1.1 ÜBERSICHT 5
1.2 DANKSAGUNG 6

2 EINFÜHRUNG IN GRUNDLEGENDE KONZEPTE VON WAM 7

2.1 WAM 7
2.1.1 DIE ENTWURFSMETAPHER WERKZEUG 7
2.1.2 DIE ENTWURFSMETAPHER MATERIAL 8
2.1.3 DIE ENTWURFSMETAPHER AUTOMAT 8
2.2 JWAM 9

3 GRUNDLAGEN UND MOTIVATION EINES SOFTWARETECHNISCHEN ARCHIVS 11

3.1 UMGANGSFORMEN, KONSTRUKTIONSPRINZIPIEN UND ARBEITSGEGENSTÄNDE EINES SOFTWARETECHNISCHEN ARCHIVS 12
3.2 KOOPERATION 13
3.3 BEHÄLTERARTIGE MATERIALIEN IM ARCHIV 14
3.4 ERSTER ANSATZ EINES ARCHIVS 16
3.5 EIGNUNG EINES EINFACHEN ARCHIVS FÜR DIE VERWALTUNG BEHÄLTERARTIGER MATERIALIEN 17

4 KONSTRUKTIONSASPEKTE EINES ARCHIVS FÜR BEHÄLTERARTIGE MATERIALIEN 19

4.1 KOOPERATIONSMODELLE UND METHODEN DES KONKURRIERENDEN ZUGRIFFS 19
4.2 STRUKTUR BEHÄLTERARTIGER MATERIALIEN AUßERHALB UND INNERHALB DES ARCHIVS 21
4.3 ERHALT DER FACHLICHEN EIGENSCHAFTEN EINES BEHÄLTERARTIGEN MATERIALS BEI DER ARCHIVIERUNG 22
4.4 VERSCHRÄNKTES AUSLEIHEN BEHÄLTERARTIGER MATERIALIEN 23
4.5 WIEDERERKENNUNG IN VERTEILTEN UMGEBUNGEN 26
4.6 EINHALTUNG DES VERTRAGSMODELLS IM ARCHIV 27
4.7 PERSISTENZ DES ARCHIVS 28
4.8 ZUSAMMENFASSUNG 29

5 PERSISTENZ MIT PJAMA 31

5.1 KURZE EINFÜHRUNG IN DIE PERSISTENZ-THEMATIK UND PERSISTENTE PROGRAMMIERSPRACHEN 31
5.1.1 ORTHOGONALE PERSISTENZ 31
5.1.2 PERSISTENTE OBJEKTORIENTIERTE PROGRAMMIERSPRACHEN UND STRATEGIEN FÜR PERSISTENZ 33
5.2 ORTHOGONALE PERSISTENZ MIT JAVA: PJAMA 34
5.2.1 STANDORT VON PJAMA IM MARKT UND ANWENDUNGSGEBIETE VON PJAMA 35
5.2.2 ENTWURFSASPEKTE VON PJAMA 36
5.2.3 KRITERIEN ORTHOGONALER PERSISTENZ IN ANWENDUNG AUF PJAMA 39
5.2.4 ARCHITEKTUR DES PJAMA-SYSTEMS 39
5.2.5 EINSCHRÄNKUNGEN DES PJAMA-PROTOTYPS 41

6 IMPLEMENTATION UND DOKUMENTATION EINES PERSISTENTEN ARCHIVS FÜR BEHÄLTERARTIGE MATERIALIEN 42

6.1 STRUKTUR DES HIERARCHIVE 42
6.1.1 DIE SCHNITTSTELLE FÜR ARCHIVIERBARE MATERIALIEN 42
6.1.2 DAS MATERIAL ARCHIV 44
6.1.3 REGISTRIERUNG EINES BENUTZERS (REGISTER, UNREGISTER) 46
6.1.4 MIT DEM ARCHIV ARBEITEN (GETARCHIVEID, GETDIRECTORY) 47
6.1.5 DIE BESTANDLISTE UND DIE AUSLEIHLISTE 49
6.1.6 MATERIALIEN INS ARCHIV EINFÜGEN (ADDENTRY) 50
6.1.7 MATERIALIEN AUS DEM ARCHIV AUSLEIHEN (BORROWENTRY) 51
6.1.8 MATERIALIEN INS ARCHIV ZURÜCKSTELLEN (GIVEBACKENTRY) 51
6.1.9 MATERIALIEN IM ARCHIV FREIGEBEN (FREEENTRY) 53
6.1.10 MATERIALIEN IM ARCHIV LÖSCHEN (REMOVEENTRY) 53
6.1.11 FACHLICHE ANTEILE BEARBEITEN (GETDPSID) 54
6.1.12 KOOPERATION: AUSLEIHLISTEN ABFORDERN (GETBORROWERS) 55

6.1.13 KOOPERATION: NACHRICHTEN VERSCHICKEN 55
6.2 INTEGRATION VON CLIENT-/SERVER-FÄHIGKEIT: RMI 57
6.3 INTEGRATION VON PERSISTENZ DURCH SERIALISIERUNG 60
6.4 INTEGRATION VON PERSISTENZ MIT PJAMA 61
6.4.1 ANWENDUNG DER ENTWICKLUNGSUMGEBUNG VON PJAMA 61
6.4.2 ERWEITERUNGEN IN DER KONSTRUKTION DES ARCHIV 62
6.4.3 GARBAGE COLLECTION 64
6.5 PJAMA IM ZUSAMMENSPIEL MIT RMI 64
6.5.1 ERWEITERUNGEN ZUR DATENSICHERUNG UND BEENDIGUNG DES STORES 65
6.5.2 ZUSAMMENSPIEL MIT MESSAGING-MECHANISMUS DES FRAMEWORKS JWAM 66
6.5.3 PROBLEME MIT EINER NICHT PERSISTENTEN REGISTRY 67
6.6 BEURTEILUNG DER VERWENDBARKEIT VON PJAMA 68
6.7 BEURTEILUNG DES HIERARCHIVE BEZÜGLICH DER ENTWURFSZIELE 68

7 ANHANG 71

7.1 LITERATURVERZEICHNIS 71

1 Einleitung

Die vorliegende Arbeit behandelt das Problem der kooperationsunterstützenden Verwaltung behälterartiger Materialien in Archiven, auf das die Autoren im Rahmen des Projektseminars „Java, Corba, WWW“ im Fachbereich Informatik der Uni Hamburg aufmerksam wurden. Ziel der Arbeitsgruppe[1] der Autoren innerhalb dieses Seminars war die Konstruktion einer verteilten Anwendung zur gemeinsamen Bearbeitung von Dokumenten, wie sie üblicherweise bei der Softwarekonstruktion nach dem Werkzeug & Material-Ansatz [Züllighoven 98] verwendet werden. Hierzu zählen z.B. Glossare und Szenarien als Ergebnisse von Interviewserien [Floyd 97, Züllighoven 98]. Den Umgang und die Probleme damit haben die Autoren im einjährigen Softwaretechnik-Projekt *HIPPO* kennengelernt.

Schätzte die Arbeitsgruppe das im folgenden abgehandelte Problem zunächst noch als relativ geringfügig ein, so führten die gruppeninternen Diskussionen darüber nach und nach zu immer neuen, komplexeren Teilaspekten dieses Gesamtproblems, welches sich schließlich zu einem eigenen Projekt verselbständigte. Schließlich trat die Thematik der Persistenz innerhalb des Problems stärker in den Vordergrund, so daß diese einen größeren Teil der vorliegenden Arbeit ausmacht.

Da für das Verständnis dieser Arbeit grundlegende Konzepte aus dem Werkzeug & Material-Ansatz (*WAM*) notwendig sind, gibt Kapitel 2.1 hierzu einen kurzen Überblick. Das am Arbeitsbereich Softwaretechnik in der Sprache Java entwickelte Rahmenwerk *JWAM* orientiert sich an den WAM-Metaphern und war Ausgangspunkt des genannten Projektseminars. Deshalb ist diese Arbeit von Anfang an eng mit JWAM verbunden gewesen. Kapitel 2.2 gibt einen kurzen Einblick in JWAM.

1.1 Übersicht

Im ersten Teil dieser Studienarbeit werden die zugrundeliegenden Konzepte des Werkzeug & Material-Ansatzes prinzipiell und in praktischer Umsetzung durch JWAM beschrieben. Danach wird der Einsatzkontext softwaretechnischer Archive beschrieben, deren Umgangsformen hergeleitet und implementationsübergreifende Konstruktionsprinzipien erläutert. Schließlich wird die Notwendigkeit der Entwicklung eines speziellen Archivs für behälterartige Materialien begründet.

Der zweite Teil ab Kapitel 4 geht auf die zu erwartenden Probleme bei den verschiedenen Konstruktionsmöglichkeiten für das Archiv ein. Dabei werden verschiedene Lösungsansätze vorgestellt und sich ergebende Folgeprobleme diskutiert. Die Darstellung beschränkt sich hierbei auf prinzipielle und konstruktionsbedingte Schwierigkeiten und bietet verschiedene

[1] bestehend aus Niels Fricke, Oliver Kempe, Sven Lammers, Carola Lilienthal, Eike Steffens, Norbert Schuler, Michael Otto

Lösungsideen an. Für die Konstruktion einer idealen Lösung werden schließlich Entwicklungsziele eingegrenzt.

Den Kern dieser Arbeit bildet ab Kapitel 5 die Vorstellung einer konkreten Implementation eines persistenten Archivs auf der Basis der Software *PJama*. Hierzu wird zunächst die Thematik Persistenz eingeführt und PJama vorgestellt. Schließlich werden die auftretenden prinzipiellen und speziellen Probleme der Implementation diskutiert und eine Bewertung anhand der Entwicklungsziele vorgenommen.

1.2 Danksagung

Für ihre persönliche Unterstützung bei technischen Problemen mit der Implementation von PJama und besonders PJRMI danken die Autoren Susan Spence von der Universität Glasgow, die zum Entwicklerteam von PJama zählt, Bernd Mathiske von Sun Laboratories, der ebenfalls an PJama beteiligt ist und zuvor wissenschaftlicher Mitarbeiter am Fachbereich Informatik der Universität Hamburg war, sowie Frank Griffel vom Arbeitsbereich VSYS, der uns in einem längeren Gespräch viel Einblick in die Hintergründe und Ausblicke von PJama verschaffte.

Außerdem schulden die Autoren den genannten Mitgliedern der Arbeitsgruppe „WAM-Dokumente“ aus dem Projektseminar „Java, Corba, WWW“ Dank, mit denen die Grundlagen softwaretechnischer Konstruktionsalternativen für Archive erarbeitet wurden.

2 Einführung in grundlegende Konzepte von WAM

Eine gründliche Einführung in den Werkzeug & Material-Ansatz (WAM) kann an dieser Stelle nicht geleistet werden; der geneigte Leser sei hier auf [Züllighoven 98] verwiesen, an dessen Darstellung sich dieses Kapitel anlehnt. Im folgenden sollen nur die für das Verständnis dieser Arbeit notwendigen Grundzüge von WAM skizziert werden.

2.1 WAM

WAM ist ein Methodenrahmen zur Entwicklung objektorientierter Software. Der wesentliche Punkte dabei ist der, daß WAM keine starre Anleitung im Sinne einer Sammlung von Dogmen für die Softwareentwicklung darstellt, sondern vielmehr eine Sicht auf den Vorgang der Entwicklung selbst. Dies nennt man eine *Methode zweiter Ordnung* oder – wie oben – Methodenrahmen.

WAM bietet also eine Anleitung, wie zu einem Problem eine konkrete Konstruktionstechnik und eine dazu passende Vorgehensweise gefunden werden kann. Hierzu bietet WAM einen Satz langjährig erprobter Konstruktions-, Analyse- und Dokumentationstechniken und ineinander passende Konzepte.

Eine der wichtigsten Ideen von WAM ist es, die Gegenstände und Prinzipien des Anwendungsbereichs als Basis für das softwaretechnische Modell zu begreifen. Die Begrifflichkeiten der Anwendungswelt fließen so in die Architektur der Software ein. Dieser Umstand wird *Stukturähnlichkeit* genannt.

Bei der Umsetzung von Strukturähnlichkeit helfen dem Anwendungsentwickler nach WAM *Leitbilder* und *Entwurfsmetaphern*. Ein Leitbild ist ein griffiges Motiv, das dem Anwender und dem Entwickler hilft, Software zu verstehen und zu entwerfen. Ein Beispiel für ein Leitmotiv ist der *Arbeitsplatz für qualifizierte und eigenverantwortliche Tätigkeiten*. Eine Entwurfsmetapher beschreibt dagegen einen konkreten Gegenstand oder ein Konzept des Anwendungssystems durch einen Gegenstand der Alltagswelt [Züllighoven 98].

Der Werkzeug & Material-Ansatz hat seinen Namen nach zwei seiner wichtigsten Entwurfsmetaphern, dem Werkzeug und dem Material. Das A in WAM steht für eine weitere wichtige Metapher, den Automaten.

2.1.1 Die Entwurfsmetapher Werkzeug

Mit Werkzeugen verändert oder sondiert der Mensch Materialien. Dabei kann ein Werkzeug in wechselnden Kontexten und für verschiedene Materialien benutzt werden. Werkzeuge vergegenständlichen wiederkehrende Arbeitshandlungen [Züllighoven 98].

Dieses Bild von Werkzeugen läßt sich auch auf softwaretechnische Werkzeuge übertragen. Dabei ist jedoch die direkte Übertragung von Form und Handhabung handwerklicher Werkzeuge selten sinnvoll; hier muß ein eigener Weg gefunden werden.

In Softwareprodukten, die nach dem Werkzeug & Material-Ansatz entwickelt wurden, hat der Benutzer schließlich einen Satz teils hochspezialisierter Software-Werkzeuge zur Verfügung, mit denen er „Materialien", also letztlich Arbeitsergebnisse, bearbeitet und produziert.

Dieses Bild wird sofort anschaulich, wenn man sich als Beispiel ein Zeichenprogramm vorstellt mit der Malfläche als Material und der (Software-)Sprühdose als Werkzeug. Gewöhnlich ist der Abstraktionsgrad der Metaphern Material und Werkzeug jedoch höher.

2.1.2 Die Entwurfsmetapher Material

Materialien sind, wie oben angedeutet, Teile des Arbeitsergebnisses. Sie werden von Werkzeugen oder Automaten bearbeitet und haben einen anwendungsfachlichen Hintergrund. Die Eigenschaften von Materialien lassen sich meist sinnvoll auf ihre softwaretechnischen Entsprechungen übertragen [Züllighoven 98].

Für diese Arbeit hat noch eine spezielle Ausprägung von Materialien eine besondere Bedeutung: der Behälter.

Behälter fassen unterschiedliche Materialien zu einem neuen fachlichen Ganzen zusammen. Sie können eine gewisse Konsistenz der Gesamtheit der in ihnen enthaltenen Materialien wahren. Dazu führen sie oft Inhaltsverzeichnisse; an ihnen kann erkannt werden, ob und welche Materialien aus dem Behälter entnommen wurden.

Oft repräsentieren Behälter eine Ordnung, etwa durch Register oder Sortierung der enthaltenen Materialien. Ein Beispiel für einen Behälter im verwendeten Bild des Zeichenprogramms wäre eine Bildermappe. Behälterartige Materialien werden in Kapitel 3.3 noch einmal genauer untersucht.

2.1.3 Die Entwurfsmetapher Automat

Automaten erledigen ihre Aufgabe mit wenigen Einstellmöglichkeiten „auf Knopfdruck" und ohne äußere Eingriffe. Ihre Aufgabe ist meist eine Routinetätigkeit oder ein festgelegter Prozeß, dessen Ergebnis genau bekannt ist [Züllighoven 98].

Auch Automaten arbeiten also auf Materialien, bieten aber dem Benutzer meist keine Interaktion außer einem An-/Aus-Schalter. Ein Beispiel im Rahmen des Zeichenprogramms wäre ein Automat, der im Abstand von 10 Minuten das aktuelle Bild sichert. Auch die „Undo"-Funktion kann als Automat aufgefaßt werden.

2.2 JWAM

JWAM ist ein am Arbeitsbereich Softwaretechnik im Fachbereich Informatik der Uni Hamburg in der Sprache Java entstandenes Rahmenwerk, das die Softwareentwicklung nach dem Werkzeug & Material-Ansatz unterstützt. Der Name JWAM steht für die Sprache Java und die bereits eingeführten Charakteristika Werkzeug, Automat und Material.

JWAM ist aus dem C++ Rahmenwerk *BibV30* (siehe [Weiß 97]) hervorgegangen, dabei wurden aber lediglich Konzepte übertragen – JWAM ist von Grund auf eigenständig konstruiert worden. Nach der ersten Version im Oktober 1997 erfolgte eine Neuordnung der Package-Struktur zur Version 1.1 im April 1998. Waren die Packages zuvor rein aufgabenbezogen unterteilt (vgl. [Lippert 97]), so wurde das Framework in der neuen Aufteilung primär nach der Schichtenarchitektur (siehe unten) und in den Ästen wiederum aufgabenbezogen strukturiert. Die Version 1.2 ist für den Jahreswechsel 1998/1999 anvisiert. Vom Ansatz her ist JWAM ein Application-Framework, das auch die Entwicklung komplexer Anwendungen unterstützt.

Im wesentlichen ist JWAM nach der sogenannten U-Architektur (siehe [Bäumer 98]) mit drei aufgabenteilenden Schichten strukturiert (siehe Abbildung 1): der Handhabungs- und Präsentationsschicht, die beispielsweise Klassen zur Konstruktion von Werkzeugen und zur Verwaltung einer grafischen Oberfläche bereithält, der Technologieschicht, welche technische Hilfsmittel wie den Nachrichtendienst anbietet (auch ein Persistenzmechanismus würde hier angesiedelt werden), sowie der Systembasisschicht, in der grundlegende Konzepte wie das Vertragsmodell (siehe [Meyer 90], [Fricke 98]), Reaktionsmechanismen oder die Container-Klassenbiliothek jConLib (siehe [Bohlmann 98]) angesiedelt sind. Die Einteilung in Schichten äußert sich bei der JWAM-Konstruktion in der hierarchischen Aufteilung des Rahmenwerks in die Packages `handling`, `technology` und `system`.

Die Dokumentation von JWAM ist zum Zeitpunkt der Erstellung dieser Arbeit in der Entstehung (siehe [JWAM]); an dieser Stelle näher auf konkrete Konstrukte von JWAM einzugehen, würde aber kaum Sinn machen. Der für diese Arbeit relevante Teil des Frameworks, der Kommunikationsdienst, wird in Kapitel 6.1.13 eingeführt.

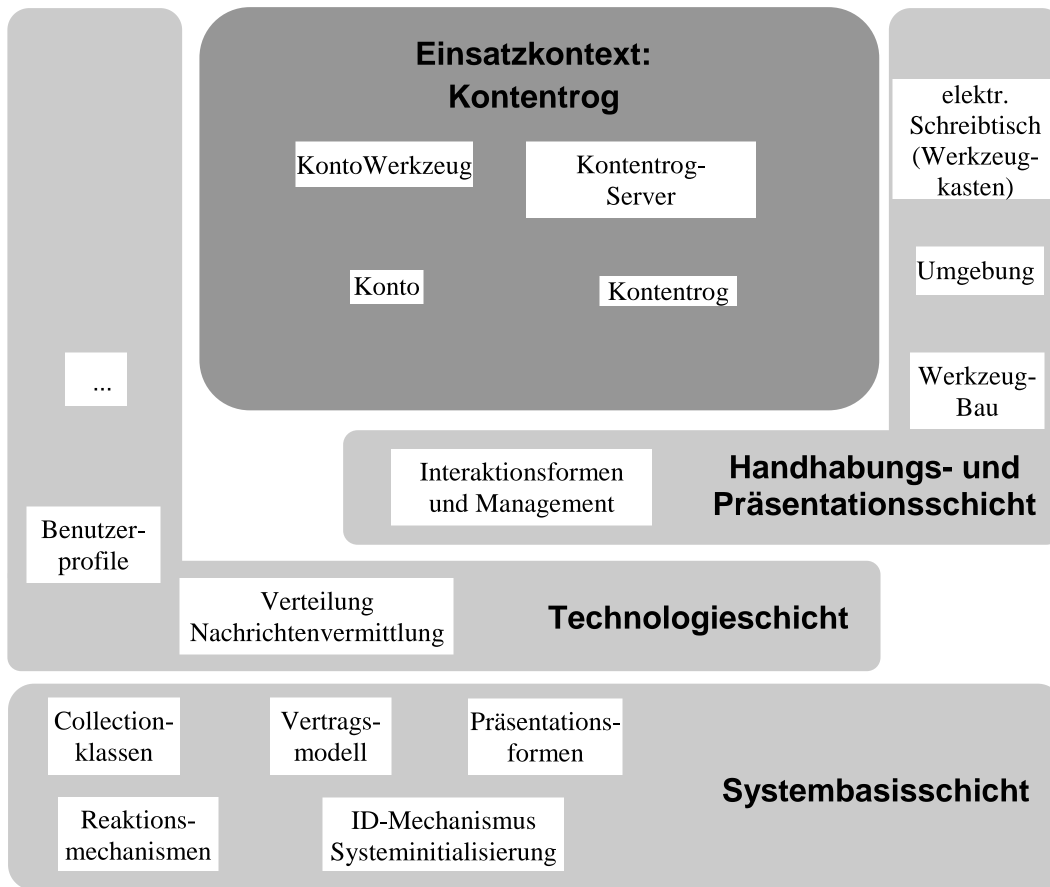

Abbildung 1: Schichten des JWAM-Rahmenwerks

(Zeichnung von Carola Lilienthal und Martin Lippert)

3 Grundlagen und Motivation eines softwaretechnischen Archivs

Als illustrierendes Beispiel in dieser Arbeit soll an einigen Stellen die Bearbeitung von Planetten (grob: zusammengefaßte Dokumente zu einem Patienten) in Krankenhäusern verwendet werden, da diese sich gut eignet, um behälterartige Materialien in Archiven zu beleuchten. Die vorgestellten Konstruktionen sind aber nicht auf diesen Einsatzkontext beschränkt, sondern möglichst universell gehalten. Daher sind zum Verständnis der folgenden Ausführungen Detailkenntnisse über die Handhabung von Patientendokumenten nicht notwendig; eine grobe Kenntnis des Themenkomplexes Material – Dokument - Behälter (siehe Kapitel 2.1.2, 3.3) reicht hierzu aus.

Die Bearbeitung von Dokumenten eines Patienten ist ein Prozeß, der in Teamarbeit erfolgt. Dies bedeutet, daß mehrere Benutzer – in diesem Falle Schwestern oder Pfleger – gleichzeitig oder versetzt an einer gemeinsamen Aufgabe arbeiten. Dazu müssen sie sich nach bestimmten Regeln verständigen und auch immer wieder gemeinsam verwendete Materialien austauschen. Die verschiedenen Arten, in denen Benutzer miteinander kooperieren können, nennt man Kooperationsmodelle [Züllighoven 98]. Die für diese Arbeit in Frage kommenden Modelle werden in Kapitel 3.2 vorgestellt.

Die bei der gleichzeitigen Bearbeitung von Dokumenten, z.B. Kurvenblättern oder Pflegeberichten, typischerweise auftretenden Probleme sind

- das gegenseitiges Überschreiben von Änderungen
- der wechselseitige Ausschluß bei der Bearbeitung
- verschiedene Versionen eines Dokuments liegen bei verschiedenen Benutzern
- die gegenseitige Absprache bei der Aufteilung der anfallenden Arbeit

Zur Verwaltung gemeinsam genutzter Dokumente soll ein Mittel verwendet werden, das sich der kritischen Koordinierungsprobleme annimmt. Dieses Kooperationsmittel wird im folgenden als *Archiv* bezeichnet.

Der Zweck eines softwaretechnischen Archivs ist es also, Materialien zu verwalten und allen Benutzern bereitzustellen. Weiterhin muß das Archiv die Koordination der Benutzer im Umgang mit den Materialien ermöglichen, also insbesondere die genannten Konfliktfälle ausschließen.

Bevor Lösungsansätze und die damit verbundenen Schwierigkeiten erläutert werden, soll im folgenden wesentliche Konstruktionsprinzipien eines softwaretechnischen Archivs erläutert werden.

3.1 Umgangsformen, Konstruktionsprinzipien und Arbeitsgegenstände eines softwaretechnischen Archivs

Die Herleitung der Umgangsformen eines softwaretechnischen Archivs geschieht in Anlehnung an den intuitiven Umgang mit einem realen[2] Archiv.

In einem Archiv gibt zunächst einmal die Möglichkeit, neue Materialien im Archiv zu erfassen. Hierzu ordnet der *Archivar* des Archivs dem Material eine für das Archiv eindeutige Kennzeichnung zu, die er in seiner Bestandsliste einträgt und anhand derer er dem Material einen Platz im Archiv zuordnet.

Daneben können im realen Archiv Materialien ausgeliehen werden. Diese werden vom Archivar anhand eines Titels, Autors o.ä. der archivinternen Kennzeichnung zugeordnet und einem Benutzer ausgehändigt. Hierbei kann jedoch der Fall auftreten, daß das Material bereits ausgeliehen ist, und der Archivar dem Benutzer nur die Nichtverfügbarkeit des Materials mitteilen kann.

Andere relativ einsichtige Umgangsformen eines realen Archivs sind z.B. *Zurückgeben eines ausgeliehenen Materials*, *Nachfrage über den Verbleib eines Materials*, *Ausgeben eines Inhaltsverzeichnisses des Archivs* oder *Entfernen eines Dokuments aus dem Archiv*. Weitere Umgangsformen sind denkbar, bergen aber keine Erkenntnisse über neue Konstruktionsprinzipien.

Als erste Erkenntnis soll festgehalten werden, daß sich das reale Archiv intuitiv als aus zwei Teilen bestehend darstellt: dem eigentlichen Archiv und einer darüber wachenden Instanz, dem Archivar. Die Nachbildung der Aufgaben eines Archivars aus dem realen Archiv ist auch in der softwaretechnischen Konstruktion nach WAM-Gesichtspunkten sinnvoll. Der Archivar bildet dabei eine Kontrollinstanz, die die Integrität des Archivs bei Mehrbenutzerzugriff gewährleisten kann und die Funktionalität des Archivs von dessen Material trennt.

Die Trennung des Archivs vom Archivar ist auch aus anderen Gründen heraus zweckmäßig: Zum ersten ist das Archiv allein sinnvoll verwendbar – beispielsweise, wenn sich ein Benutzer das gesamte Archiv vom Archivar herausgeben lassen will oder wenn es sich um eine Lösung für einen Einzelplatz handelt. Zum zweiten wäre durch diese Trennung damit auch ein verteiltes Archiv denkbar, bei dem ein zentraler Archivar auf verschiedenen Rechnern gespeicherte Archive anspricht.

Als weitere Erkenntnis stellt sich ein reales Archiv ganz selbstverständlich als eine zentrale Anlaufstelle für alle Benutzer dar. Daß dies in Rechner-Umgebungen keineswegs so selbstverständlich ist, zeigt sich schon aus der Tatsache, daß zur Schaffung einer zentralen Anlaufstelle zunächst einmal alle Arbeitsplätze verbunden werden müssen.

---

[2] Der hier verwendete Archiv-Begriff entspricht nicht dem Konzept eines Archivs aus den Verwaltungswissenschaften. Er ist als sehr intuitiver Zugang zu verstehen.

Die Vorteile einer zentralen Anlaufstelle für alle Benutzer liegen sowohl im realen Archiv als auch in verteilten Umgebungen auf der Hand: Daten können ohne Redundanz gespeichert werden, und Anfragen über Verfügbarkeit und Umfang sowie Anforderungen von Daten müssen nur an eine einzige Instanz gestellt werden. Nachteile wie mangelnde Risikoverteilung und fehlende Redundanz für Datensicherheit und Verfügbarkeit können hier vernachlässigt werden, da sie hardwaretechnisch minimiert werden können (z.B. über RAID-Systeme[3] oder Backup-Leitungen).

Als weiteres ganz wesentliches Charakteristikum eines Archivs muß dessen Langlebigkeit oder *Persistenz* (siehe Kapitel 5) herausgestellt werden, auch wenn diese aus dem obigen Beispiel nur indirekt hervorgeht. Aber schon die Beschreibung „Aufbewahrungsort“ für ein Archiv impliziert dessen Dauerhaftigkeit. Ein sinnvoller Einsatz für transiente Archive ist hingegen nur schwer vorstellbar. Ein softwaretechnisches Archiv muß deshalb in ihm gespeicherte Materialien dauerhaft, also über mehr als einen Programmlauf hinweg, aufbewahren.

Außer den bisher abgeleiteten strukturellen Ideen können auch Arbeitsgegenstände wie Kennzeichnungen, Inhaltsverzeichnisse, Benutzer und Bestandslisten aus dem realen Archiv aufgegriffen und für das softwaretechnische Archiv angepaßt werden. Wie dies konkret aussehen könnte, zeigt Kapitel 6.

Neben den sinnvollen Grundprinzipien, die bei der Konstruktion des Archivs übernommen werden können, erfordert aber vor allem die Umsetzung eines Kooperationsmodells die Entwicklung eigener Umgangsformen und Architekturen für die softwaretechnische Realisation.

3.2 Kooperation

Die folgende Darstellung lehnt sich an die Beschreibung in [Züllighoven 98] an, um noch einmal kurz die Grundbegriffe zum Verständnis von Kooperation zu erläutern:

„Ein Kooperationsmodell ist ein Modell, das entweder explizit oder implizit die Zusammenarbeit (Kooperation) beschreibt und regelt.“ [Züllighoven 98]

In einem Kooperationsmodell wird die Art der kooperativen Zusammenarbeit beschrieben, die immer aus einer Koordination zwischen den beteiligten Benutzern besteht, also einer Zusammenarbeit zum Erreichen eines gemeinsamen Ergebnisses. Die Koordination umfaßt die gegenseitige Abstimmung von Arbeitsteilung und beruht auf wechselseitigen Konventionen oder ausdrücklichen Regeln.

---

[3] Redundant Array of Independent Disks: parallel geschaltete Festplattensysteme mit Datenredundanz als Sicherung gegen Datenverlust

Implizite Kooperation bedeutet, daß mehreren Benutzern der konkurrierende Zugriff auf Materialien ermöglicht und verdeutlicht wird, aber eine Kooperation und Koordination nicht gegenständlich verwirklicht ist.

Bei expliziter Kooperation ist für jeden Benutzer sichtbar, mit welchen anderen Benutzern er kooperativ zusammenarbeitet. Für Materialweitergabe und Koordination stehen ihm in den Arbeitsplatz integrierte Kooperationsmittel und -medien zur Verfügung.

Das Archiv stellt hierbei sowohl Kooperationsmittel als auch –medium dar. Es ist Kooperationsmittel, weil über das Archiv Rückschluß auf Entleiher von Materialien möglich ist. So können sich die Benutzer zumindest außerhalb des Systems koordinieren. Es ist Kooperationsmedium, weil über das Archiv Materialien zwischen den einzelnen Arbeitsplätzen ausgetauscht werden können.

Ein für die Kooperation mit Hilfe eines persistenten softwaretechnischen Archivs ebenfalls zentraler Aspekt ist die Frage, ob auf das Material als Original oder als Kopie zugegriffen wird. Folgende Arten des konkurrierenden Zugriffs auf die Materialien sind denkbar:

- exklusiver Zugriff auf das Material; es gibt keine Kopien
- exklusiver Zugriff auf ein Original; Kopien sind möglich
- Zugriff nur auf Kopien möglich; Kopien können als Originale zurückgestellt werden

In den nachfolgenden Kapiteln werden Ansätze von Archiven mit Zugriff nur auf Kopien der verwalteten Materialien dargestellt. Die Probleme mit dem Entnehmen von Originalen werden in Kapitel 4.1 skizziert.

3.3 Behälterartige Materialien im Archiv

Ein softwaretechnisches Archiv dient zur kooperativen Verwaltung von Materialien. Im folgenden sollen deshalb Materialstrukturen genauer untersucht werden, um sie für ein Archiv handhabbar machen zu können.

Grundsätzlich wurden einfache Materialien und behälterartige Materialien schon in Kapitel 2.1.2 eingeführt. Um zu einem übergreifenden Ansatz für die Klassifizierung von Materialien zu gelangen, wird in der Arbeitsgruppe JWAM am Arbeitsbereich Softwaretechnik zur Zeit über eine Einordnung von Beziehungstypen diskutiert. Als vorläufiges Ergebnis wurden folgende Beziehungstypen herausgearbeitet:

- Über einen (Quer-)*Verweis*[4] referenziert ein Objekt ein anderes. Die Kopplung ist sehr schwach.

---

[4] Mit Verweis ist in diesem Fall nicht eine technische Referenz (Pointer) gemeint, sondern ein Querverweis auf ein im Zusammenhang stehendes Objekt.

- Über die *Enthält-Beziehung* wird festgelegt, daß ein Objekt ein anderes enthält.
- Die *Besteht-Aus-Beziehung* ist stärker als die Enthält-Beziehung, da keine weiteren Referenzen auf das benutzte Objekt erlaubt sind.
- Die *Service-Beziehung* ermöglicht, daß ein Objekt die Dienste anderer Objekte in Anspruch nehmen kann.

Als behälterartige Materialien nach dieser Terminologie werden im folgenden Materialien bezeichnet, welche andere Materialien in Sinne der obigen Enthält- und Besteht-Aus-Beziehung benutzen können. Dabei bilden einfache Materialien einen Spezialfall von behälterartigen Materialien.

Materialien, die auch über andere Beziehungstypen als Enhält- oder Besteht-Aus verknüpft sind, sollen als komplexe Materialien bezeichnet werden. Sie sind nicht Gegenstand dieser Arbeit. Momentan werden alle obigen Beziehungstypen technisch gleich mit Referenztypen realisiert, so daß komplexe Materialien noch nicht adäquat abgebildet werden können. Eine technische Nachbildung der unterschiedlichen Beziehungstypen könnte dieses Problem lösen.

Ein Beispiel für behälterartige Materialien sind die betrachteten Pflegedokumente im Krankenhaus. Eine Pflegebericht ist ein behälterartiges Material, welches als einfache Materialien Pflegeberichtseinträge enthält. Eine Planette ist ebenfalls ein behälterartiges Material, welches unter anderem Pflegeberichte, Kurvenblätter, Visitenbögen usw. enthalten kann. In Abbildung 2 wird ersichtlich, daß behälterartige Materialien hierarchisch aufgebaut sind und eine Baumstrukur bilden.

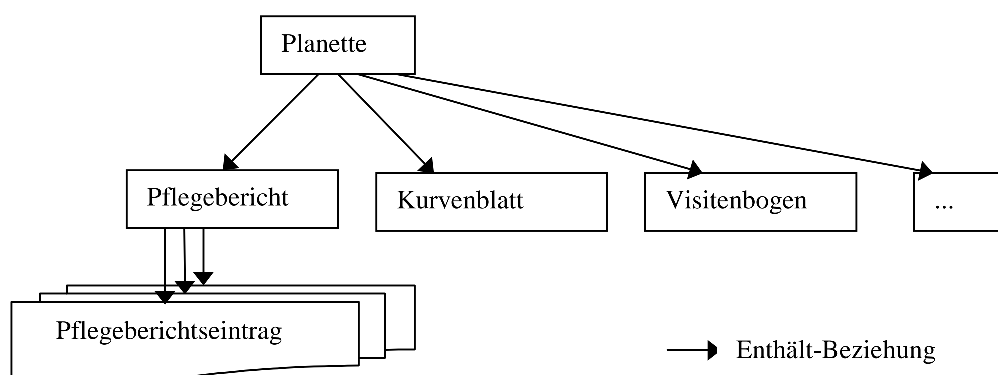

Abbildung 2: Behälterartige Materialien bilden eine Baumstruktur

Nun kann man ein solches behälterartiges Material sehr leicht zu einem komplexen Material verwandeln, beispielsweise indem man fordert, daß Pflegeberichtseinträge auf Einträge im Pflegemaßnahmenbogen verweisen können müssen.

Die Handhabung der zyklischen Struktur komplexer Materialien in einem persistenten softwaretechnischen Archiv führt zu Komplikationen, die den Rahmen dieser Studienarbeit bei weitem sprengen. Deshalb beschränkt sich die restliche Arbeit auf die Archivierung hierarchischer oder behälterartiger Materialien.

3.4 Erster Ansatz eines Archivs

Für einen Ansatz eines einfachen Archivs sollen die wesentlichen Forderungen an Funktionalität und Architektur eines softwaretechnischen Archivs umgesetzt werden. Diese Forderungen umfassen die Realisation des Archivs als zentrale Anlaufstelle, die Trennung zwischen dem Automaten Archivar und dem Material Archiv und die aus dem realen Archiv übernommenen und an das verwendete Kooperationsmodell angepaßten Umgangsformen. Auf die Realisation von Persistenz wird zunächst verzichtet, da diese später für die jetzt vorgestellten Konstruktionen transparent integriert wird.

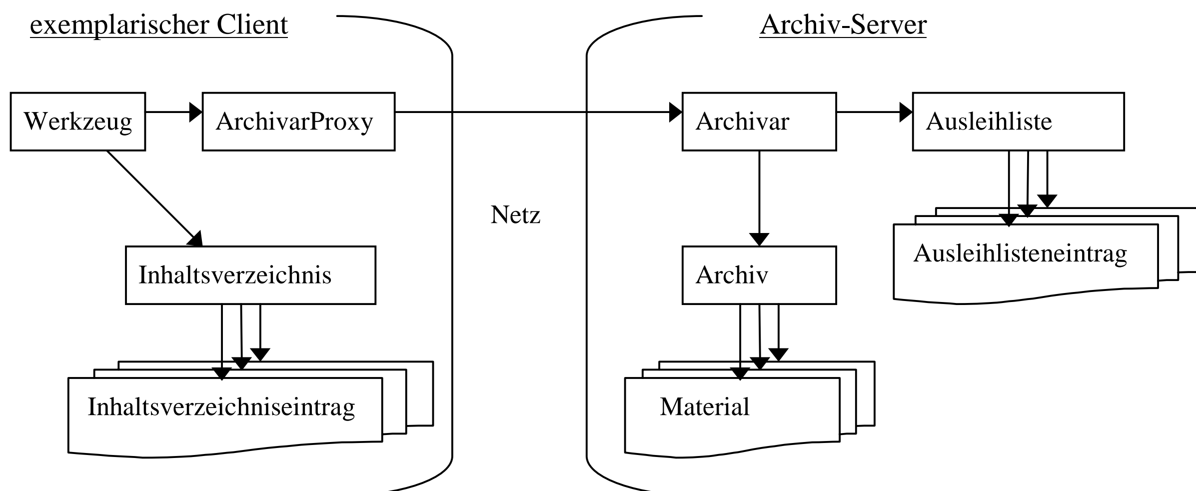

Abbildung 3: Architektur eines einfachen Archivs

Wesentliche Umgangsformen des Archivars sind (nach Kapitel 3.1):

- `GibAuskunftÜberAusleiher(Kennzeichen) -> Liste von Ausleihern`
- `GibInhaltsverzeichnis() -> Inhaltsverzeichnis`
- `LeiheKopieVonMaterialAus(Kennzeichen, Benutzer)`
- `StelleKopieVonMaterialAlsOriginalZurück(Kennzeichen, Material)`
- `ErfasseMaterial(Material)`

Durch den Entwurf des Archivs als Client-/Server-Lösung mit Archivar und Archiv auf Serverseite und den benutzenden Werkzeugen auf Clientseite ist die gewünschte zentrale Anlaufstelle realisiert.

Der Archivar und das Archiv sind als voneinander getrennte eigenständige Objekte verwirklicht. Der Archivar benutzt das Material Archiv und kapselt es nach außen.

Für die Verwendung durch die Werkzeuge in der Client-/Server-Architektur bildet ein nach dem Proxy-Muster [Gamma et al. 96] gebautes Objekt den lokalen Stellvertreter des

Archivars auf Clientseite. Dabei wird keine fachliche Funktionalität zum Archivar hinzugefügt oder ausgeblendet, sondern nur der Netz-Zugriff gekapselt.

Die Umgangsformen sind dem gewählten Kooperationsmodell mit seinem nur-Kopie-Zugriffsmechanismus angepaßt. Der Benutzer fordert zunächst das Inhaltsverzeichnis des Archivs beim Archivar an und sucht sich in diesem das gewünschte Material mit dessen Kennzeichen heraus. Eine Kopie dieses Materials kann er sich nun vom Archivar geben lassen. Im Gegensatz zum realen Archiv ist dies durch die nur-Kopie-Logik immer möglich.

Schwierigkeiten handelt man sich dafür beim Zurückstellen einer veränderten Kopie als neues Original in das Archiv ein, denn das vorhandene Original könnte in der Zwischenzeit von einem anderen Benutzer verändert oder gelöscht worden sein. Ein bedingungsloses Zurückstellen würde dann vorgenommene Änderungen überschreiben. Dieses Problem wurde bereits zu Beginn dieses Kapitels als zentral erkannt und ist bei einem kooperationsunterstützenden Archiv unbedingt zu verhindern.

Deshalb kann das Zurückstellen in obigem Modell fehlschlagen, nämlich genau im geschilderten Fall. Um trotzdem eine Kooperation zu unterstützen, kann der Benutzer gegebenenfalls die Liste der Ausleiher eines Materials anfordern und sich mit diesen koordinieren.

Weitere benötigte Umgangsformen wie An- und Abmelden als Benutzer sowie das erstmalige Erfassen von Materialien im Archiv sollen an dieser Stelle nicht weiter ausgeführt werden, da sie für das Modell keine neuen Erkenntnisse bringen.

3.5 Eignung eines einfachen Archivs für die Verwaltung behälterartiger Materialien

Sicherlich wird man den einfachen Ansatz eines Archivs in der oben geschilderten Weise oder leicht abgewandelt einsetzen können. Dabei darf man jedoch nicht übersehen, daß im Ansatz des einfaches Archivs keine Annahmen über die Struktur der verwalteten Materialien getroffen werden. Dies mag in kleineren Anwendungen noch ohne Probleme funktionieren, in größeren Anwendungen wird man jedoch zunehmend auf Materialien stoßen, die eine Struktur wie die in Kapitel 3.3 beschriebenen hierarchischen Materialien aufweisen.

Versucht man nun, solchen behälterartigen Materialien mit dem einfachen Archiv gerecht zu werden, so stößt man auf Probleme, die anhand des folgenden Beispiels deutlich gemacht werden sollen:

Die Station eines Krankenhauses verwaltet die Planetten seiner Patienten in einer Planettenliste, die von den Arbeitsplätzen der Station aus bearbeitet werden können soll. Die Rechner sind vernetzt und die Station verwendet ein Archiv obigen Zuschnitts, um die Planettenliste zentral zu speichern.

- Wenn ein Pfleger die Planette eines Patienten bearbeiten möchte, muß er dazu die gesamte Liste aus dem Archiv nehmen, da er im einfachen Archiv nicht direkt auf das einzelne Planetten-Objekt zugreifen kann. Dieses ist nämlich in der archivierten Planettenliste enthalten und dem Archivar deshalb nicht bekannt. Allgemein bietet das einfache Archiv keinen Zugriff auf in einem archivierten Objekt enthaltene Objekte. Stattdessen muß immer das oberste enthaltende behälterartige Material aus dem Archiv ausgeliehen werden. Abbildung 4 verdeutlicht diesen Sachverhalt.
- Wenn zwei Krankenschwestern gleichzeitig Planetten aus der Liste bearbeiten wollen, behindern sie sich gegenseitig, da beide die komplette Liste ausgeliehen haben, und die zuletzt speichernde Schwester ihre Planettenliste nicht wieder in das Archiv zurückstellen kann. Dies ist auch dann der Fall, wenn beide Krankenschwestern verschiedene Planetten bearbeiten. Je umfangreicher die behälterartigen Materialien sind, die im einfachen Archiv gespeichert werden, um so häufiger können gegenseitige Behinderungen auftreten, die durch das Archiv-*Modell*, nicht aber durch die Veränderung gleicher einfacher Materialien oder gleicher Äste behälterartiger Materialien, also wirklicher Zugriffskollisionen, begründet sind.

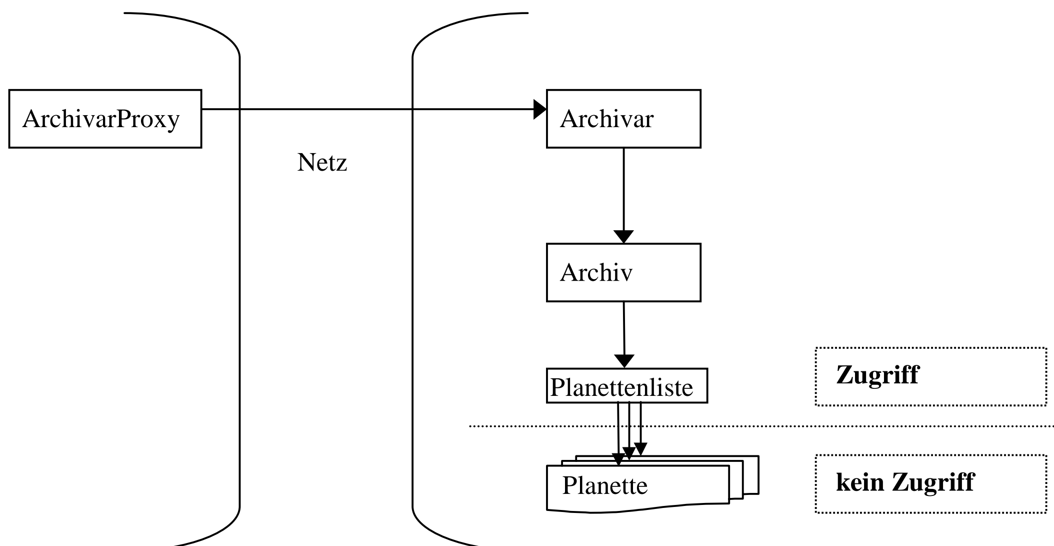

Abbildung 4: behälterartige Materialien im einfachen Archiv

Zusammenfassend findet man beim einfachen Archiv einige gute Ansätze verwirklicht. Das hauptsächliche Manko ist aber die Tatsache, daß der Archivar jedes Material wie ein einfaches Material behandelt. Insgesamt wird dieser Ansatz damit behälterartigen Materialien nicht gerecht. In dieser Arbeit soll deshalb ein spezielles Archiv für die Verwaltung behälterartiger Materialien konstruiert werden.

Im folgenden Kapitel sollen deshalb Lösungsansätze, die behälterartige Materialien für den Archivar handhabbar machen, eingeführt werden. Außerdem wird die bisher grobe Modellierung des Archivs verfeinert und angepaßt dargestellt.

4 Konstruktionsaspekte eines Archivs für behälterartige Materialien

Im folgenden Kapitel werden die verschiedenen bei der Konstruktion eines Archivs für behälterartige Materialien auftretenden Probleme behandelt, verschiedene mögliche Lösungs*ideen* aufgezeigt und Entwurfsziele eines idealen Archivs abgesteckt.

4.1 Kooperationsmodelle und Methoden des konkurrierenden Zugriffs

Ein persistentes Archiv als softwaretechnische Lösung verkörpert nicht per se eines der Kooperationsmodelle impliziter oder expliziter Kooperation. Vielmehr stellt es die grundlegende Architektur zur Verwaltung kooperativ genutzter Materialien zur Verfügung. Das letztlich realisierte Kooperationsmodell ist Sache der das Archiv benutzenden Werkzeuge und Automaten.

Dennoch müssen einige grundlegende Entscheidungen für Eigenschaften der unterstützten Kooperation im voraus getroffen werden. Eine davon ist die Art des konkurrierenden Zugriffs auf Materialien im Archiv, wie in Kapitel 3.2 aufgelistet.

Der exklusive Zugriff auf das Original-Material ermöglicht immer nur einem Benutzer zur Zeit, ein Material zu sichten und zu bearbeiten. Alle anderen Benutzer sind bei dieser Form der Zugriffsverwaltung vom Zugriff auf ausgeliehene Materialien ausgeschlossen und müssen gegebenenfalls lange Wartezeiten während ihrer Arbeit hinnehmen. Vergißt der ausleihende Benutzer, das Material nach der Bearbeitung wieder freizugeben, oder er löscht es und stellt es absichtlich nicht zurück ins Archiv, so kann das Material überhaupt nicht mehr erreicht werden. Auf der anderen Seite ist dem ausleihenden Benutzer der sichere und exklusive Zugriff auf das Material garantiert, so er es denn aus dem Archiv entnehmen kann. Er muß sich keinerlei Gedanken über etwaige Koordination machen; dieses Problem liegt hier bei den konkurrierenden Benutzern.

Man beschränkt also den Anwendungsbereich des softwaretechnischen Archivs bei ausschließlicher Verwendung des exklusiven Zugriffs auf solche Materialien, die nicht oft benutzt werden oder für die das Anfertigen von Kopien fachlich nicht sinnvoll ist.

Ist den anderen Benutzern zusätzlich zur Arbeit mit dem Original noch der Zugriff auf Kopien dieses Materials gestattet, so können Probleme wie Wartezeiten vermieden werden. Jedoch benötigen Benutzer, die sich Kopien eines Materials ausgeliehen haben und nicht mit dessen ausschließlicher Betrachtung zufrieden sind, eine neue Koordinationsform, um beim Zurückstellen des Originals informiert zu werden, damit sie ihrerseits das Original-Material ausleihen können.

Generell gilt: Erlaubt man auch das Anfertigen von Kopien, so hat man das Problem, Originale von Kopien unterscheiden zu müssen. Im Anwendungsraum findet man also sowohl Original-Materialien wie auch beliebige Kopien von ihnen.

Nun lassen sich zwar softwaretechnische Konventionen finden, um diese Unterscheidung vornehmen zu können, etwa, indem man Kopien nur über eine bestimmte Operation erzeugt und sie danach kennzeichnet; unabsichtlich aber können sehr wohl technische Kopien (deep copies) angefertigt werden, die das Archiv nicht mehr von den Originalen unterscheiden kann. Die wichtigste Aufgabe des Archivs, die konsistente Verwaltung von Materialien, könnte dieses mithin nicht mehr erfüllen.

Ein ähnliches Problem gilt auch für den exklusiven Zugriff nur auf Original-Materialien: Auch hier kann nicht verhindert werden, daß eben doch Originaldoubletten außerhalb des Archivs erzeugt werden, die sich beim Zurückstellen eventuell gegenseitig überschreiben, da sie ununterscheidbar sind.

Die in dieser Arbeit vorgestellten Implementationen von Archiven geben dem Benutzer nur Kopien von Materialien heraus. Diese Methodik wird im folgenden als *nur-Kopie-Logik* bezeichnet.

Oben genannte Schwierigkeiten der Unterscheidung von Originalen und fachlichen bzw. technischen Kopien treten bei der nur-Kopie-Logik nicht auf. Hier werden Originale immer im Archiv verwahrt – der Benutzer bekommt stets nur eine Kopie. Dabei handelt man sich neue Probleme ein, doch diese sind anderer Natur.

Das Archiv muß bei der nur-Kopie-Logik nunmehr verhindern, daß sich zwei Benutzern beim Zurückstellen ihrer veränderten Kopien als neue Originale gegenseitig überschreiben. Dies kann durch Zeitstempel realisiert werden, die das Datum der letzten Änderung eines Materials bei Herausnahme aus dem Archiv für jeden Benutzer festhalten. Die Verwaltung dieser Stempel kann entweder vom Material selbst oder vom Archiv übernommen werden.

Beim Zurückstellen überprüft das Archiv dann, ob die Kopie des Benutzers das gleiche Datum trägt wie das im Archiv verbliebene Original-Material. Ist dies der Fall, so kann der Benutzer zurückschreiben und das Original-Material erhält einen aktuellen Zeitstempel.

Wenn nicht, so ist eine *Zugriffskollision* aufgetreten – der Benutzer darf nicht zurückschreiben, weil das Original inzwischen von einem anderen Benutzer verändert worden ist. Da es keine allgemeine Lösung zur Regelung solcher Fälle geben kann, müssen sich die Benutzer außerhalb des Archivs koordinieren. Für die Zwischenzeit bietet das Archiv die Möglichkeit, das geänderte Material als neues Original ins Archiv aufzunehmen.

Als Alternative zur Verwendung von Zeitstempeln können auch Versionsnummern verwendet werden, beispielsweise wenn Zeitstempel in zeitkritischen Anwendungen nicht mehr genau genug sind. Bei dieser Versionslogik wird für jedes archivierte Material eine Versionsnummer geführt und diese beim Zurückstellen einer Kopie erhöht.

	fachliche Problemfälle		technische Problemfälle	
Zugriffsmethode	**Ausleihen eines Materials**	**Zurückstellen eines Materials**	**Unterscheidung von Originalen und Kopien**	**Erkennung von Originaldoubletten**
...nur Originale	Zugriffskollisionen möglich (beherrschbar)	problemlos	---	problematisch
...Originale und Kopien	Zugriffskollisionen möglich (beherrschbar)	Zugriffskollisionen möglich (beherrschbar)	problematisch	problematisch
...nur Kopien	problemlos	Zugriffskollisionen möglich (beherrschbar)	---	---

Tabelle 1: Mögliche Problemfälle im Archiv bei unterschiedlichen Methoden des konkurrierenden Zugriffs

Zusammenfassend birgt eine Zugriffslösung mit Originalen viele eher technische Probleme, während die nur-Kopie-Logik fachliche Hürden stellt, die aber einigermaßen leicht überwunden werden können. Tabelle 1 stellt die Schwachpunkte und Stärken der verschiedenen Zugriffslösungen gegenüber.

4.2 Struktur behälterartiger Materialien außerhalb und innerhalb des Archivs

Die Handhabbarkeit behälterartiger Materialien durch den Archivar wurde in Kapitel 3.5 als wesentliche Voraussetzung für eine sinnvolle Modellierung des zu konstruierenden persistenten Archivs begründet. Da der Archivar nun aber nicht die Struktur jedes möglichen behälterartigen Materials kennen kann, sollen Umgangsformen gefunden werden, die von jedem Material implementiert werden müssen, wenn es von dem in dieser Arbeit vorgestellten Archiv gespeichert werden soll.

In Sprachen mit Mehrfachvererbung bietet sich hierfür eine Oberklasse für archivierbare Materialien an, die die Umgangsformen von archivierbaren Objekten schon teilweise implementiert.

Die in dieser Arbeit verwendete Sprache Java unterstützt allerdings keine Mehrfachvererbung, so daß das Erben von einer Oberklasse für archivierbare Materialien nicht sinnvoll ist, da es andere fachliche Vererbungen unmöglich macht. Die alternative Lösung ist eine bloße Schnittstelle (*Interface*) für alle zu archivierenden Materialien.

Festzuhalten bleibt in jedem Fall, daß – im Gegensatz zum einfachen Archiv – ein archivierbares Material ein speziell angepaßtes Material ist. Bereits vorhandene Materialien können also nicht ohne Änderung vom Archiv verwaltet werden.

Um auf in behälterartigen Materialien enthaltene Objekte zugreifen zu können, braucht der Archivar eine Sicht in das Material. Ein vom Material erzeugtes Inhaltsverzeichnis ist eine

Möglichkeit dazu. Anhand von dessen Einträgen muß der Archivar dann Objekte aus dem Material lesen, schreiben und löschen etc. können. Alternativ kann das Material eine Iterationsschnittstelle anbieten, mit deren Hilfe der Archivar die Einträge des Materials untersuchen und bearbeiten kann.

Die Erhaltung der behälterartigen Struktur ist aber keine unbedingte Voraussetzung für die Konstruktion eines Archivs für behälterartige Materialien. Ein internes „Aufbrechen“ der Materialien und eine Verwaltung, die nur die vorhandene behälterartige Materialien simuliert, ist ebenfalls denkbar. Diese Lösung würde unterliegenden einfachen Persistenzmechanismen wie z.B. relationalen Datenbanken zugute kommen, da die Hierarchie des ursprünglichen Materials dann nicht mehr Teil des archivierten Materials wäre; ein Wegspeichern in einfachen Tabellen wäre mithin wesentlich leichter.

4.3 Erhalt der fachlichen Eigenschaften eines behälterartigen Materials bei der Archivierung

Als eine Möglichkeit der Strukturierung von Materialien in einem Archiv wurde im letzten Kapitel der Zugriff über eine definierte Schnittstelle eingeführt. Werden Materialien archiviert, so kann der Archivar über diese Schnittstelle mit den Materialien arbeiten. Allerdings stellt diese Schnittstelle nun die einzige Möglichkeit des Zugriffs auf die archivierten Materialien dar; andere fachliche Umgangsformen als die für alle archivierbaren Materialien gültigen bleiben dem Archivar und damit letztlich dem Benutzer des Archivs verwehrt.

Um bei dieser Form der Archivierung überhaupt eine Möglichkeit zu haben, auf die fachlichen Anteile archivierter Materialien zuzugreifen, ohne das Material ganz aus dem Archiv zu nehmen, gibt es nun mehrere Möglichkeiten.

Zum einen ist eine Abtrennung eines *fachlichen Anteils* eines Objekts vom restlichen Objekt denkbar. Als fachlicher Anteil soll derjenige Teil des behälterartigen Materials bezeichnet werden, der fachliche Funktionalitäten und Eigenschaften des Materials umsetzt, im Gegensatz zum restlichen Teil, der die Behälterfunktionalität realisiert.

Ein Beispiel für ein Objekt dieser Art wäre eine Planette, das viele andere Dokumente enthält. Als fachlicher Anteil soll hier ein Objekt *PlanetteDSP* abgetrennt werden, welches die fachlichen Methoden, etwa zum Setzen und Überprüfen der Reiter der Planette, implementiert. Dieser fachliche Anteil wird in einer Besteht-Aus-Beziehung (siehe Kapitel 3.3) mit dem Behälter verbunden und kann über die in der allgemeinen Schnittstelle definierten Umgangsformen über den Archivar aus dem Archiv genommen, bearbeitet und wieder zurückgestellt werden.

Diese Abtrennung wäre nicht mehr nötig, wenn man eigene netzweite Referenztypen (siehe Kapitel 3.3) einsetzt, die mit einem behälterartigen Objekt nicht auch alle enthaltenen Objekte mit über das Netz transferieren.

Eine andere Möglichkeit, eine Abtrennung zu umgehen und trotzdem noch auf den fachlichen Anteil von Objekten im Archiv zugreifen zu können, ist es, außerhalb der Kontrolle des Archivars mit den Materialien im Archiv zu arbeiten.

Dies kann zum einen mit dynamischen Methodenaufrufen am Archivar geschehen, so denn die verwendete Sprache dies unterstützt. Der Archivar ruft dann die gewünschten Methoden direkt und ohne Kenntnis ob deren Wirkung am Material auf und reicht die Ergebnisse ohne Kenntnis ihrer Bedeutung an den Benutzer weiter. Zum zweiten ist auch die Arbeit mit Material-Proxies [Gamma et al. 96] oder anderer indirekter Objekte ein gangbarer Weg, Zugriff auf Objekte im Archiv außerhalb der Kontrolle des Archivars vorzunehmen.

In diesen beiden letzen Fällen kann der Archivar die Aufgaben Koordination und Kontrolle von Zugriffskonflikten nicht mehr aufrechterhalten, was die Wahrung der Archivkonsistenz einschränkt und den praktischen Einsatz dieser Methoden zumindest fragwürdig macht.

Demgegenüber erhöht eine Abtrennung des fachlichen Anteils die Komplexität der Konstruktion archivierbarer Materialien, sichert aber die Integrität und die Fähigkeit des Archivs, als Kooperationsmittel und -medium zu dienen und Zugriffskonflikte zu behandeln.

4.4 Verschränktes Ausleihen behälterartiger Materialien

Durch die Handhabung behälterartiger Materialien durch das Archiv können bei Ausleihe und Zurückgabe von (Teil-)Materialien spezielle Konstellationen auftreten, deren Folgen und sinnvolle Behandlung hier untersucht werden müssen.

Speichert man behälterartige Materialien im Archiv, und werden anschließend von einem Benutzer der gesamte Behälter und von einem anderen Benutzer ein darin enthaltenes Material ausgeliehen (hier als *verschränktes Ausleihen* bezeichnet, siehe Abbildung 5), bearbeitet und wieder zurückgestellt, können folgenden Fälle auftreten:

- Der erste Benutzer ändert die Kopie des einfachen Materials im ausgeliehenen Behälter und stellt diesen zurück, der zweite Benutzer ändert das gleiche aber einzeln ausgeliehene einfache Material und stellt es ebenfalls zurück.
- Der erste Benutzer ändert das einfache Material im Behälter nicht, sondern nur den Behälter und stellt diesen zurück. Der zweite Benutzer stellt sein einfaches Material zurück.
- Der erste Benutzer ändert im Behälter ein anderes Material als das des zweiten Benutzers und stellt den Behälter dann zurück. Der zweite Benutzer stellt sein einfaches Material auch zurück.

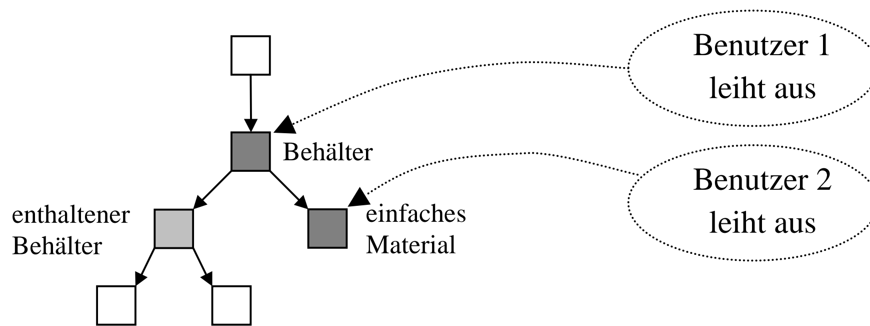

Abbildung 5: Beispielhafte Material-Struktur im Archiv

Da das Archiv keinerlei Kontrolle über die lokalen Aktivitäten des ersten Ausleihers hat und deshalb nicht wissen kann, ob und wieviel am ausgeliehenen Behälter geändert wurde, wenn dieser zurückgestellt wird, muß der Archivar den Zeitstempel des Behälters und aller enthaltenen Materialien aktualisieren, wenn dieser zurückgestellt wird. Damit wird das Zurückstellen des einzeln ausgeliehenen einfachen Materials unmöglich gemacht.

Ebenso muß der Archivar, wenn der zweite Benutzer zuerst das einzeln ausgeliehene einfache Material zurückstellt, auch die Behälter, in denen dieses enthalten ist, als aktualisiert kennzeichnen, um ein Überschreiben durch das Zurückstellen des Behälters durch den ersten Benutzer vermeiden zu können.

Daher gilt für alle obigen Szenarios, daß immer die erste zurückgestellte Änderung zählt, da sie in beiden Fällen den Zeitstempel des anderen Materials auch aktualisiert, und die zweite Zurückstellung verweigert wird, auch wenn beide Benutzer nicht direkt auf demselben Material gearbeitet haben.

Allgemein bedeutet dies, daß wenn ein Material zurückgestellt wird, alle Behälter, in denen es im Archiv direkt oder indirekt enthalten ist als aktualisiert gekennzeichnet werden, und wenn es selbst ein Behälter ist, auch alle darin direkt oder indirekt enthaltenen Materialien:

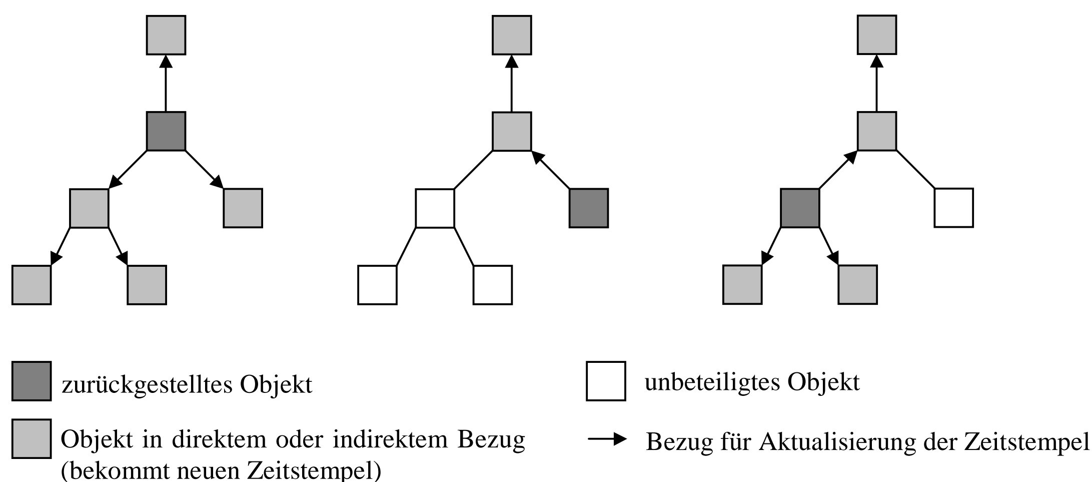

Abbildung 6: Aktualisierung der Zeitstempel beim Zurückstellen in drei Konstellationen

Ohne Konflikt beim Zurückstellen können zwei Benutzer allerdings auf zwei einzelnen ausgeliehenen Materialien in einem gemeinsamen Behälter arbeiten und diese zurückstellen, da diese Materialien ihre Zeitstempel nach obiger Logik nicht gegenseitig aktualisieren können und beim Zurückstellen auch mit der erweiterten Betrachtung verschränkter Ausleihe nur der Zeitstempel des zurückzustellenden Objekts Ausschlag über Erfolg oder Ablehnung gibt.

Eine Alternative zur strikten Ablehnung nicht mehr aktueller Versionen mit der Zeitstempellogik bildet eine Logik zum Zusammenfügen („merge“) der Datenstrukturen, die im Konfliktfall bei behälterartigen Materialien alle direkt oder indirekt enthaltenen Objekte mit denen des Originals im Archiv vergleicht und alle möglichen Aktualisierungen mit der zurückgestellten Kopie durchführt, beispielsweise wenn der Behälter nur aktualisiert wurde, weil eines seiner Einzelobjekte aktualisiert wurde, aber andere Einzelobjekte noch nicht aktueller als die zurückzustellende Kopie sind.

Ob ein solches Zusammenfügen allerdings sinnvoll für den Benutzer ist, ist nicht sicher. Er könnte z.B. eine explizite Abfrage bekommen, ob er zusammenfügen möchte und, wenn er sich dazu entscheidet, einen ausführlichen Bericht über die vorgenommenen Änderungen erhalten.

Verwandt mit dem Problem verschränkten Ausleihens ist das Löschen von Materialien, wenn übergeordnete oder enthaltene Materialien ausgeliehen sind. In diesem Fall macht das Zurückstellen keinen Sinn mehr, und entweder das Löschen oder das Zurückstellen muß daher in diesem Fall vom Archivar verhindert werden.

Es bieten sich hier zwei Vorgehensweisen an: Die erste Möglichkeit ist es, das Löschen von Materialien zu verbieten, falls ein in Bezug stehendes Material oder das Material selbst ausgeliehen ist. Dann können grundsätzlich keine Materialien gelöscht werden, die irgendein Benutzer noch in Bearbeitung oder Ansicht hält.

Die zweite Möglichkeit ähnelt der oben gezeigten Logik des Zurückstellens: Das Löschen ist grundsätzlich erfolgreich, jedoch schlägt danach das Zurückstellen der Kopie des eigentlichen, übergeordneten oder enthaltenen Materials fehl. Dies wird durch die Aktualisierung der Zeitstempel der übergeordneten Materialien genau wie bei Änderungen durch Zurückstellen erreicht.

4.5 Wiedererkennung in verteilten Umgebungen

Das in dieser Arbeit besprochene Archiv soll in einer verteilten Umgebung eingesetzt werden. In der momentanen Entwicklungsstufe von Sprachen, so auch in Java, gibt es noch keine Objektidentitäten und Referenzen, die in einer solchen Umgebung eindeutig sind, und auch im Java-Framework JWAM, in dessen Rahmen die Implementation des hier vorgestellten Archivs durchgeführt wurde, sind noch keine netzweit eindeutigen Referenzen oder Objektidentitäten verwirklicht[5]. Daher kann zur Zeit weder die Sprache noch das Framework zur Identifizierung von Objekten in einer verteilten Umgebung verwendet werden.

Damit das Archiv die herausgegebenen Materialien und deren Kopien über das Netz verfolgen und deren Identität feststellen kann, wenn ein ausgeliehenes Material ins Archiv zurückgestellt wird, wurde das Konzept der *Archiv-ID*s entwickelt. Diese IDs werden an jedes zu archivierende Material durch den Archivar vergeben und müssen netzweit eindeutig sein. Dies ist deshalb problemlos realisierbar, weil das Archiv von Anfang an als zentrale Anlaufstelle konzipiert wurde (siehe Kapitel 3.1). Die Vergabe von Archiv-IDs kann lokal bei der Erzeugung des Materials geschehen, indem sich der Erzeuger beim Archivar eine neue Archiv-ID holt. Denkbar ist außerdem die Vergabe durch den Archivar, wenn das Material zum ersten Mal ins Archiv gestellt wird. Die Archiv-ID bleibt dem Material solange zugeordnet und gültig wie sein Original im Archiv verbleibt.

Damit der Archivar mit den Archiv-IDs der Materialien umgehen kann, muß die allgemeine Schnittstelle von archivierbaren Materialien noch um Umgangsformen zur Verwaltung dieser ergänzt werden und der Archivar muß neue Archiv-IDs herausgeben bzw. neu ins Archiv gestellte Materialien mit IDs versehen können.

Neben der Möglichkeit, Archiv-IDs zentral über das Archiv zu verwalten und zu vergeben, sind noch weitere Möglichkeiten vorstellbar. Während ein behälterartiges Material, welches archiviert werden soll, vom Archiv eine eindeutige Archiv-ID zugeordnet bekommt, erteilt es selbst allen seinen enthaltenen Materialien wiederum IDs, die diese in Einheit mit der Archiv-ID des Behälters eindeutig identifizieren. Dabei muß das Material jedoch immer im Zusammenhang mit dem Behälter bleiben, weil es sonst seine Identität verlieren kann.

[5] Im Laufe des Entstehens dieser Arbeit wurden netzweite Identifikatoren in JWAM integriert. Dabei handelt es sich um einen Ansatz, der das Auftreten gleicher Identifikatoren für verschiedene Objekte sehr unwahrscheinlich macht, aber nicht ausschließt.

Angesprochen werden kann das Material dann immer über seine eigene ID plus der Archiv-ID seines Behälters.

Bislang ist das Archiv nur zur Verwaltung von behälterartigen Materialien geeignet, aber die Archivs-ID weisen nun die Richtung der Möglichkeit, komplexe Materialien zu speichern. So spricht prinzipiell nichts dagegen, Archiv-IDs neben ihrer Funktion zur Identifizierung von archivierten Materialien auch als Referenz zwischen Materialien einzusetzen und so den Beziehungstyp „Verweis“ umzusetzen. Die Konsequenzen dieser Überlegung werden hier aber nicht weiter erläutert.

Neben der Wiedererkennung von Materialien ist auch die Wiedererkennung von Benutzern ein wichtiges Problem. Die bisher beschriebenen Lösungsansätze beruhen darauf, daß erkannt werden kann, welcher Benutzer ein Material zurückstellt. Für das Aufspüren von Zugriffskollisionen etwa ist es zentral für den Archivar zu wissen, welcher Benutzer ein Material in welchem Zustand ausgeliehen hat.

Da die Autoren dieser Arbeit damit rechnen, daß eine Benutzerbehandlung über kurz oder lang in das Rahmenwerk JWAM integriert werden muß und wird, wurde für diese Arbeit eine sehr einfache Überbrückungslösung eingesetzt.

Jeder Benutzer wird einfach anhand seines Namens wiedererkannt; dieser Name wird aber weder zentral verwaltet noch überprüft, so daß die in dieser Arbeit besprochenen Realisationen von Archiven noch leicht durch „Fälschen“ von Benutzern überlistet werden können. In allen entsprechenden Methoden wie Ausleihen und Zurückstellen muß dem Archivar der Benutzeraccount zur Wiedererkennung übergeben werden.

Das Wiedererkennen von Benutzern ist sicherlich ein zentrales Thema bei der Betrachtung verteilter Umgebungen; im Gegensatz zur Wiedererkennung von Materialien kann aber hier mit der beschriebenen einfachen Lösung sehr gut gelebt werden, so daß in dieser Arbeit die Benutzerverwaltung nicht weiter diskutiert werden soll.

4.6 Einhaltung des Vertragsmodells im Archiv

Materialien im Sinne des WAM-Ansatzes haben sondierende und den Zustand des Materials verändernde Umgangsformen. Man verwendet die sondierenden Umgangsformen z.B. um zu prüfen, ob bestimmte zustandsverändernde Umgangsformen auf das Material anwendbar sind.

So prüft man etwa bei einer Liste, ob sie Einträge hat, bevor man sich dem ersten Eintrag geben läßt, oder man testet, ob ein in seiner Größe beschränkter Behälter voll ist, bevor man ein Element hineinfügt.

Diese Teilung und Anwendung von Umgangsformen ist in der Softwaretechnik als *Vertragsmodell* [Züllighoven 98] bekannt. Dieses Modell soll im Umgang mit Materialien im Archiv eingehalten werden.

In verteilten Umgebungen allgemein und im betrachteten Archiv im speziellen kann eine solche Aufteilung der Umgangsformen in sondierende und die Werte oder Eigenschaften des Materials verändernde Umgangsformen nicht mehr ohne weiteres aufrechterhalten werden.

Mehrere verschiedene Benutzer können gleichzeitig sondierende Funktionen aufrufen und alle dasselbe Ergebnis erhalten, welches aber nur dem ersten ausführenden Benutzer die anschließende zustandsverändernde Aktion gewährleistet. Damit solche Konflikte verhindert werden können, benötigt das Archiv entweder eine Form von Transaktionsmanagement oder veränderte Umgangsformen.

Formen von Transaktionsmanagement sind aus dem Datenbankbereich bekannt und werden in dieser Arbeit nicht weiter behandelt.

Die „einfache" Lösung ohne Transaktionsmanagement und ohne Berücksichtigung des Vertragsmodells sieht so aus, daß die zustandsverändernden Funktionen keine Garantie auf Durchführbarkeit bekommen und selbst wie die sondierenden Funktionen dem Benutzer Auskunft über Erfolg oder Mißerfolg Ihrer Ausführung zurückgeben.

4.7 Persistenz des Archivs

Arbeitet man mit persistenten objektorientierten Programmiersprachen, so stellt die gewünschte Persistenz der Objekte im Archiv kein wesentliches Problem dar. Bisher zählte die Sprache Java, die in dieser Arbeit verwendet wird, so wie alle kommerziell relevanten Programmiersprachen nicht zu diesem Sprachtypus, so daß unverwandte Lösungen eingesetzt werden mußten, wollte man die Langlebigkeit seiner Daten erreichen.

Die Integration eines beliebigen Persistenz-Mechanismus in die Konstruktion eines softwaretechnischen Archivs stellt an sich noch kein Problem dar. Es gibt im Gegenteil eine Menge möglicher Lösungen.

Ein Beispiel für eine Persistenz-Konstruktion ist die einfache Serialisierung aller Java-Objekte im Archiv und Speicherung in bzw. Auslesen aus einer Datei. Diese setzt aber voraus, daß alle persistent zu machenden Objekte im Archiv die Schnittstelle `Serializable` erfüllen. Dabei vereinfacht zwar die gemeinsame Schnittstelle der Materialien zur Archivierbarkeit die notwendigen Änderungen an Archiv und Material, kann aber einen sichtbaren Eingriff des Persistenz-Mechanismus in die Archiv- und Material-Konstruktion durch die verpflichtende Implementation von `Serializable` nicht vermeiden. Zu Vergleichszwecken wird diese Lösung trotzdem in Kapitel 6.3 demonstriert.

In die gleiche Richtung wie die Serialisierung aller persistenten Objekte gehen Lösungen mit eigenem I/O-Interface für alle Objekte im Archiv. Unter diese fallen auch Lösungen mit relationalen Datenbanken (RDBMS), die explizite I/O-Methoden zur Speicherung der einzelnen Objekte im Archiv benötigen. Hinzu kommt hier, daß eine Veränderung der

Klassenstruktur auch eine Veränderung des Datenbankschemas bedingen würde, wodurch doppelte Pflege der Datenstrukturen nötig wird.

Weitere Lösungen bieten objektorientierte Datenbanken (OODBMS) oder objektrelationale Datenbanken (ORDBMS). Mit dem Thema der Anbindung von Archiven an solche Datenbanksysteme beschäftigen sich einige aktuelle Studienarbeiten des Arbeitsbereichs Softwaretechnik, so daß hier nicht näher darauf eingegangen werden soll.

Seit kurzem [Atkinson et al. 96b] ermöglicht eine Konstruktion, die im wesentlichen eine veränderte Laufzeitumgebung für Java und damit keine Spracherweiterung darstellt, Java als eine persistente objektorientierte Programmiersprache zu benutzen. Diese Konstruktion trägt den Namen *Persistent Java* (*PJama*) und ist Gegenstand des nächsten Kapitels.

4.8 Zusammenfassung

Nachdem nun die einzelnen Entwurfsaspekte diskutiert worden sind, sollen noch einmal die Entwurfsziele für die Konstruktion eines softwaretechnischen Archivs zusammengefaßt werden. Dabei soll auch diskutiert werden, ob es überhaupt möglich ist, ein ideales Archiv nach den Entwurfszielen zu konstruieren oder inwieweit diese realistisch sind.

- Die Konstruktion des Archivs soll möglichst autark sein, aber trotzdem leicht in eine Anwendung einzubetten.

Die Autarkie des Archiv ist sicherlich wünschenswert zur leichteren Verwendbarkeit in verschiedenen Umgebungen. Wenn das Archiv aber Teil eines Frameworks werden soll, verblaßt dieses Entwurfsziel gegenüber der besseren Einbettung in dieses, und es wird Konstruktionen und bestehende Konzepte des Frameworks in seiner eigenen Konstruktion verwenden.

- Der Aufwand zur Anpassung von Materialien zur Speicherung im Archiv soll minimal sein. Am besten wäre, wenn jedes Material ohne Anpassung im Archiv gespeichert werden könnte.

- Das Archiv soll eine Unterstützung von Kooperation gewährleisten. Am besten wäre die Unterstützung aller Kooperationsmodelle und aller Formen des konkurrierenden Zugriffs zur Wahl durch den Anwender.

Hierbei wurde jedoch noch nicht untersucht, ob die Unterstützung mehrerer Kooperationsformen und Zugriffsmodelle überhaupt sinnvoll ist, oder in welcher Weise dies durchführbar wäre. Beispielsweise müßte ja zumindest eine Benutzergruppe immer wieder dasselbe Modell im Umgang mit bestimmten Materialien vorfinden. Solche Fragen sind aber in dieser Arbeit nicht untersucht worden.

- Das Archiv soll persistent sein mit einem leistungsfähigen Persistenzmechanismus, der auch den schnellen Zugriff auf Daten und die Speicherung größerer Datenmengen erlaubt.

- Die Verwendbarkeit des Archivs über ein Netzwerk soll gewährleistet sein, wobei die verwendete Netztechnologie keine Rolle spielen und möglichst unsichtbar sein sollte.

- Die Handhabbarkeit von strukturierten (komplexen) Materialien durch das Archiv soll gegeben sein.

Das letzte Entwurfsziel steht in offensichtlichem Widerspruch zu dem Anspruch, Materialien ohne Anpassung im Archiv speichern zu können. Man muß hier beide Ziele sorgfältig gegeneinander abwägen und eine Kompromißlösung finden, wenn man nicht auf eine der Vorgaben vollständig verzichten kann.

5 Persistenz mit PJama

Persistenz wurde in den vorigen Kapiteln als ein wesentliches Charakteristikum eines softwaretechnischen Archivs herausgestellt. Durch den Kontext dieser Arbeit im Java- und JWAM-Umfeld bot sich der Einsatz von Persistent Java (PJama) geradeheraus an. Eine genauere Betrachtung der Konzepte von Persistenz und PJama im besonderen erscheint für diese Arbeit angebracht, nicht zuletzt, weil PJama gerade erstmals für Studienzwecke verfügbar gemacht wurde.

In diesem Kapitel soll deshalb zunächst auf die Definition, Eigenschaften und Strategien von Persistenz eingegangen werden, bevor PJama als Lösung der Wahl für die Implementation eines persistenten softwaretechnischen Archivs in Entwurfsaspekten und Architektur erläutert wird.

5.1 Kurze Einführung in die Persistenz-Thematik und persistente Programmiersprachen

5.1.1 Orthogonale Persistenz

Für Datenbanken, ganz gleich ob es sich um relationale, objektorientierte oder objektrelationale Systeme handelt, ist Persistenz das schlagende Charakteristikum. Für Programmiersprachen ist Persistenz jedoch ein neuer Anspruch. Typischerweise dient rund ein Drittel des Codes der meisten kommerziellen Programme lediglich dem Verschieben von Daten zwischen den verschiedenen Speichermedien [Atkinson et al. 90]. Gute Persistenzmechanismen können deshalb erheblich zur Steigerung der Effizienz von Programmen beitragen.

Allgemein kann Persistenz als Abstraktion von Speicher angesehen werden, wobei die Unterschiede zwischen Hauptspeicher und externem Speicher aufgehoben werden und nur definiert wird, wie lange ein Objekt verfügbar soll. Die Lebensdauer von Objekten oder Daten sollen also auch über einen Programmlauf hinaus verlängert werden können [Heuer 92].

Atkinson definiert Persistenz in [Atkinson 91] wie folgt:

Definition *Persistenz*:

> [Persistenz ist die] Fähigkeit der Daten (Werte oder Objekte), *beliebige Lebensdauern* (so kurz wie möglich oder so lang wie nötig) anzunehmen [, wobei die Kriterien orthogonaler Persistenz erfüllt sein müssen].

Die Kriterien für orthogonale Persistenz bleiben in dieser ersten Definition noch unvollständig. 1995 formulieren Atkinson und Morrison [Atkinson et al. 1995]:

Aufstellung der *Kriterien orthogonaler Persistenz*: (Übersetzung der Autoren)

- Das Prinzip der Unabhängigkeit von Persistenz: Die Form eines Programms ist unabhängig von der Langlebigkeit der Daten, die es verändert. Programme sehen gleich aus, egal ob sie langlebige oder kurzlebige Daten verändern.
- Das Prinzip der Typ-Orthogonalität: Objekte beliebiger Typen soll das volle Spektrum von Persistenz erlaubt sein. Es gibt keine Spezialfälle, in denen Objekte nicht persistent oder transient gemacht werden können.
- Das Prinzip der Identifikation von Persistenz: Die Methode, wie persistente Objekte erkannt und zur Verfügung gestellt werden können, ist orthogonal zum restlichen System. Eine Methode zur Erkennung persistenter Objekte ist nicht im Typsystem [der Programmiersprache] verankert.

Um nun verständlich zu machen, warum keine der herkömmlichen in Kapitel 4.7 angesprochenen Methoden der Persistentmachung die Kriterien orthogonaler Persistenz erfüllen kann, soll kurz das Problem des „impedance mismatch“ erläutert werden.

Heuer erklärt in [Heuer 92] wie folgt:

Definition *impedance mismatch*:

> [Impedance mismatch ist] das Nicht-Zusammenpassen von Programmiersprachen- und Datenbankkonzepten.
>
> Was nicht zusammenpaßt, sind vor allen Dingen
>
> - das Typsystem der Programmiersprache und das Datenbankmodell, sowie
> - die Paradigmen der Programmiersprache (etwa imperativ) und der Datenbankoperationen (etwa kalkülartig)

Der impedance mismatch läßt sich also nur vermeiden, wenn Programmiersprache und Datenbankmodell auf gleichen Konzepten basieren. Dies gilt jedoch für keine kommerzielle Datenbank, auch keine objektorientierte, da diese immer mehrere Sprachen bedienen, so daß Paradigmenbrüche oder Typkonflikte auftreten müssen. Jedoch läßt sich der impedance mismatch mit objektorientierten Datenbanken im Vergleich zu relationalen Systemen deutlich mindern.

Durch den impedance mismatch muß also zumindest das Prinzip der Unabhängigkeit von Persistenz oder das Prinzip der Typ-Unabhängigkeit als Kriterien für orthogonale Persistenz verletzt sein.

5.1.2 Persistente objektorientierte Programmiersprachen und Strategien für Persistenz

Seit den späten siebziger Jahren hatte M.P. Atkinson an Datenbanken, Persistenz und deren Anwendung geforscht. Atkinson war 1983 einer der Hauptbeteiligten an der Entwicklung der ersten orthogonal persistenten Programmiersprache, PS-Algol.

Persistente Programmiersprachen sind spezielle Abkömmlinge von Sprachen, bei denen eine Datenbank und eine Programmiersprache zur Laufzeit oder – wenn möglich – zur Compilierzeit zu einem einzigen System zusammengefügt werden. Dieser Ansatz wurde zuerst bei der Entwicklung von PS-Algol erforscht, später dann bei Sprachen wie Amber, Galileo, Napier88 und OPAL.

Zu Beginn der neunziger Jahre, als das objektrelationale Datenbankmodell entwickelt wurde, verbreiteten Stonebraker et al. in [Stonebraker et al. 90] die Idee, daß persistente Programmiersprachen auf herkömmliche Datenbank-Engines aufgesetzt werden sollten. Diese sollten durch Compilererweiterungen und ein Laufzeitsystem unterstützt werden.

Diese Strömungen laufen jedoch gegen die heutige Definition orthogonaler Persistenz, denn mit bloßem „Aufbohren" des Datenbanksystems kann das Problem des impedance mismatch nicht gelöst werden. Anders verhält es sich nun bei der 1996 vorgestellten orthogonal persistenten Programmiersprache PJama, die auf Java basiert [Atkinson et al. 96b], und an deren Entwicklung Atkinson maßgeblich beteiligt war.

Die Kriterien orthogonaler Persistenz stellen sehr gut die Unterschiede zwischen persistenten objektorientierten Programmiersprachen und traditionellen Datenbanksystemen heraus. Selbstverständlich muß auch heute einer persistenten objektorientierten Programmiersprache eine (meist nicht query-fähige) objektorientierte Datenbank unterliegen. Diese ist aber derart an die Programmiersprache angepaßt, daß jedes Programm ohne Änderung persistente und transiente Daten speichern kann. Dadurch sind die Prinzipien der orthogonalen Persistenz erfüllt und der impedance mismatch verschwindet. In einem vollentwickelten System haben deshalb Datenbank und Programmiersprache die gleichen Eigenschaften.

Orthogonale Persistenz von Objekten wird oft, auch in PJama, durch das *Persistenz durch Erreichbarkeit*s-Konzept (Erläuterung siehe unten) verwirklicht. Diese Art der Persistentmachung ist ein vollständig anderer Ansatz als der relationaler oder objektrelationaler Datenbanksysteme, die Abfrage- und Definitionssprachen, meist SQL oder eine Weiterentwicklung, zur Navigation und Manipulation innerhalb der Datenbank verwenden [Danielsen 98].

Zur Abrundung sollen hier kurz die wesentlichen tatsächlich verwendeten Persistenz-Strategien mit objektorientiertem Bezug vorgestellt werden:

- Persistenz durch Vererbung: Eine Klasse erbt die Persistenzfähigkeit von einer vordefinierten persistenten Klasse. Ein Exemplar dieser Klasse kann persistent oder

transient sein, aber der Übergang zwischen persistent und transient muß explizit ausgeführt werden. Damit widerspricht diese Strategie dem Prinzip der Identifikation von Persistenz (Persistenz ist Teil des Typsystems) und dem Prinzip der Unabhängigkeit von Persistenz (explizite Operationen nötig). Persistenz durch Vererbung wird z.B. von Versant und Objectivity in deren C++ Anbindung verwendet.

- Persistenz durch Instanziierung: Ein Objekt bekommt seine Persistenzfähigkeit bei der Instanziierung. Diese Strategie widerspricht dem Prinzip der Unabhängigkeit von Persistenz, wird aber tatsächlich z.B. bei der C++ Anbindung an ObjectStore verwendet. Ein weiteres Problem ist das folgende: Was passiert, wenn ein persistentes Objekt auf ein transientes verweist? Um diese Schwierigkeit auszuräumen, muß das Typsystem der Sprache eingeschränkt werden.

- Persistenz durch Erreichbarkeit: Ein Objekt wird persistent, wenn es von einem anderen persistenten Objekt aus (über Referenz-Beziehung) transitiv erreicht werden kann. Diese Strategie, die auch transitive Persistenz heißt, wird in PJama eingesetzt und ist die einzige, die orthogonale Persistenz erlaubt.

Letztere Strategie ähnelt dem Konzept der Serialisierung in Java, so daß die beiden Lösungen in der Implementation eines persistenten softwaretechnischen Archivs technisch relativ leicht gegeneinander ausgetauscht werden können, wie später gezeigt wird.

Interessant an dieser Stelle ist die Tatsache, daß es momentan noch kein objektorientiertes Datenbanksystem (OODBMS) gibt, das Persistenz durch Erreichbarkeit realisiert (und damit orthogonale Persistenz erreichen könnte) und gleichzeitig die Spezifikationen ODMG 2.0 der Object Database Management Group (ODMS) erfüllt [Danielsen 98].

5.2 Orthogonale Persistenz mit Java: PJama

Als Netscape 1995 Java zum ersten Mal in seinen Navigator integrierte, verhalf das der Sprache von Sun Microsystems über Nacht zu großem Interesse und langfristig zu seinem heutigen Erfolg. Die Protagonisten der Persistenz-Bewegung sahen sofort ihre Chance („golden opportunity“, [Atkinson et al. 96a]), den Persistenz-Gedanken in einer kommerziell relevanten Programmiersprache zu verankern.

Schon im Februar 1996 lag von einer Arbeitsgruppe der Universität Glasgow die erste Version eines Papiers über Entwurfsaspekte eines persistenten Java vor [Atkinson et al. 96a]. Von Sun tatkräftig unterstützt gelang innerhalb weniger Monate die Realisierung von $PJava_0$ [Jordan 96], einem ersten Prototypen eines persistenten Java, der dem heutigen PJama schon sehr ähnelt.

Inzwischen wurde PJava in PJama umbenannt und weiterentwickelt. Die aktuelle Version trägt die Nummer 0.5.6.9 und ist nur für akademischen oder evaluierenden Gebrauch von Sun direkt erhältlich.

5.2.1 Standort von PJama im Markt und Anwendungsgebiete von PJama

Die Autoren von PJama sind nicht der Meinung, daß PJama den gesamten Markt an Datenbanken jeder Art überflüssig macht[6], weil ja nun die Haltung persistenter Daten schon in die Programmiersprache integriert wurde. Vielmehr besetzt PJama als orthogonal persistente Programmiersprache nur einen bestimmten Bereich des Marktes, der dauerhafte Datenspeicherung abdeckt.

Die besondere Stärke von PJama ist sicherlich die Effizienz, mit der ein Applikationsprogrammierer dauerhafte Datenspeicherung implementieren kann. Diese Effizienz äußert sich besonders dann, wenn komplexe Objekt-Strukturen persistent gemacht werden müssen und der Bruch zu relationalen oder auch objektorientierten Datenbanken sehr groß wäre – und damit letztlich aufwendig zu implementieren.

Der Effizienzgewinn bei der Programmierung mit orthogonal persistenten Programmiersprachen läßt sich gut mit dem Prinzip der Garbage Collection vergleichen. Es wird Verantwortung vom Programmierer genommen, da er sich nicht mehr um dauerhafte Datenspeicherung oder im anderen Fall Deallokation von Speicher zu kümmern braucht. Auch die Garbage Collection hatte es als Konzept in der Informatikwelt anfangs schwer, weil sich Programmierer der Kontrolle über ihre Programme beraubt sahen.

Konkrete Einsatzgebiete von orthogonal persistenten Programmiersprachen könnten Workflow-Systeme sein, in denen es stark auf die Kooperation von Benutzern ankommt. Hier das ganze System in einem persistenten Zustand zu halten, brächte enorme Vorteile; über die Flüchtigkeit von Materialkopien auf Benutzer-Rechnern bräuchte man sich dann keine Gedanken mehr zu machen. Auch Entwicklungsumgebungen für Programmierer sind ein ideales Einsatzgebiet. Der Programmierer würde dann beim nächsten Systemstart seine Entwicklungsumgebung genau so vorfinden, wie er sie verlassen hat, ohne Kosten für den Programmierer der Entwicklungsumgebung.

Auf dem Informatiksektor läßt sich ein Trend beobachten[7], der in Richtung Persistenz geht – sei es nun im kleinen Rahmen mit PJama oder bis hin zu persistenten Betriebssystemen. Das letzte Ziel ist gar nicht so abwegig, wenn man sich einmal die Betriebssysteme der inzwischen stark verbreiteten Personal Digital Assistants (*PDA*s) ansieht. Hierbei handelt es sich um zwar kleine, aber dennoch schon vollständig persistente Betriebssysteme, bei denen der Benutzer gleich beim Einschalten den letzten Zustand vorfindet. Dies erfordert letztlich

[6] Bernd Mathiske von Sun Laboratories, einer der Beteiligten des PJama-Projekts, in einem Vortrag im Juli 1998 im Fachbereich Informatik

[7] laut Frank Griffel vom Arbeitsbereich VSYS der Informatik an der Uni Hamburg in einem Gespräch im September 1998

aber nicht nur einen Bruch mit den Konzepten von Dateisystemen, sondern ein völlig eigenes Bedienkonzept für zukünftige Betriebssysteme.

Aus diesem Markttrend auf eine rosige Zukunft für PJama zu schließen, wäre aber verfrüht. Aus markttaktischen Überlegungen heraus hat Sun selbst das Datenbanksystem ObjectStore lizensiert, also ein vom Ansatz her konkurrierendes Produkt. PJama läuft nur noch bis Ende 1999 als Forschungsprojekt. Danach wird es eventuell eingestellt; ob es jemals als Version für Endkunden verfügbar sein wird, ist ungewiß.

Daneben darf man die angestammten Einsatzgebiete herkömmlicher Datenbanksysteme nicht vergessen; auf diesen können und sollen orthogonal persistente Programmiersprachen die Datenbanken nicht verdrängen.

Geht es nämlich um die Haltung riesiger Datenmengen, schnelle, viele und selektive Zugriffe auf die Daten mit vielen Verknüpfungen, so werden auch in Zukunft weiterhin relationale oder objektrelationale Datenbanksysteme eingesetzt werden. Auch bei Aufgabenstellungen wie Data Mining werden orthogonal persistente Programmiersprachen keinen Einsatz finden.

5.2.2 Entwurfsaspekte von PJama

Nach Ansicht der Entwickler von PJama eignet sich Java besonders gut für die Verankerung orthogonaler Persistenz, weil das Typsystem wohldefiniert, einfach konstruiert und konsequent umgesetzt ist.

Der Class Loader von Java setzt bereits in der nichtpersistenten Version dynamische Bindung ein, was das Einhängen von neuem Code zur Behandlung persistenter Daten und Programme wesentlich vereinfacht. Außerdem werden Zeiger und Speicher von der Abstract Machine (JAM) implizit behandelt anstatt explizit vom Anwendungsprogrammierer.

Ziel der Entwicklung von PJama war ein orthogonal persistentes System, das folgende Eigenschaften erfüllt [Atkinson et al. 96a]:

- keinen Verlust an Sicherheit bei der Benutzung von Persistenz und im Idealfall einen Gewinn an Konsistenz
- keinen Einfluß auf die Performanz von Programmen, die keine Persistenz benutzen; minimale Einbußen oder, wenn möglich, Performanzsteigerungen bei Programmen, die Persistenz benutzen
- möglichst geringe Änderungen an der Sprache
- einfach zu benutzende Persistenz für die meisten Programmierer, erreicht durch ein Standard-Transaktionenmodell

- leistungsfähige und präzise Kontrolle transaktionaler Programmierung für Programmierer, die diese benötigen

Vorausblickend soll hier angemerkt werden, daß diese Punkte bis auf den letzten relativ zufriedenstellend verwirklicht werden konnten. Ein ausgefeiltes Transaktionenmodell ist für PJama in Planung, aber noch nicht integriert, so daß im folgenden nicht mehr darauf eingegangen wird.

PJama speichert alle persistenten Objekte in einem sogenannten *Store*. Beim Arbeiten mit Persistenz muß daher zunächst einmal ein Store erzeugt und dem persistente Daten haltenden Programm kenntlich gemacht werden. Über die Architektur des PJama-Systems und die Bedeutung des Stores gibt Kapitel 5.2.4 Aufschluß.

Ein ganz wesentlicher Punkt in PJama, der dieses von objektorientierten Datenbanksystemen unterscheidet, ist die einheitliche Behandlung von Daten und Programmcode. Dies bedeutet insbesondere, daß die Klassendefinitionen (*Schema*) mit dem enthaltenen Programmcode, der für die Behandlung der gespeicherten Objekte notwendig ist, im Store neben den eigentlichen Objektdaten abgespeichert werden. Dies schließt alle Super-Klassen und alle Klassen referenzierter Objekte und deren Super-Klassen rekursiv ein.

Dieses Verfahren stellt die Konsistenz zwischen Programmcode und Daten sicher und hat daneben weitere Vorteile, aber auch Kosten: Die Vorteile beinhalten das Vermeiden mehrfacher Initialisierungen statischer Variablen, Tests usw., die in vorhergehenden Ausführungen des Codes schon vorgenommen wurden. Als Nachteile mögen gelten, daß der Store mit wachsender Zahl benutzter Klassen natürlich ebenfalls wächst, und daß ein spezieller Class Loader benötigt wird, um eventuelle Änderungen an Klassendefinitionen in den Store einzuflechten (*Schema-Evolution*). Hierzu sei angemerkt, daß Schema-Evolution noch nicht in der gegenwärtigen PJama-Version enthalten ist.

Objekte in PJama werden persistent gemacht, indem sie im `PJavaStore`-Objekt als neue persistente Wurzelobjekte eingehängt werden. Danach ist dann das Objekt selbst und alle von ihm aus erreichbaren Objekte persistent. Wie bereits erwähnt, setzt PJama also transitive Persistenz um.

Zur Veranschaulichung, wie man in PJama ein Objekt als persistent markiert, soll folgendes Codebeispiel dienen. Zuvor soll ein leerer Store mit dem zu PJama gehörenden Shell-Kommando `opjcs` (siehe Kapitel 6.4.1) erzeugt worden sein.

```
PJStore _store = PJStoreImpl.getStore();
Archive _myarc = new Archive();

_store.newPRoot("MeinArchiv", _myarc);
```

Mit der Methode `newPRoot` am Store-Objekt markiert der Applikationsprogrammierer das Behälter-Objekt `_myarc` als persistent. Das Objekt ist nach erfolgreicher Beendigung des Programms für alle späteren Programmaufrufe im Store verfügbar. Es muß dann unter dem Namen „MeinArchiv“ wieder abgerufen werden:

```
PJStore _store = PJStoreImpl.getStore();
Archive _myarc = (Archive) _store.getPRoot("MeinArchiv");
```

Nun mag es umständlich erscheinen, alle persistenten Objekte explizit markieren und importieren zu müssen. Dazu muß man sich jedoch die transitive Persistenz vor Augen führen: Markiert man ein Behälter-Objekt als persistent, so sind alle darin enthaltenen Objekte automatisch schon persistent und der Abruf des Behälters holt auch die enthaltenen Objekte wieder aus dem Store. In der Realität muß man tatsächlich nur sehr wenige Wurzel-Objekte explizit markieren.

Da diejenigen Klassen, die mit persistenten Daten umgehen, ebenfalls persistent gemacht werden, ist in einem leeren Store zumindest immer das `PJavaStore`-Objekt und alle erreichbaren Klassen „eingefroren“. Ein Store enthält und beschreibt sich also selbst („self-contained and self-describing“ [Atkinson et al. 96a]). Für obiges Beispiel wird also auch der Code der Klasse `Archive` und alle dafür benötigten Klassen gespeichert.

Endet ein persistente Objekte haltendes Programm ohne Fehler, so werden alle als persistent markierten oder erreichbaren Objekte in den Store gespeichert. Diesen Vorgang nennt man *Promotion*. Da dies allein selbst als Standard-Transaktionsverhalten noch zu dürftig ist, hat der Programmierer die Möglichkeit, globale Stabilisierungen zu jedem (natürlich möglichst datenkonsistenten) Zeitpunkt durchzuführen. Als globale Stabilisierung wird hierbei verstanden, daß *alle* zwischenzeitlich veränderten Objekte dauerhaft gemacht werden. Die Möglichkeit, nur Gruppen von Objekten zu stabilisieren, gibt es noch nicht.

Ein besonderes Problem für die Persistenz stellen statische Variablen dar. Diese werden nur einmal initialisiert wenn die Klasse geladen wird. Im normalen Java ist dies einmal pro Programmausführung. In einer persistenten Umgebung muß umgedacht werden. Hier gibt es nun offensichtlich das Bedürfnis nach zwei verschiedenen Semantiken für statische Variablen:

- persistente statische Variablen, die nur einmal zu Beginn ihrer Lebensdauer initialisiert werden, z.B. Aufrufzähler

- transiente statische Variablen, die mit jeder Programmausführung neu initialisiert werden, also jedesmal, wenn sie aus dem Store geholt werden. Ein Beispiel hierfür wären Variablen, die nach der Systemumgebung gesetzt werden (welche sich von Aufruf zu Aufruf ändern kann).

Unglücklicherweise wurde das in Java vorgesehene Schlüsselwort `transient` von Sun, genauer JavaSoft, mißbräuchlich für eine Kennzeichnung im Kontext des Serialisierungs-Konzepts verwendet. Bis dieser Mißstand ausgeräumt ist, haben sich die Entwickler von PJama eine andere Kennzeichnung transienter Variablen einfallen lassen müssen. Wie bei der Registrierung persistenter Wurzelobjekte müssen so auch transiente Variablen einer „Registrierungsstelle“ kenntlich gemacht werden. Daß dies nicht gerade eine zufriedenstellende Verfahrensweise ist, dürfte klar sein, ist aber nicht eine Schwäche des PJama-Entwurfs.

5.2.3 Kriterien orthogonaler Persistenz in Anwendung auf PJama

PJama ist als orthogonal persistente Programmiersprache konzipiert worden und erfüllt die drei gezeigten Kriterien orthogonaler Persistenz zwar auch fast vollständig, aber eben nicht ganz, wie jetzt kurz skizziert wird:

- Unabhängigkeit von Persistenz: PJama akzeptiert unmodifizierten Java-Code und Programmoperationen sehen auf langlebigen Daten wie auf kurzlebigen Daten gleich aus. *Aber*: Das explizite Registrieren von persistenten Wurzel-Objekten ist ein pragmatischer Kompromiß, der allen Problemen aus dem Weg geht, die entstehen würden, wenn alle geladenen Klassen implizit persistente Wurzel-Objekte wären. Viele dieser Objekte enthalten transiente Daten, die nicht als solche gekennzeichnet sind. In PJama hat also die Applikation die volle Kontrolle darüber, welche Klassen persistent gemacht werden [Jordan 96].

- Typ-Orthogonalität: In PJama können neben den Java-Datentypen auch Klassen persistent gemacht werden. Trotzdem gibt es als Ausnahme solche Datentypen, die nicht langlebig sein können; Kapitel 5.2.5 zeigt dies für Objekte des Typs `Thread`, AWT-Objekte und einige I/O-Klassen.

- Identifikation von Persistenz: In PJama gibt es einen Mechanismus, wie persistente Daten erkannt werden können; die transitive Persistenz oder Persistenz durch Erreichbarkeit regelt, daß jedes Objekt, das von einem persistenten Wurzel-Objekt aus erreichbar ist, ebenfalls persistent wird. Dieses Kriterium orthogonaler Persistenz ist also vollständig erfüllt.

Streng genommen ist PJama also noch keine vollständig orthogonal persistente Lösung; die genannten Einschränkungen bedingen allerdings keinen impedance mismatch.

5.2.4 Architektur des PJama-Systems

Wenn in Kapitel 5.1.2 gesagt wurde, daß einer orthogonal persistenten Programmiersprache eine Datenbank unterliegt, so ist dies hier nicht im herkömmlichen Sinn einer query-fähigen externen Datenbank-Engine zu verstehen. Vielmehr handelt es sich bei der PJama-„Datenbank“ um ein Konglomerat aus Cache-Strukturen mit Koppelung an einen dauerhaften virtuellen Speicher – im Falle von PJama ist dies RVM[8] [Mashburn et al. 94].

Ein Blick auf die Architektur von PJama lohnt sich in jedem Fall:

---

[8] Recoverable Virtual Memory

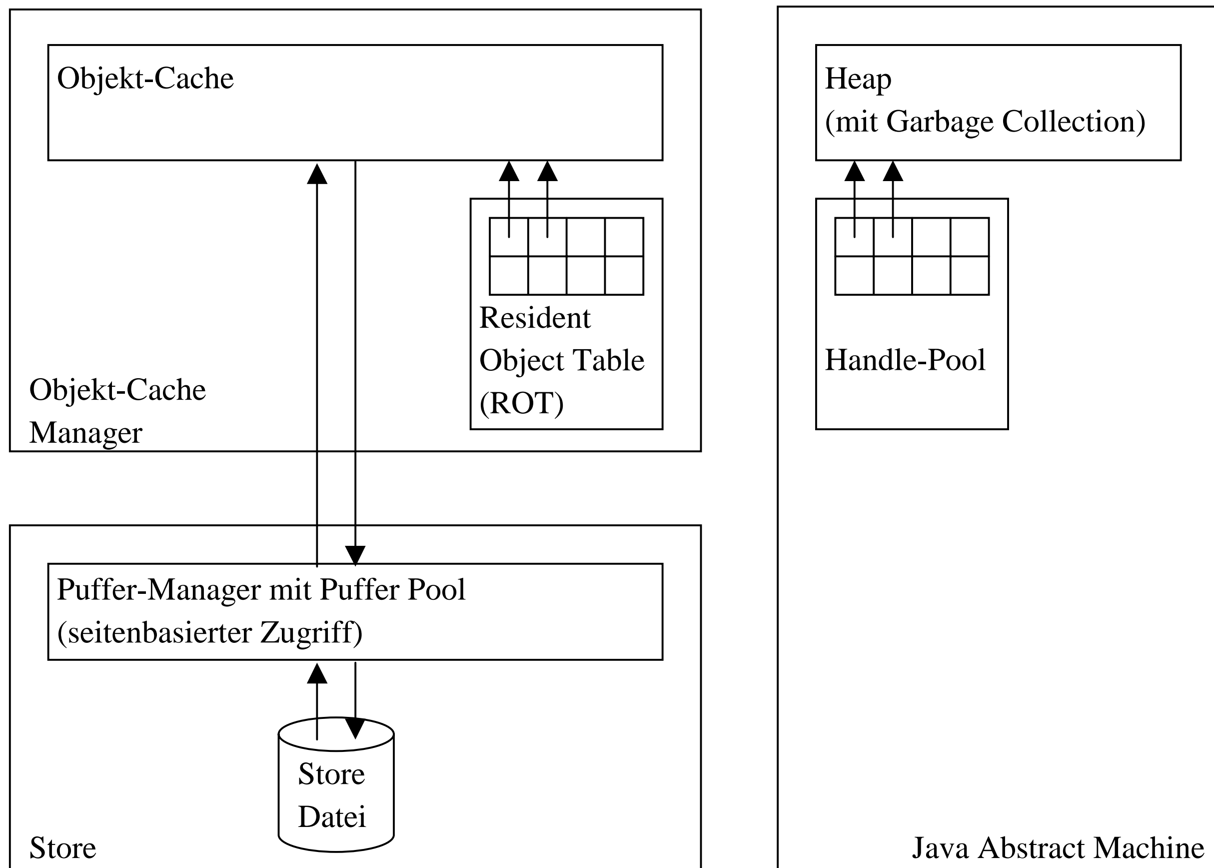

Abbildung 7: Die Objekt-Cache-Architektur von PJama (vereinfachte Darstellung nach [Atkinson et al. 96a])

Die Änderungen an der Java Abstract Machine und am Interpreter sind tatsächlich minimal geblieben; insbesondere der Heap mit Garbage Collection und die internen Handles, die zum Zugriff auf transiente Objekte benutzt werden, bleiben unverändert im Format von Java. Es ist in Java nämlich so, daß es zwar für den Programmierer so erscheint, als verwiesen Objekte wie etwa in C++ direkt aufeinander, tatsächlich aber eine Indirektion dazwischen liegt – die sogenannten Handles. Dieser Mechanismus wird nun für die Persistenz genutzt.

Alle Persistenz-Aktivitäten werden vom Objekt-Cache Manager und vom Puffer-Manager gehandhabt. Objekte im Objekt-Cache werden von der JAM über Handles, die von denen transienter Objekte auf dem Heap unterscheidbar sind, adressiert. Die Handles für persistente Objekte werden in der *Resident Object Table* (*ROT*) verwaltet.

Der Objekt-Cache wird durch das Laden von `PJavaStore`-Klasse und –Exemplar sowie aller abhängigen Objekte und Klassen initialisiert. Wird danach ein persistentes Objekt dereferenziert, das nicht in der ROT steht, so lädt es der Objekt-Cache Manager über den Puffer-Manager in den Objekt-Cache.

Während der Stabilisierung des Stores erhalten neu als persistent erkannte Objekte auf dem Heap eigene Handles in der ROT, werden in den Objekt-Cache promoviert und dann in den Store geschrieben (vereinfachte Beschreibung nach [Jordan 96] und [Atkinson et al. 96]).

5.2.5 Einschränkungen des PJama-Prototyps

Wie bereits erwähnt, existiert PJama noch nicht in einer Release-Version. Daher müssen einige Einschränkungen bei der Benutzung von Persistenz mit PJama in Kauf genommen werden.

Zunächst einmal handelt es sich bei PJama ja um einen modifizierten Java-Interpreter. Somit ist das „Write Once, Run Anywhere“-Prinzip von Java verletzt, denn für den Einsatz persistenter Programm muß der spezielle PJama-Interpreter verwendet werden. Dieses Problem wäre durch eine Integration von PJama in Java zu lösen.

Desweiteren kann nur ein einzelner Store und damit ein Exemplar der `PJavaStore`-Klasse pro ausführender Virtual Machine verwendet werden. Mehrere Stores für verschiedene Teile z.B. eines großen Frameworks sind nicht möglich; hierbei handelt es sich um ein architekturbedingtes Problem, das durch die enge Verknüpfung des Java-Heaps mit Objekt-Cache und Store entsteht. Zudem ist die Größe des Stores auf 2 GByte beschränkt – ein Limit, das aus den maximalen Dateigrößen der meisten Betriebssysteme resultiert. Hier ist Abhilfe für die Zukunft geplant; eine automatische Aufteilung auf mehrere Store-Dateien soll möglich werden.

Die weiteren Einschränkungen sind unangenehmer: So kann PJama keine Threads persistent machen. Hierzu müssen laut den Autoren von PJama neue grundsätzliche Überlegungen angestellt werden, so daß eine Implementation für die nahe Zukunft unwahrscheinlich erscheint. Was die Unmöglichkeit persistenter Threads für problematische Auswirkungen hat, wird bei der Integration von PJama in das softwaretechnische Archiv gezeigt.

Außerdem fehlt PJama die Fähigkeit, Objekte des *Abstract Window Toolkit*s (*AWT*) persistent zu machen. Das AWT ist stark plattformabhängig und enthält nicht unerhebliche Mengen nativen Codes. In nativem Code kann PJama jedoch keine Datenkonsistenz sicherstellen, da hier ungeprüft auf Speicher zugegriffen werden kann. Ein weiteres Problem sind dabei Referenzen auf externen Zustand („*external state*“, [Jordan 96]), die z.B. in Form von Fensterhandles vom Betriebssystem verwendet werden können, in späteren Programmausführungen aber nutzlos sind.

Aus ähnlichen Gründen können einige I/O-Klassen ebenfalls nicht persistent gemacht werden. Zumindest die Beschränkung bezüglich des AWT wird aber mit dem Einsatz der in Java 1.2 enthaltenen systemunabhängigen GUI-Bibliothek *SWING* gegenstandslos werden.

Die genannten Einschränkungen stellen zwar einige Ärgernisse, aber kein grundsätzliches Hindernis für die geplante Integration in ein persistentes Archiv dar. Die Überlegungen aus den Kapiteln 3 und 4 sollen nun zusammen mit den Betrachtungen über Persistenz und PJama dieses Kapitels in die Implementation eines persistenten softwaretechnischen Archivs einfließen. Die praktische Anwendung von PJama wird dabei in Kapitel 6.4 erläutert.

6 Implementation und Dokumentation eines persistenten Archivs für behälterartige Materialien

Zu dieser Arbeit gehört die Implementation eines softwaretechnischen Archivs, im folgenden kurz Archiv genannt. Einerseits soll damit eine Beispiellösung mit allen Konstruktionsaspekten vorgestellt werden, anderseits wird aufgezeigt, inwieweit die in den vorherigen Kapiteln aufgestellten Entwurfsziele tatsächlich umgesetzt werden können.

6.1 Struktur des hierArchive

Im folgenden wird der prinzipielle Aufbau der verwirklichten Implementation eines persistenten Archivs erläutert. Aufgrund der besonderen Eignung für behälterartige oder hierarchische Materialien hat die Lösung den Namen *hierArchive*.

6.1.1 Die Schnittstelle für archivierbare Materialien

Jedes Material, welches im Archiv archiviert werden soll, muß eine Schnittstelle erfüllen, unter der es der Archivar ansprechen und verwalten kann. Weil aber die Materialien auch noch über die Archivierbarkeit hinaus unterschiedlich behandelt werden sollen, wird dies auch schon in der Schnittstelle widergespiegelt. Dies hat den großen Vorteil, daß das Archiv und der Archivar über die Typprüfung des Materials eine typsichere Behandlung der archivierten Objekte durchführen kann. Eine Lösung ohne Typunterscheidung könnte zu Fehlern durch falsche Anwendung der Schnittstelle führen, und die Schnittstelle müßte zumindest um eine Methode zur Unterscheidung zwischen behälterartigen und einfachen Materialien erweitert werden.

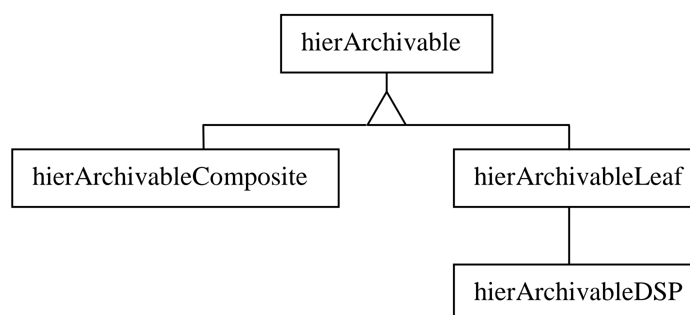

Abbildung 8: Vererbungsbaum der Schnittstelle für archivierbare Materialien

Die grundlegende Schnittstelle für alle archivierbaren Materialien namens `hierArchivable` besteht aus den vier grundlegenden Methoden, die alle archivierbaren Materialien erfüllen müssen.

```
public abstract interface hierArchivable extends Serializable
{
        public              ID getID();
        public void         setID(ID id);

        public String       getTitle();
        public void         setTitle(String title);
}
```

Jedes archivierbare Material muß eine ID, wie sie im Archiv-Paket definiert wird, und einen Titel, der einfach als String gehandhabt werden kann, verwalten. Über die ID wird die Wiedererkennung der Objekte (siehe Kapitel 4.5) auch über das Netz hinweg gewährleistet. Näheres zur Handhabung von IDs findet man in den folgenden Kapiteln.

Die Methode `setID` ist dabei nur für die Klassen des Archivs von Interesse, da die ID automatisch durch das Archiv erzeugt wird (siehe Kapitel 6.1.7). Wenn es in der verwendeten Sprache eine Möglichkeit dazu gibt, sollte sie dem Anwender des Archivs verborgen bleiben.

Über die Titel kann das Archiv dem Benutzer eine erste Idee vom Inhalt des Archivs bzw. eines Behälters im Archiv geben, ohne diesen schon herausgeben zu müssen. Titel und IDs sind wichtige Bestandteile des Inhaltsverzeichnisses des Archivs.

Um ein behälterartiges Material im Archiv speichern zu können, muß dieses allerdings noch weitere Methoden implementieren, damit das Archiv mit ihm arbeiten kann. Diese sind in der von `hierArchivable` erbenden Schnittstelle `hierArchivableComposite` definiert.

```
public interface hierArchivableComposite extends hierArchivable
{
        public void               addEntry(hierArchivable entry);
        public void               removeEntry(ID id);

        public void               startIteration();
        public hierArchivable     getNextEntry();
        public boolean            hasMoreEntries();

        public hierArchivableDSP  getDSP();
        public void               setDSP(hierArchivableDSP dsp);
}
```

An jedem Behälter müssen Methoden zu Einfügen neuer Einträge und zum Löschen von vorhandenen Einträgen zur Verfügung stehen.

Weiterhin setzt jedes behälterartige Material eine Schnittstelle zur Iteration durch seinen Behälter-Anteil um. Auch wenn zur Realisierung der Behälter im Archiv die zum JWAM-Framework gehörende jConLib (siehe [Bohlmann 98]) benutzt wird, wird an dieser Schnittstelle eine eigene Lösung verwendet, da die Integration des Archivs in JWAM nicht das Hauptaugenmerk bei dieser Implementation war, und deshalb eine für den Benutzer unabhängige Lösung konstruiert werden sollte. Daher steht auch nicht das Prinzip des stabilen Cursors zur Verfügung, und man kann nur einmal zur Zeit durch den Behälter iterieren.

Mit den Methoden getDSP und setDSP muß jeder Behälter seinen fachlichen Anteil (engl. *D*omain *S*pecific *P*art DSP) als ein eigenes Objekt ausgeben und wieder entgegennehmen können (siehe Kapitel 4.3). Das hierArchive unterstützt auf diese Art und Weise das Ausleihen und Verändern des fachlichen Anteils von Behältern, ohne daß diese dazu komplett aus dem Archiv genommen werden müssen.

Um die Objekte, die den fachlichem Anteil eines Behälters realisieren, von anderen archivierbaren Objekten unterscheiden zu können, gibt es eine eigene Schnittstelle namens hierArchivableDSP für fachliche Anteile.

```
public interface hierArchivableDSP extends hierArchivableLeaf
{}
```

Objekte mit dieser Schnittstelle müssen keine weiteren Methoden erfüllen, da die eigene Schnittstelle nur zur Typunterscheidung dient. Die Schnittstelle erbt von der Schnittstelle für nicht behälterartige und damit einfache archivierbare Materialien. Dies ist deshalb so gewählt, weil fachliche Anteile genaugenommen einen Spezialfall eines einfachen Materials darstellen.

```
public interface hierArchivableLeaf extends hierArchivable
{}
```

Archivierbare Materialien, die keine Behälter sind, haben ebenso wie fachliche Anteile eine leere Schnittstelle, die von hierArchivable erbt, also nur die grundsätzlichen Methoden für ein zu archivierendes Material enthält. Die Schnittstelle heißt hierArchivableLeaf und dient ebenfalls nur der Typunterscheidung gegenüber fachlichen Anteilen und Behältern. Technisch würde es tatsächlich keinen Unterschied ausmachen, ob hierArchivableDSP von hierArchivable oder hierArchivableLeaf erbt.

Der Vererbungsbaum der Schnittstellen entspricht in der Konstruktion dem Kompositum-Muster der Softwaretechnik [Gamma et al. 96] und ist dementsprechend benannt. Die Schnittstelle für fachliche Anteil ist dabei nur eine Erweiterung.

6.1.2 Das Material Archiv

Das eigentliche Objekt, in dem die Materialien gespeichert werden, ist in der gezeigten Konstruktion eines softwaretechnischen Archivs nur ein einfacher Behälter, der ebenfalls die Schnittstelle für behälterartige zu archivierende Materialien implementiert. In dieser Konstruktion wird der Behälter hierArchive genannt, darf aber in folgendem nicht mit dem Archiv als Begriff für die gesamte Konstruktion verwechselt werden. Das die Schnittstelle hierArchivableComposite implementierende hierArchive reicht dem Archivar als Material vollkommen aus und da das Archiv selbst archivierbar ist, sind spätere Konstellationen mit verteilten Archiven und zentralem Archivar leicht vorstellbar.

```
public class hierArchive implements hierArchivableComposite
{
        private conBag            _content;
        private Cursor            _iterator;
```

```
            private ID                  _id;
            private String              _title;
            ...
      }
```

Das hierArchive ist ein Beispiel für eine minimale Implementation der Schnittstelle für behälterartige Materialien.

Der Behälter selbst wird intern durch eine Abbildung auf einen Behälter der jConLib implementiert und alle Behältermethoden ebenfalls auf diesen übertragen. ID und Titel werden als einfache Attribute gesetzt und können über die Schnittstelle gelesen und geschrieben werden. Allein der fachliche Anteil bildet im hierArchive eine Ausnahme; da dieses in der gewählten Realisierung keinen fachlichen Anteil braucht, bleiben die Methoden zum Schreiben und Lesen desselben leer.

An dieser Stelle bleibt zu überlegen, ob das hierArchive wirklich als bloßes hierArchivableComposite angesehen werden kann, oder ob die Aufteilung der Aufgaben zwischen Archivar und Archiv anders sinnvoller gelöst werden könnte.

Der Archivar, genannt hierArchivist (engl. Archivist = Archivar), implementiert die gesamte Logik des Ausleihens, Zurückstellen und aller verwandten Aktionen des Archivs unter Benutzung des hierArchive und anderer, teilweise schon angesprochener Arbeitsgegenstände. Für den Benutzer außerhalb bietet der Archivar den einzigen Zugang zur Funktionalität des softwaretechnischen Archivs.

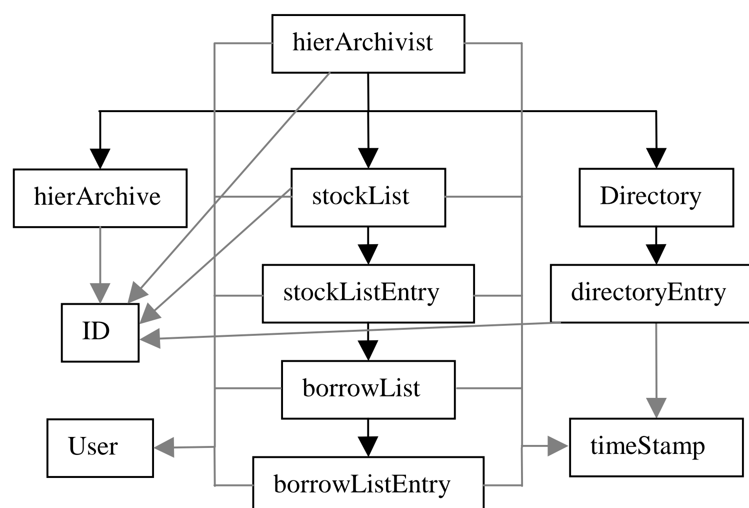

Abbildung 9: Benutzt-Beziehung der Arbeitsgegenstände im Archiv

Im hierArchive werden die Materialien selbst gespeichert, während in der Bestandsliste, der stockList, alle notwendigen Daten zu Ausleihern und Ausleihzeiten zu einem gespeichertem Material gehalten werden. Inhaltsverzeichnisse, genannt Directory, werden dynamisch vom Archivar erzeugt und dem Benutzer zur Einsicht in archivierte Materialien ausgegeben.

Die Objekte `ID`, Benutzer (`User`), und Zeitstempel (`timeStamp`), werden von den verschiedensten Klassen verwendet und in den nachfolgenden Kapiteln näher erläutert.

Da der Archivar für den Benutzer die gesamte Archiv-Funktionalität nach außen kapselt, soll kurz ein Blick auf die Schnittstelle des `hierArchivist` geworfen werden.

```
public class hierArchivist implements Serializable
{
        public User register(String username) { ... }
        public void unregister(User user) { ... }

        public void unregister_message_callback(fpObject cb_obj,
               String cb_method, Class message) { ... }
        public void unregister_message_callback(fpObject cb_obj,
               String cb_method, Class message) { ... }

        public ID getArchiveID(){ ... }
        public Directory getDirectory(ID id) { ... }

        public boolean addEntry(hierArchivable entry, ID containerid) { ... }
        public boolean giveBackEntry(hierArchivable entry, User user) { ... }
        public hierArchivable borrowEntry(ID id, User user) { ... }
        public void freeEntry(ID id, User user) { ... }
        public void removeEntry(ID id) { ... }

        public ID getDSPID(ID id) { ... }
        public borrowList getBorrowers(ID id) { ... }
}
```

Zur Bedeutung der einzelnen Umgangsformen geben die nächsten Kapitel genaueren Aufschluß. Zum Thema Einhaltung des Vertragsmodells nimmt Kapitel 6.1.8 Stellung.

6.1.3 Registrierung eines Benutzers (`register`, `unregister`)

Die integrierten Umgangsformen zur Registrierung eines Benutzers am Archiv bilden derzeit nur einen Ersatz für eine später im Framework verankerte Benutzerverwaltung (siehe Kapitel 4.6). Entsprechend einfach ist die Benutzerklasse (`User`) gehalten; sie funktioniert im wesentlichen auf Basis eines Strings zur Identifikation. Das einmal zu Anfang erzeugte Benutzerobjekt wird in den meisten nachfolgenden Aktionen des Benutzers mit dem Archiv benötigt; es hat wichtige Bedeutung für Ausleihvorgänge und die Zurückstell-Logik. Die abschließende Abmeldung des Benutzers am Archivar ist nicht zwingend vorgegeben, gehört aber zur korrekten Verwendung des Archivs, damit der Benutzer wieder aus allen Ausleihlisten gelöscht werden kann.

Sowohl Anmeldung als auch Abmeldung des Benutzers am Archiv bilden also einen Rahmen bei die Verwendung des Archivars. Dieser wird momentan noch nicht erzwungen, so daß bei falscher Anwendung Probleme auftreten können. Spätere Versionen könnten versuchen, dieses Rahmenwerk bei jeder Archivbenutzung zwingend zu machen, in dem sie beispielsweise die Anmeldung schon in den Konstruktor des Archivars aufnehmen. Die

Abmeldung könnte durch Timeout-Mechanismen oder mit der Garbage-Collection des Archivar-Proxies erzwungen werden.

Nach den Grundsätzen des Vertragsmodells würde man hier noch ein sondierende Funktion in der Art von

```
public boolean isRegistered(User user) { ... }
```

erwarten. Diese wurde jedoch bewußt weggelassen, damit das Archiv keine Informationen zu seinen benutzenden Anwendern speichern muß. Das Archiv würde sonst für jeden Benutzer Zustandsinformationen speichern müssen, die bei Verbindungsabbrüchen zwischen Benutzer und Archiv, zum Beispiel durch Netzprobleme, verworfen oder für einen Wiederaufbau der Verbindung nach Qualitätssicherungsprotokollen verwendet werden müßten. Das Thema Client/Server-Modelle für verteilte Dienste ist allerdings auch im Arbeitsbereich Softwaretechnik erst in der Diskussion; deshalb wurde das Archiv vorerst nach einem „stateless"-Modell ohne Wissen über seine Benutzer und daher ohne die Möglichkeit einer sondierenden Methode `isRegistered` implementiert.

6.1.4 Mit dem Archiv arbeiten (`getArchiveID`, `getDirectory`)

Der Zugriff des Benutzers auf im Archiv befindliche Materialien erfolgt ausschließlich über die Archiv-ID des Materials. Diese muß der Benutzer jedoch nicht selbst vor der Archivierung erzeugen; sie wird wie später beschrieben beim Einfügen in das Archiv vom Archivar vergeben.

Da der Benutzer nun auf der Client-Seite nichts über die vom Archivar vergebenen IDs weiß, muß er zum Einstieg die ID des obersten Knotens im Archiv kennen – diese wird vom `hierArchive`-Objekt getragen. Zum Abfragen der ID dieses Objekts steht dem Benutzer eine eigene Methode zur Verfügung (`getArchiveID`).

Der Archivar kann zu jedem archivierten Behälter ein Inhaltsverzeichnis aller enthaltenen Objekte erzeugen und an den Benutzer herausgeben. Mit der ID des Archivs selbst kann der Benutzer nun also eine Liste aller archivierten Objekte der ersten Stufe abfordern. Da im Inhaltsverzeichnis wiederum die IDs der enthaltenen Objekte verzeichnet sind, erhält der Benutzer sukzessive Zugriff auf den gesamten Inhalt des hierarchischen Archivs.

Die Methode `getDirectory(ID id)` generiert ein Inhaltsverzeichnis zu einem Behälter und gibt dieses an den Benutzer weiter.

```
public class Directory implements Serializable
{
        ...
        public int getNumberOfEntries() {...}

        public void startIteration() {...}
        public boolean hasMoreEntries() {...}
```

```
        public directoryEntry getNextEntry() {...}

        public void addEntry(directoryEntry entry) {...}
}
```

Der Benutzer kann über die am Inhaltsverzeichnis zur Verfügung stehenden Schnittstelle durch dieses iterieren, um die einzelnen Inhaltsverzeichniseinträge einzusehen. Die Iterationsschnittstelle ist von derselben Art wie die an der Schnittstelle `hierArchivable` für zu archivierende Materialien und es wird aus demselben Grund wie dort kein Objekt der jConLib ausgegeben.

Ein einzelner Inhaltsverzeichniseintrag ist wie folgt aufgebaut:

```
public class directoryEntry implements Serializable
{
        private String          _title;
        private ID              _identification;
        private boolean         _isContainer;
        private timeStamp       _changeTime;

        public directoryEntry(String title, ID identification,
              boolean isContainer, timeStamp changeTime) {...}
        ...
 }
```

Bei der Generierung des Inhaltsverzeichnisses wird zu jedem Material, welches im abgefragten Behälter enthalten ist, ein Inhaltsverzeichniseintrag mit der ID des Materials, seinem Titel und einem Flag, ob es selbst wiederum ein Behälter ist, angelegt. Diese Informationen kann der Archivar über die `hierArchivable`-Schnittstelle des Materials und über dessen genaue Typbestimmung feststellen. Weiterhin ist im Inhaltsverzeichniseintrag der Zeitstempel der letzten Änderung aus dem zugehörigen Bestandslisteneintrag enthalten.

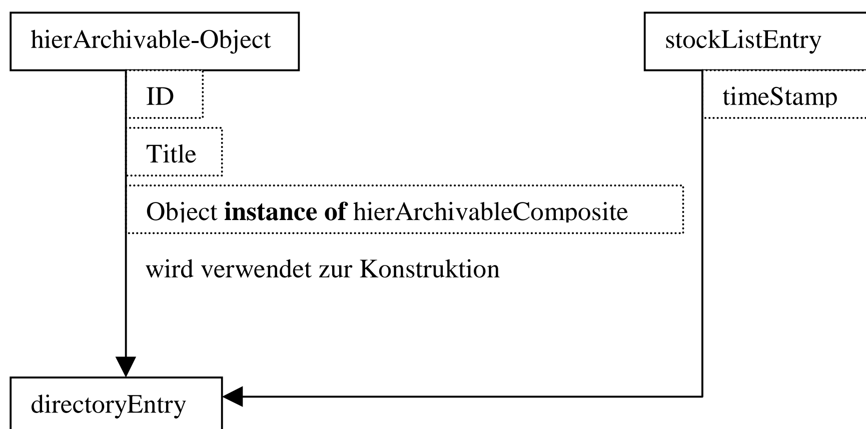

Abbildung 10: Generierung eines Inhaltsverzeichniseintrags

Bei der Benutzung der Inhaltsverzeichniseinträge sollen nun die Titel der Materialien zur Präsentation innerhalb der Anwendung dienen – sie abstrahieren von der Archiv-ID und können fachlich motiviert sein. Demgegenüber dienen die IDs zur Arbeit mit dem Archiv und sollten dem Benutzer der Applikation verborgen bleiben. Die zusätzlich enthaltenen Daten

über ein Material sollen die Netzlast und den Arbeitsaufwand für den Programmierer verringern, indem sie weiteren wahrscheinlichen Abfragen an den Archivar vorweggreifen. Beispielsweise ist die Information, ob ein Material wiederum ein Behälter ist, relativ elementar und kann auch zur Präsentation des Materials in der Applikation erwünscht sein.

6.1.5 Die Bestandliste und die Ausleihliste

Neben dem Material Archiv führt der Archivar eine Bestandsliste, in der er alle zur Verwaltung der archivierten Materialien nötigen Daten speichert. Ihr Aufbau wird in Abbildung 11 verdeutlicht.

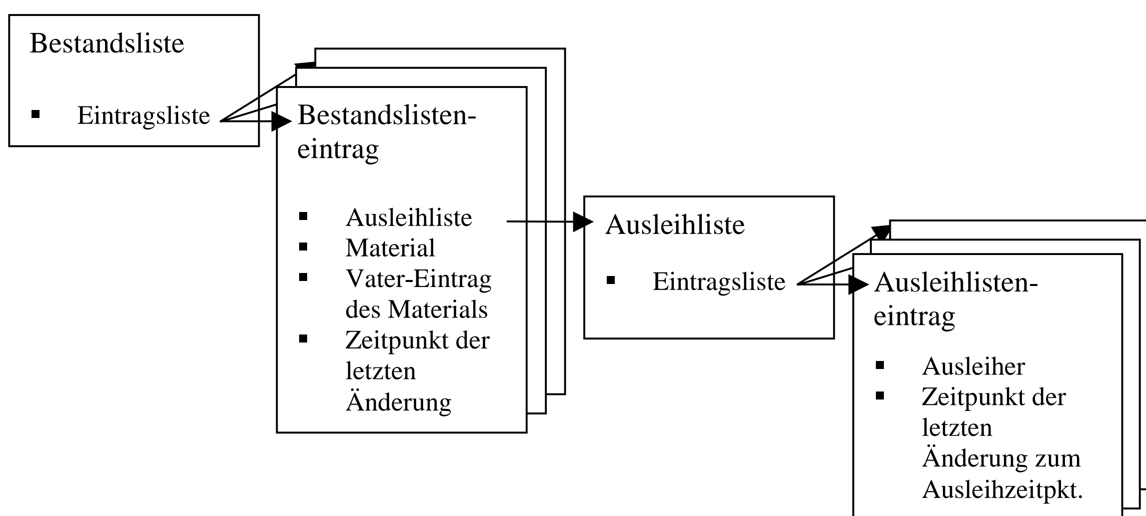

Abbildung 11: Aufbau von Bestands- und Ausleihliste

Für jedes im Archiv gespeicherte Material, also auch für die verschachtelt in einen Behälter enthaltene Materialien, ist in der Bestandsliste (`stockList`) des Archivs ein Bestandslisteneintrag (`stockListEntry`) vorhanden. Dieser Eintrag wird beim Einfügen des Materials in das Archiv angelegt und erst beim Löschen des Materials aus dem Archiv entfernt. Er speichert neben einer Referenz auf das Material auch eine Referenz auf den Behälter, in dem das Material liegt. Dadurch kann der Archivar über den Bestandlisteneintrag eines Materials jederzeit auch an den Behälter und jeden weiteren übergeordneten Behälter des Materials herankommen. Dies ist beispielsweise beim Zurückstellen von Materialien in das Archiv von Bedeutung.

Weiterhin wird im Eintrag ein Zeitstempel mit dem Zeitpunkt der letzten Änderung des Materials im Archiv gehalten, also dem Zeitpunkt, wann das Material zuerst eingefügt oder zuletzt zurückgestellt wurde. Auch dieser ist für das Zurückstellen von Bedeutung. Schließlich ist jedem Bestandslisteneintrag noch eine Liste von Ausleiheinträgen (`borrowList`) zu dem Material zugeordnet. Jeder Ausleiheintrag (`borrowListEntry`) speichert dabei einen Benutzer, der das Material ausgeliehen hat und den Zeitstempel der letzten Änderung des Materials zum Ausleihzeitpunkt, also quasi die Version des Materials.

Auch diese Daten dienen der Kontrolle beim Zurückstellen, aber auch als Information für das Archiv benutzende Werkzeuge, die so beispielsweise eine Liste derjenigen Benutzer abfragen können, die ein Material zur gleichen Zeit bearbeiten (siehe Kapitel 6.1.13).

Die Zeitstempel in dieser Konstruktion sind einfache Objekte, die ein Datum speichern und einen Wertvergleich über diesem durchführen können. In einer älteren Lösung des Archivs wurden die Zeitstempel ebenso wie die IDs jedem Material selbst zugeordnet. Dadurch mußte das Material eine größere Schnittstelle erfüllen und selbst die Zeitstempel verwalten. Allerdings war dadurch keine so verschachtelte Speicherung der Ausleihdaten wie in dieser Konstruktion notwendig, sondern es reichte die Verknüpfung von Ausleihern und ausgeliehenem Material zu vermerken.

Statt Zeitstempel wären auch Versionsnummern für die Materialien denkbar. Diese hätten den Vorteil, daß sich eine Weiterentwicklung des Archivs zu einem System mit Versionierung ergeben könnte, und daß auch bei sehr kleinen Zeitintervallen zwischen zwei Ausleihvorgängen keine gleichen Zeitstempel existieren könnten. Fachlich handelt es sich bei der Zeitstempel-Logik nämlich tatsächlich zweifelsfrei um eine Versions-Logik.

Man sollte die Zeitstempel allerdings nicht vergessen, denn sie lassen sich dann immer noch gut verwenden, wenn man das Archiv um eine Historisierungskomponente erweitern möchte.

6.1.6 Materialien ins Archiv einfügen (`addEntry`)

Das Archiv ist zu Beginn natürlich bis auf den Wurzelknoten `hierArchive` vollkommen leer und jedes Inhaltsverzeichnis dazu ebenfalls.

Um neue Materialien in das Archiv zu stellen, bietet der Archivar eine Methode namens `addEntry` an. Dieser Methode wird das Material selbst und die ID des Behälters, in die das Material eingefügt werden soll, übergeben. Zunächst einmal wird man hier die ID des `hierArchive` übergeben, damit das Material direkt unter dem Archiv-Objekt eingefügt wird.

Es ist aber auch möglich die ID eines anderen schon vorhandenen Behälters im Archiv anzugeben, damit das Material in diesem gespeichert wird. Ob diese Einfügung fachlich korrekt ist, wird dabei nicht überprüft und würde im Fehlerfall beim Einfügen in den konkreten Behälter zu einem Typkonflikt führen.

Die Generierung eindeutiger ID für jedes archivierte Material ist unbedingte Voraussetzung im Archiv. Eine ältere Lösung mit der Generierung von IDs durch die Behälter selbst hat sich als zu kompliziert erwiesen, da die IDs nicht über den Behälter hinaus eindeutig waren und ein System von zusammengesetzten IDs für eine globale Wiedererkennung benutzt werden mußte. Außerdem wurde die Schnittstelle der zu archivierenden Materialien dadurch erweitert und dessen Implementierung verkompliziert.

In der jetzigen Lösung werden die IDs vom Archivar einfach über einen statischen Zähler erzeugt. Die Verwaltung des Zählers in der IDs-Klasse brachte Probleme im Hinblick auf die

Persistenz durch Serialisierung (siehe Kapitel 6.3) mit sich. Eine Verfeinerung der ID-Technik ist mit Hinblick auf eine zukünftige Übernahme von IDs aus dem Framework JWAM jedoch nicht mehr sinnvoll.

Gleich mehrere Vorgänge erledigt der Archivar beim Einfügen von neuen Materialien. Zum ersten wird für jedes neue Material eine ID generiert und dem Material zugeordnet. Zum zweiten wird für jedes Material ein Eintrag in der Bestandsliste erzeugt und dort das Material, sein Behälter (für die obersten Materialien ist dies das `hierArchive`-Objekt) und der Zeitpunkt des Einfügens eingetragen. Für eingefügte Behälter wird der Vorgang der ID-Generierung und des Bestandslisteneintrages für jedes im Behälter und Unter-Behältern enthaltenen Materialien wiederholt. Dazu iteriert der Archivar über die an jedem Behälter vorhandene Schnittstelle durch die Materialien. Als letztes wird das Material tatsächlich in das gewünschte Wurzelobjekt eingehängt. Handelt es sich bei dem Material um einen fachlichen Anteil, wie durch Typüberprüfung festgestellt werden kann, wird er mit der Methode `setDSP` am gewünschten Behälter gesetzt.

6.1.7 Materialien aus dem Archiv ausleihen (`borrowEntry`)

Über die Methode `borrowEntry` am Archivar kann der Benutzer jedes archivierte Material ausleihen, von dem er die ID kennt.

Der Archivar greift über seine Bestandsliste auf das Material mit der ID zu und gibt eine Kopie heraus. Dabei vermerkt der Archivar am zum Material gehörenden Bestandslisteneintrag den Benutzer als Ausleiher und trägt ihn mit dem Zeitpunkt der letzten Änderung des Materials in die Ausleihliste, die zu dem Bestandslisteneintrag gehört, ein. Außerdem wird eine Nachricht an alle registrierten Benutzer verschickt, die über eine Veränderung der Ausleihliste des Materials informiert.

Das Archiv gibt das Material nur als Kopie heraus und das Original verbleibt immer im Archiv. So kann jeder Benutzer ohne Wartezeiten jederzeit auf das Material zugreifen und das Archiv muß sich nicht merken, ob und an wen das Original verliehen ist. In dieser Implementation wurde also die in Kapitel 4.1 beschriebene nur-Kopie-Logik verwirklicht.

6.1.8 Materialien ins Archiv zurückstellen (`giveBackEntry`)

Über die Methode `giveBackEntry` kann der Benutzer ausgeliehene Materialien in das Archiv zurückstellen.

Im Gegensatz zu den Forderungen des Vertragsmodells (siehe Kapitel 4.6) gibt es keine sondierende Funktionen zur Abfrage, ob das Material in das Archiv zurückgestellt werden kann, und der Archivar garantiert nicht, daß das Zurückstellen erfolgreich verläuft. Stattdessen signalisiert sie Erfolg oder Mißerfolg durch einen boolschen Rückgabewert. Die

Implementation eines Transaktionsmanagements, das auch sondierende Funktionen bei Mehrbenutzerzugriff gewährleisten könnte, lag nicht im Fokus dieser Arbeit.

Es können nur Materialien an das Archiv zurückgegeben werden, die bereits im Archiv gespeichert waren. Der Archivar überprüft dies anhand seiner Bestandsliste und verweigert gegebenenfalls die Annahme. Die Unterscheidung von erstmaligem Einfügen und Rückgabe von Materialien war ursprünglich fachlich motiviert, und auch die Implementation hat gezeigt, daß beide Vorgänge zwar Ähnlichkeiten aufweisen, letztlich aber zu verschieden sind, um sie in einer einzigen Methode aufzunehmen.

Der Hauptgrund für das Scheitern des Zurückstellens eines ausgeliehenen Materials liegt in Zugriffskollisionen mit anderen Benutzern (siehe Kapitel 4.1), die durch die nur-Kopie-Logik entstehen können.

Im Archiv werden die Konflikte beim Zurückschreiben von Kopien nach einem einfachen Zeitvergleichsverfahren gelöst. Nur eine Kopie, die nicht älter als das im Archiv befindliche Original ist, darf zurückgeschrieben werden. Der Archivar vergleicht dazu den Zeitstempel des Bestandslisteneintrages des Materials mit dem Zeitstempel, der in der Ausleihliste für dieses Material und den zurückstellenden Benutzer gespeichert wurde und der die Version angibt, welche der Benutzer hält.

Ist der erste Zeitstempel neuer als der zweite, so ist die ausgeliehene Kopie veraltet - in der Zwischenzeit wurde demnach eine andere Kopie als neues Original zurückgestellt. In diesem Fall wird das Zurückstellen der Kopie verweigert und das Archiv bleibt unverändert.

Wenn die Kopie allerdings zurückgestellt werden darf, so werden vom Archivar drei Dinge erledigt.

- Zum ersten wird das Material tatsächlich aktualisiert, also das alte Original durch die entgegengenommene Kopie ersetzt. Fachliche Anteile werden dabei wie beim Einfügen erkannt und gesetzt. Außerdem wird eine Nachricht an alle registrierten Benutzer über eine Veränderung des Materials abgeschickt.

  Um fachliche Konsistenzen bei der Arbeit mit Materialien im Archiv zu wahren, wird das Ersetzen des Originals durch den übergeordneten Behälter selbst vorgenommen. Über die an der Schnittstelle von Behältern vorhandene Methoden `removeEntry` und `addEntry` wird das Material zuerst aus dem Behälter entfernt und dann neu eingefügt. Der Behälter kann dabei fachliche Überprüfungen des ersetzten Materials selbst durchführen.

- Zum zweiten muß die Bestandsliste aktualisiert werden. Das neue Original wird in den Bestandslisteneintrag des Materials eingetragen und der Zeitstempel der letzten Veränderung des Materials wird durch einen aktuellen Zeitstempel ersetzt. Da wie in Kapitel 4.4 beschrieben eine Änderung eines Materials auch seinen Behälter verändert, wird der neue Zeitstempel auch für alle übergeordneten Behälter gesetzt. Ebenso wurde in Kapitel 4.4 festgehalten, daß mit der Änderung des Materials auch alle seine enthaltenen

Objekte aktualisiert werden müssen, so daß auch diese den neuen Zeitstempel bekommen. So kann gewährleistet werden, daß der Vergleich des Zeitstempels nur des zurückzustellenden Materials ausreicht, um sicherzustellen, daß keine indirekt beteiligten Materialien neuer als das Material selbst sind und damit Material-Inkonsistenzen auftreten.

- Als drittes wird der Eintrag des Benutzers aus der Ausleihliste zum Material gelöscht, da dieser nun das Material zurückgegeben hat und damit fachlich nicht mehr in der Hand hält. Deshalb wird eine Nachricht an alle registrierten Benutzer über eine Veränderung der Ausleihliste eines Materials verschickt.

In der verwendeten Konstruktion ist nicht gewährleistet, daß der Benutzer seine Kopie des Materials nun auch wirklich vernichtet. In jedem Fall kann er sie nicht ein weiteres Mal in das Archiv zurückstellen, da sich der Zeitstempel des Originals mit dem ersten Zurückstellen aktualisiert hat, und seine Kopie damit veraltet ist.

6.1.9 Materialien im Archiv freigeben (`freeEntry`)

Über die Methode `freeEntry` am Archivar kann der Benutzer jedes ausgeliehene Material über seine ID im Archiv wieder „freigeben", also als nicht mehr von ihm ausgeliehen kennzeichnen.

Diese Methode ist wichtig, wenn der Benutzer das ausgeliehene Material nicht zurückstellen möchte, etwa, weil er seine Änderungen verwerfen will oder das Material sicher nur zum Lesen behalten möchte. Der Benutzer verschwindet nach Aufruf der Methode aus der Ausleihliste zu dem Material; dadurch ist gewährleistet, daß andere Benutzer nicht durch scheinbaren „Konkurrenten" bei der Bearbeitung des Materials verwirrt werden.

Nach dem Aufruf von `freeEntry` ist das Zurückgeben des Materials für diesen Benutzer auf keinen Fall mehr möglich.

6.1.10 Materialien im Archiv löschen (`removeEntry`)

Mit der Methode `removeEntry` kann der Benutzer über den Archivar ein Material, dessen ID er kennt, im Archiv löschen.

Ähnlich dem Vorgang des Zurückstellens kann der Archivar nicht für die Erfüllung der Methode garantieren, sondern gibt dem Benutzer einen boolschen Wert zurück, der aussagt, ob die Methode erfolgreich war.

In der jetzigen Konstruktion des Archivs spielt es keine Rolle, ob das zu löschende Material ausgeliehen ist oder nicht - es kann jederzeit von Archivar gelöscht werden. Die Entscheidung für diese Konstruktion hat den Vorteil, daß der Benutzer in keinem Moment auf ausgeliehene Materialien warten muß, wenn er sie aus dem Archiv löschen möchte. Da alle ausgeliehenen

Materialien nur Kopien sind, ist es auch anschaulich sinnvoll, daß man das vorhandene Original jederzeit löschen darf. Außerdem folgt die Logik des Löschens so der des Zurückstellens, die besagt, daß immer der erste Zugriff zweier konkurrierender Benutzer erfolgreich ist.

Der Nachteil ist natürlich, daß ein neuer Konfliktfall auftritt, wenn der Benutzer seine ausgeliehene Kopie nicht mehr in das Archiv zurückstellen kann, weil das zugehörige Original fehlt. Deshalb wird beim Löschvorgang eine Nachricht an alle Benutzer geschickt, die das gelöschte Material ausgeliehen haben, um sie über die Löschung zu informieren.

Beim Löschvorgang selbst wird das Material aus seinem Behälter und aus der Bestandsliste mit allen zugehörigen Daten gelöscht und damit restlos aus dem Archiv entfernt. Ein zu löschender fachlicher Anteil wird mit den entsprechenden Methoden am Behälter mit einer leeren Referenz überschrieben und auf diese Art gelöscht. Ähnlich wie beim Zurückstellen eines Material werden die Zeitstempel aller Behälter, in denen das Objekt enthalten war, aktualisiert, so daß sie als erneuert gelten, und nun ebenfalls keine älteren ausgeliehenen Kopien von ihnen zurückgegeben werden dürfen.

Die Frage, ob und inwieweit ein Material, das in einem Archiv gespeichert wurde, überhaupt jemals wieder gelöscht werden können soll, muß noch diskutiert werden. Wenn das Archiv beispielsweise eine Historie führt, müssen zumindest Teile der Daten zum Archiv wie die Ausleihdaten und Zeitstempel auch über die Löschung hinaus erhalten bleiben. Das Archiv dieser Arbeit hat aber mehr die Koordination von Materialien zwischen Benutzern als Hintergrund, so daß die gewählte Lösung angemessen erscheint.

6.1.11 Fachliche Anteile bearbeiten (`getDPSID`)

Um auch die fachlichen Anteile von im Archiv befindlichen Behältern bearbeiten zu können, ohne diese mit allen daran hängenden Materialien ausleihen zu müssen, wurde in dieser Konstruktion des Archivs die Schnittstelle für Behälter so vorgegeben, daß jeder zu archivierende Behälter seinen fachlichen Anteil als ein eigenständiges Objekt herausgeben und wieder entgegennehmen kann. Dabei ist es die Entscheidung des Behälters, ob er seinen fachlichen Anteil auch intern als eigenes Objekt mit einer Referenz verwaltet. Denkbar wäre auch, daß das herausgegebene Objekt dynamisch erzeugt wird.

Zum Ausleihen eines Materials benötigt man wie bereits gesehen dessen ID. Da in der vorliegenden Konstruktion der fachliche Anteil eines Behälters so behandelt wird wie im Behälter enthaltene Materialien (siehe Abbildung 12), benötigt man zum Ausleihen eines fachlichen Anteils ebenfalls dessen ID. Diese bekommt man über die Methode `getDSPID` am Archivar, der man die ID des Behälters übergibt, zu dem man den fachlichen Anteil erfahren möchte.

Im Anschluß kann man den fachlichen Anteil mit den gleichen Methoden wie alle anderen Materialien ausleihen, zurückstellen, löschen oder neu einfügen. In den einzelnen Methoden

wird dies über eine Typüberprüfung erreicht, die fachliche Anteile anders behandelt als sonstige Materialien. Diese spezielle Handhabung bezieht sich jedoch nur auf die Behandlung der Objekte im Archiv. In der Bestandsliste und außerhalb des Archivs werden die fachlichen Anteile genau wie alle anderen Materialien behandelt. Das Vaterobjekt zu einem fachlichen Anteil ist sein Behälter genau wie für die anderen Materialien, die in diesem enthalten sind. Ein fachlicher Anteil hat ebenso eine ID, einen Zeitstempel und eine Ausleihliste, welche Benutzer ihn mit welchem aktuellen Zeitstempel ausgeliehen haben.

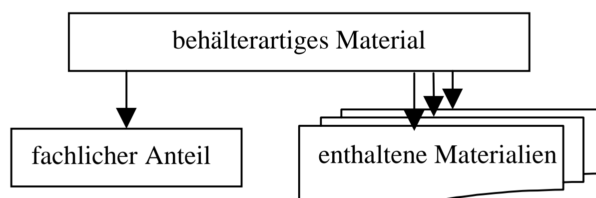

Abbildung 12: Modellsicht des fachlichen Anteils

Abstrakt stellt der fachliche Anteil eines Behälters eine spezielle Ausprägung eines im Behälter enthaltenen einfachen Materials dar. Diese Beziehung drückt sich auch in dessen Typisierung aus (siehe Kapitel 6.1) – der Typ des fachlichen Anteils wird vom Typ enthaltener einfacher Materialien abgeleitet (`hierArchivableDSP` erbt von `hierArchivableLeaf`).

6.1.12 Kooperation: Ausleihlisten abfordern (`getBorrowers`)

Mit der Methode `getBorrowers` stellt der Archivar den das Archiv benutzenden Werkzeugen eine Hilfe zur Koordination mit anderen Benutzern zur Verfügung. Die Methode liefert zur angegebenen ID die Ausleihliste mit allen Benutzern, die das Material ebenfalls ausgeliehen haben. Diese könnte dann im entsprechenden Werkzeug dauerhaft angezeigt werden. So kann der ausleihende Benutzer gegebenenfalls feststellen, daß er das Material alleine ausgeliehen hat und ohne zu erwartende Konflikte zurückstellen kann. Oder er sieht, daß das Material auch bei anderen Benutzern im Gebrauch ist und kann sich mit diesen außerhalb des Systems koordinieren.

Wenn die externe Koordination allerdings zu spät kommt, und ein anderer Benutzer das Material schon vor ihm zurückgestellt hat, hat er keine Möglichkeit mehr, sein Material in das Archiv zurückzustellen – außer, indem er es neu hinzufügt. Es bleibt zu überlegen, ob man in die Konstruktion des Archivs noch die Möglichkeit des unbedingten Überschreibens von neueren Materialien im Archiv nach Rückfrage gestatten möchte.

6.1.13 Kooperation: Nachrichten verschicken

Als Koordinationsmittel für die Benutzer des Archivs verwendet dieses den mehrprozeßraumfähigen Kommunikationsdienst, der bereits im Framework JWAM integriert

ist (siehe Kapitel 2.2). Der Archivar verschickt hierüber zu unterschiedlichen Anlässen Nachrichten (*Messages*) an die entsprechenden Benutzer.

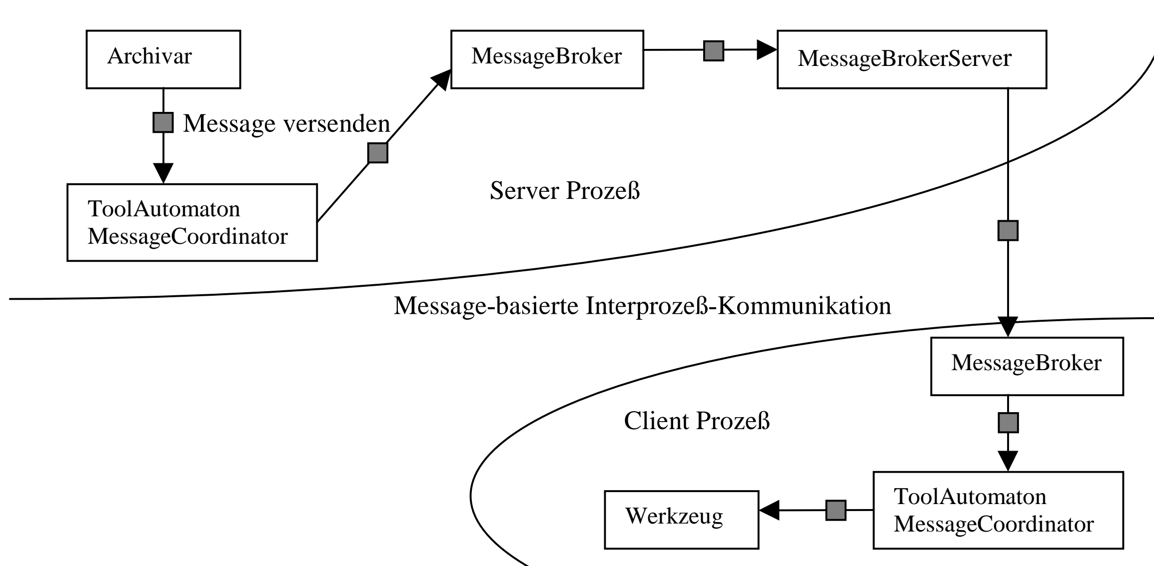

Abbildung 13: Messaging im Mehrprozeßraum mit JWAM

Normalerweise wird das Messaging im Framework zur Kommunikation zwischen Clients in unterschiedlichen Prozeßräumen verwendet und dabei ein dritter Prozeßraum für den Server benutzt. Da das Archiv aber selbst den Server für Client-Werkzeuge bildet, kann es parallel zum Messaging-Server in einem Server-Prozeßraum eingesetzt werden. Trotzdem bleibt die Architektur zum Versenden von Nachrichten auch im Server-Prozeßraum die gleiche wie im Client. Dies ist nicht zwingend, bringt aber Vorteile gegenüber zwei getrennten Server-Prozeßräumen (siehe Kapitel 6.2 und 6.3), weil beispielsweise dieselbe Registry bei der Verwendung vom RMI verwendet werden kann.

Damit das Messaging funktioniert, müssen sich alle Clients für die Nachrichten registrieren, die sie interessieren. Diese Registrierung muß explizit erfolgen und wird nicht im Rahmen der Anmeldung als Benutzer erledigt. Um wenigstens die Kenntnis über den Kommunikationsdienst des Frameworks gering zu halten, bildet der Archivar die Routinen zur Registrierung am Kommunikationsdienst nach, so daß die Aufrufe am Archivar selbst geschehen,

```
_archivist.register_message_callback(this, "materialUpdated",
       msgHierArchiveMaterialUpdated.class);
_archivist.register_message_callback(this, "materialRemoved",
       msgHierArchiveMaterialRemoved.class);
_archivist.register_message_callback(this, "borrowersUpdated",
       msgHierArchiveBorrowersUpdated.class);
```

während sie intern auf das Framework abgebildet werden.

```
public void register_message_callback(fpObject cb_obj, String cb_method,
        Class message)
{
```

```
            ToolAutomatonMessageCoordinator.instance().register(cb_obj,
                  cb_method, message);
    }
```

Sobald der Archivar eine Nachricht zu verschicken hat, wird über das Messaging-System die registrierte Methode aufgerufen. Für einen sauberen Abschluß muß der Client vor seiner Beendigung die Registrierung am Archivar wieder entfernen.

Die erste im Archiv benutze Nachricht ist `msgHierArchiveMaterialUpdated` und informiert die Clients darüber, wenn ein Material im Archiv aktualisiert wurde. Dies ist dann der Fall, wenn ein Benutzer ein Material zurückgestellt hat. Der Archivar erzeugt eine neues Message-Objekt und übergibt dieses dem Messaging-System:

```
    ToolAutomatonMessageCoordinator.instance().sendMessage(
            new msgHierArchiveMaterialUpdated(id, user));
```

Am Client wird dann direkt die registrierte Methode zu dieser Nachricht aufgerufen und die Nachricht selbst als Parameter übergeben. Neben der Information über die Änderung transportiert die Nachricht, welches Material geändert wurde, und welcher Benutzer das Material verändert hat.

Die zweite Nachricht hat die gleichen Parameter wie die erste und heißt `msgHierArchive-MaterialRemoved`. Sie wird verschickt, wenn ein Benutzer ein Material aus dem Archiv gelöscht hat.

Die dritte verwendete Nachricht heißt `msgHierArchiveBorrowersUpdated` und gibt den Clients die Information, daß sich die Ausleihliste eines Materials im Archiv geändert hat. Dies geschieht, wenn eine Benutzer ein Material entweder zurückgestellt und damit wieder freigegeben hat, oder wenn er sich ein Material neu ausgeliehen hat. Diese Nachricht wird ebenfalls vom Archivar erzeugt und in gleicher Weise versandt und trägt als Zusatzinformationen die ID des Materials und die geänderte Ausleihliste.

Damit muß der Client die Ausleihliste nicht selbst verwalten, wenn er beispielsweise die aktuellen Ausleiher eines Materials darstellen möchte. Anderseits kann er nur im Abgleich mit einer vorherigen Ausleihliste feststellen, welcher Benutzer neu hinzugekommen oder weggefallen ist. Eventuell bleibt hier zu überlegen, ob der betroffene Benutzer noch als Parameter in die Nachricht aufgenommen werden soll.

6.2 Integration von Client-/Server-Fähigkeit: RMI

In der Konstruktion des Archivs zu dieser Arbeit wurde für die Implementation der verteilten Nutzung die Remote Method Invocation (*RMI*) eingesetzt, da es in Java ab Version 1.1 zur Verfügung steht und die einfachste Lösung darstellt. Die notwendige Konstruktion ist aber weitestgehend gekapselt, so daß sie auch leicht gegen eine andere Verteilungsmethode wie

CORBA ausgetauscht werden könnte. Die Verteilung selbst ist mit Hilfe des Proxy-Musters [Gamma et al. 96] gekapselt.

Um RMI benutzen zu können, müssen die archivierten Materialien sowie einige Hilfsklassen serialisierbar sein. Für die Materialien ist dies einfach zu erreichen, indem die Schnittstelle `hierArchivable`, die sie implementieren müssen, von `java.io.Serializable` erbt. Hier kann also die Nutzung der Verteilungsmethode nicht verborgen werden; man bedenke aber, daß die Nutzung anderer Verteilungsmethoden meist noch tiefergehende Eingriffe erfordert.

Alle weiteren über das Netz zu übertragenden Klassen des Archivs (z.B. `borrowList` als Ausleihliste) müssen selbständig `Serializable` implementieren, damit sie als Parameter für RMI-Aufrufe benutzt werden können. Im Zuge der Demonstration von Persistenz durch Serialisierung mußte jedoch sowieso jede Klasse des Archivs serialisierbar sein, so daß dies als Nebeneffekt automatisch erfüllt war.

RMI bildet eine Kapsel um den Archivar herum, wie in der Abbildung illustriert.

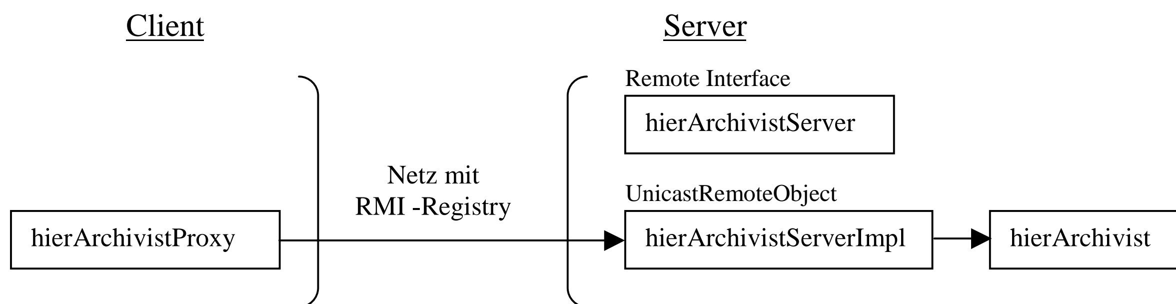

Abbildung 14: RMI-Kapsel um den Archivar

Auf der Server-Seite enthält dabei die Klasse `hierArchivistServerImpl` den Archivar als Referenz, während sie selbst das verteilt aufrufbare Objekt bildet. Dazu muß sie von der Klasse `java.rmi.server.UnicastRemoteObject` erben und eine Schnittstelle implementieren, welche alle verteilt aufrufbaren Methoden auflistet. Die Klasse hat dann lediglich die Aufgabe, alle an ihr aufgerufen Methoden direkt an den referenzierten wirklichen Archivar weiterzugeben, während sie selbst sich der Außenwelt als Archivar präsentiert. Ebenfalls synchronisiert sie die Aufrufe über das Java-Schlüsselwort `synchronized`, damit nicht zwei nebenläufige verteilte Aufrufe zu Inkonsistenzen im Archiv führen.

```
public class hierArchivistServerImpl extends UnicastRemoteObject implements
hierArchivistServer
{
        private hierArchivist _archivist;

        public hierArchivistServerImpl() throws RemoteException
        {
              super();
              _archivist = new hierArchivist();
```

```
		}

		public synchronized ID getArchiveID() throws RemoteException
		{
			return _archivist.getArchiveID();
		}
		...
	}
```

Diese Schnittstelle für `hierArchivistServerImpl` heißt `hierArchivistServer`. Sie muß von `java.rmi.Remote` erben und listet auf, mit welchen Methoden das verteilt aufrufbare Objekt gerufen werden kann. Schon bei der Bekanntmachung als verteiltes Objekt in RMI wird das Objekt nicht als `hierArchivistServerImpl`, sondern als `hierArchivistServer` angegeben.

Um das verteilt aufrufbare Objekt für die Benutzung zur Verfügung zu stellen, wird es von einer weiteren Klasse namens `hierArchivistServerApplication` bekanntgemacht, die anschließend in einer endlosen Warteschleife läuft und damit die laufende Server-Anwendung für das Archiv bildet. Der wichtigste Teil daraus sieht etwa wie folgt aus:

```
		hierArchivistServer _archivist = new hierArchivistServerImpl();
		Naming.rebind("hierArchivistServer", _archivist);

		... // endlose Warteschleife
```

Diese Klasse realisiert später leicht erweitert auch die Persistenz durch Serialisierung (siehe Kapitel 6.3).

Auf der Client-Seite versucht nun ein Objekt namens `hierArchivistProxy` bei der Erzeugung, über das Netz mit RMI Verbindung zum verteilt aufrufbaren Archivar-Objekt aufzunehmen. Bei Erfolg stellt nun der Proxy auf Client-Seite die gesamte Schnittstelle des Archivars lokal zur Verfügung und leitet die Methoden über das Netz weiter. Letztlich handelt es sich bei der RMI-Kapsel also um eine doppelte Indirektion.

```
	public class hierArchivistProxy
	{
		private hierArchivistServer _archivist;

		public hierArchivistProxy() throws Exception
		{
			System.setSecurityManager(new RMISecurityManager());
			try
			{
				_archivist = (hierArchivistServer)
						Naming.lookup("hierArchivistServer";);
			}
			catch (Exception e)
			{
				throw new Exception("hierArchivistServer not found");
			}
		}

		public ID getArchiveID() throws RemoteException
		{
```

```
                return _archivist.getArchiveID();
            }
            ...
    }
```

Für den Client, der das Archiv benutzt, ist die Anwendung des verteilten Aufrufs mit dem unterliegenden Netz vollkommen transparent, außer daß er einen Proxy erzeugen muß, statt des Archivars selbst. Erst wenn Fehler in der Netzverbindung auftauchen, werden diese auch an den Client weitergeleitet, der sie entsprechend behandeln muß.

Eine grundlegende Einführung in die Remote Method Invocation von Java findet sich in [Sun RMI].

6.3 Integration von Persistenz durch Serialisierung

Die Realisation von Persistenz durch einfache Serialisierung ist die einfachste schon in der Sprache Java enthaltene Lösung, um das Archiv zu sichern, hat aber einige Nachteile, so daß die Implementation nur zu Demonstrations- und Vergleichszwecken mit PJama konstruiert wurde.

Um das Archiv persistent zu machen, müssen alle im Archiv benutzen Klassen die `Serializable`-Schnittstelle implementieren. Sie müssen dazu aber keine speziellen Methoden einsetzen, außer wenn beispielsweise Klassenattribute mitgesichert werden sollen, so wie in der Implementation des Archivars.

Die Tatsache, daß bei der Serialisierung keine statischen Attribute von Klassen gespeichert werden, ist einer der Nachteile dieser Lösung. Sowohl für die Umsetzung des Singleton-Musters [Gamma et al. 96], welches gut geeignet wäre, um ein einziges Archiv pro Prozeßraum zu garantieren, als auch für einen statischen Zähler zur Vergabe von IDs bedeutete diese Lösung Probleme.

Da das Archiv mit seinen Komponenten wie zuvor gesehen nur einmal über eine Applikation gestartet wird, wurde in der Konstruktion zugunsten der leichteren Handhabung bei der Serialisierung auf das Singleton-Muster vorerst verzichtet. Um den ID-Zähler persistent machen zu können, wurde dieser aus der Klasse ID in den Archivar verlagert, da die ID-Objekte häufig als Parameter bei Aufrufen am Archivar durch RMI serialisiert werden müssen und daher eine Überladung der Standard-I/O-Routinen in der Klasse `ID` zu Inkonsistenzen im Zähler führte. Der Archivar dagegen wird niemals über das Netz transferiert und an diesem ließen sich gefahrlos die Methoden `writeObject` und `readObject` überschreiben.

Alle weiteren für die Persistenz notwendigen Eingriffe können in dieser Lösung aber aus der Archiv-Konstruktion herausgehalten werden, indem der zusätzliche Aufwand in die Klasse `hierArchiveServerApplication` (siehe Kapitel 6.2) gesteckt wird, die dann beim Einsatz von PJama einfach gegen eine andere Realisation ausgetauscht wird.

Die Klasse `hierArchiveServerApplication`, die ursprünglich nur den Archivar bzw. seine Kapsel als verteilt aufrufbares Objekt registrierte und über die Dauer seiner Laufzeit verfügbar machte, wurde für die Realisierung von Persistenz durch Serialisierung einfach um zwei Methoden namens `readArchive` und `writeArchive` zum Lesen und Schreiben des verteilt aufrufbaren Objekts `hierArchiveServerImpl` erweitert. Es wird also nicht der wirkliche Archivar `hierArchivist` gesichert, da dieser ein privates Attribut seiner Netzwerkkapsel `hierArchiveServerImpl` ist und der Applikation in dieser Stufe nicht zur Verfügung steht. Diese Indirektion hat jedoch keinen Einfluß auf das Ergebnis.

Um Datenverluste zu vermeiden, sollten die Daten des Archivs nicht nur beim Hoch- und Herunterfahren des Archivars gesichert werden, sondern in regelmäßigen Abständen. Hierzu wurde die `hierArchiveServerApplication` zu einem Thread umgebaut, der in festen Intervallen `writeArchive` aufruft. Gelesen wird das Archiv dagegen nur ein einziges Mal beim Start dieser Klasse, und zwar bevor es der RMI-Registry bekannt gemacht wird.

Zusammenfassend kann man als Vorteile der Lösung per Serialisierung die einfache Implementation durch die Zugehörigkeit zur Sprache Java und das vergleichsweise geringe Datenvolumen für die persistenten Daten aufzählen, während sich auf der anderen Seite folgende Nachteile ergeben:

Einfache mit Serialisierung erzeugte Textdateien haben keinerlei Datenbankeigenschaften für den Zugriff und keine Sicherheits- oder Transaktionskonzepte zur Unterstützung. Ebenfalls ist keine Möglichkeit zur Datensynchronisation oder inkrementellen Sicherung gegeben und diese müßte ebenso wie angepaßte I/O-Routinen manuell ergänzt werden. Insbesondere das Fehlen der inkrementellen Sicherung (wobei nur geänderte Daten abgespeichert werden) wirkt sich fatal auf das Laufzeitverhalten bei großen Datenmengen aus und läßt die Serialisierung als Lösung in diesem Fall ausscheiden.

6.4 Integration von Persistenz mit PJama

PJama bietet wahrscheinlich die zur Zeit beste Lösung zur Verwendung orthogonaler Persistenz unter Java. Handelte es sich dabei aber um wirklich konsequent umgesetzte orthogonale Persistenz, so könnte man in diesem Kapitel höchstens die Anwendung des modifizierten Java-Interpreters aufzeigen und die Konstruktion wäre völlig unverändert. Dies ist aber, wie in Kapitel 5.2.3 und 5.2.4 beschrieben, nicht der Fall, so daß dieses Kapitel Eingriffe und Erweiterungen in die Konstruktion des Archivs beschreibt, die durch die Integration von PJama nötig werden.

6.4.1 Anwendung der Entwicklungsumgebung von PJama

Da PJama bislang eine vom Java Development Kit (*JDK*) getrennte Entwicklung darstellt und noch nicht in dieses integriert wurde, muß man für die Verwendung orthogonaler Persistenz

eine speziell für PJama angepaßte Version des JDK verwenden, die sich in der Solaris-Version als ein Patch für das JDK 1.1.6 und in der Windows-NT-Version als eine auf dem JDK 1.1.6 basierende eigenständige Version darstellt. PJama selbst liegt zur Zeit in der Version 0.5.6.9 vor.

Im folgenden wird die technische Anwendung der Solaris-Version beschrieben. Die Windows-NT-Version scheint noch weniger stabil und mit mehr Fehlern behaftet, ist aber in der Anwendung genauso zu bedienen und wird daher nicht näher erläutert.

Zuerst müssen die Umgebungsvariablen angepaßt werden; dazu müssen `PATH` um die ausführbaren Programme von PJama und `CLASSPATH` um die PJama-Klassen, welche die Klassen des JDK überladen, erweitert werden (vgl. [Sun PJama]).

Zur Arbeit mit PJama gehört die Erzeugung eines leeren Stores, in dem alle persistenten Klassen und Objekte abgelegt werden können. Dies kann auf zwei verschiedene Weisen geschehen. Entweder man schreibt eine kleine Applikation, die den Store explizit neu anlegt,

```
PJStore ps = new PJStoreImpl("/home/test/hierArchive.pjs");
...
System.exit(0);
```

oder man ruft ein mit PJama mitgeliefertes Skript zum Erzeugen eines leeren Stores auf:

```
opjcs /home/test/hierArchive.pjs
```

Weitere Applikationen können dann mit diesem Store arbeiten, indem man ihn dem gepatchten Java-Interpreter `opj`, den man nun statt `java` benutzt, beim Start der Java-Applikation als Parameter mit übergibt

```
opj -store /home/test/hierArchive.pjs initStore
```

oder zuvor in einer Umgebungsvariablen setzt:

```
setenv PJSTORE /home/test/hierArchive.pjs
opj initStore
```

6.4.2 Erweiterungen in der Konstruktion des Archiv

Zur Nutzung von Persistenz mit PJama mußten am Archiv selbst keine Änderungen vorgenommen werden. Statt dessen wurde nur die Applikation zum Starten des Archivars ersetzt gegen zwei Klassen zum Initialisieren und Betrieb des PJama-Stores. Dies wird im folgenden demonstriert, bevor im nachfolgenden Kapitel das Zusammenspiel mit dem Verteilungsmechanismus RMI zum Tragen kommt.

Bei PJama muß dem Store explizit kenntlich gemacht werden, welche Objekte persistent gemacht werden sollen. Dazu registriert man, wie in Kapitel 5.2.2. gesehen, die obersten Wurzel-Objekte am Store-Objekt. Diese Registrierung wird in der Konstruktion des Archivs in der Klasse `initStore` vorgenommen.

```
PJStore pjs = PJStoreImpl.getStore();
...
hierArchivist _archivist = new hierArchivist();
pjs.newPRoot("Archivist", _archivist);
...
System.exit(0);
```

Bei einem Verlassen der Applikation ohne Fehler werden alle registrierten Objekte, deren Klassen und alle von diesen aus erreichbaren Objekte und deren Klassen im Store gespeichert. Möchte man bestimmte Objekte transient lassen, obwohl sie von persistenten Objekten aus erreichbar sind – das gilt beispielsweise für statische Variablen eines Objekts, welche man als Attribute des Klassenobjekts auffassen kann – so muß man deren Transienz explizit markieren:

```
// mache eine Beispielfeld aus dem Archivar transient
PJSystem.markTransient(hierArchivist.class, "_tamc");
```

Wenn die virtuelle Maschine mit Fehler beendet wird (z.B. über `exit(1)` oder den nicht gefangenen Wurf einer Exception), so wird der Store nicht aktualisiert und bleibt im Zustand vor allen Änderungen durch die Applikation, so daß die Konsistenz der Daten im Store bei Programmfehlern gewährleistet bleibt.

Es wäre natürlich denkbar, Erzeugung und Initialisierung des Stores, also dessen „Bevölkerung" (*population*) in einem einzigen Programm durchzuführen. Bei einem Test zeigte sich aber, daß ein Programm den Store nicht stabilisieren kann (dazu später mehr), wenn es ihn selbst erzeugt hat.

Im obigen Code-Beispiel zu `initStore` wurde der Archivar und alle benutzten Arbeitsgegenstände, also beispielsweise seine Bestandsliste und das Wurzel-Material Archiv, angelegt und über die Erreichbarkeit durch den Archivar als persistent gekennzeichnet. Jede weitere Applikation kann nun den Store mit dem enthaltenen persistenten Archivar verwenden,

```
opj –store hierArchive.pjs runStore
```

und über den registrierten Namen des Archivars auf diesen zugreifen.

```
PJStore pjs = PJStoreImpl.getStore();
hierArchivist _archivist = (hierArchivist) pjs.getPRoot("Archivist");
```

Alle Änderungen durch die Applikation werden beim Beenden ohne Fehler automatisch gesichert. Doch die ansonsten implizite Sicherung kann auch erzwungen werden, indem der Applikationsprogrammierer die Methode

```
PJStore.stabilizeAll()
```

aufruft. Diese Methode und auch die Sicherung bei Programmende sind atomar; tritt also bei der Sicherung des Stores ein Fehler auf, so bleibt der Store stets in einem konsistenten Zustand.

6.4.3 Garbage Collection

Objekte, die von keinem Objekt aus mehr erreichbar sind, werden in Java durch einen sogenannten Garbage Collector gelöscht. Nicht persistente Objekte werden bei PJama weiterhin durch den normalen Garbage Collector entfernt. Um persistente nicht mehr erreichbare Objekte aus dem Store zu löschen, gibt es in PJama zur Zeit nur die Lösung über eine Anwendung, die manuell gestartet werden muß.

```
opjgc hierArchive.pjs
```

Die Anwendung kann aber nur auf einen nicht in Gebrauch befindlichen Store angewendet werden. Diese Einschränkungen sollen zu einem späteren Zeitpunkt in PJama ausgeräumt werden.

6.5 PJama im Zusammenspiel mit RMI

Um RMI zusammen mit PJama verwenden zu können, wurde im Rahmen des PJama-Projekts eine spezielle RMI-Version namens Persistent RMI (*PJRMI*) implementiert (siehe [Sun PJRMI]).

Die Anwendung von „normalem“ RMI ohne Persistenz ist natürlich auch weiterhin möglich, wenn der PJama-Interpreter ohne Store verwendet wird. Mit Store dagegen kommen die besonderen neuen Fertigkeiten von PJRMI zum Einsatz. Dabei unterstützt PJRMI Persistenz für alle verteilt aufrufbaren Objekte und deren Abfrage über einen an der Registry bekannt gemachten Namen. Weiterhin ermöglicht PJRMI automatische Exportierung von persistenten verteilt aufrufbaren Objekten bei Starten des Stores und automatische Wiederaufnahme der Verbindung zwischen persistenten Referenzen und verteilt aufrufbaren Objekten bei der ersten Verwendung der Referenz nach dem Starten des Stores.

Der Aufruf und die Verwendung der persistenten verteilten Objekte mit RMI-Applikationen, die im normalen JDK 1.1.6, auf dem PJama basiert, geschrieben wurden, soll weiterhin möglich sein[9].

Für eine Applikation soll der Archivar als verteilt aufrufbares persistentes Objekt zur Verfügung stehen. Wie im vorigen Kapitel beschrieben, soll wiederum das Erzeugen und Registrieren einerseits und das zur Verfügung Stellen im laufenden Betrieb andererseits in zwei getrennten Applikationen durchgeführt werden. Hierzu wird die Klasse `initStore` erweitert und für den laufenden Betrieb die Klasse `runStore` hinzugefügt.

Die Initialisierung des Stores wird um die Erzeugung einer eigenen Registry erweitert, die wir selbst als persistentes Wurzel-Objekt im Store einhängen.

---

[9] nach Aussage von Susan Spence, Mitautorin von PJRMI

```
Registry _registry = new RegistryImpl(Registry.REGISTRY_PORT);
PJStore   pjs = PJStoreImpl.getStore();
pjs.newPRoot("Registry", _registry);
(PJStoreImpl.getActionManager()).bind((PJActionHandler) _registry);
```

Nach der anschließenden Erzeugung und Exportierung der RMI-Kapsel des Archivars, in der der eigentliche Archivar erzeugt wird, müßte die Kapsel eigentlich nicht mehr explizit als persistent markiert werden. Dies liegt daran, daß die Registry intern Tabellen über die in ihr verwalteten Namen und Objekte führt, über die der Archivar als registriertes Objekt natürlich erreichbar ist. Und da die Registry bereits persistent ist, wäre es somit auch der Archivar.

```
hierArchivistServer _archivist = new hierArchivistServerImpl();
Naming.rebind("hierArchivistServer", _archivist);
```

Damit nun aber die mit dem Store laufenden Applikationen den Archivar direkt „greifen“ können, muß er unter einem Namen im Store kenntlich gemacht werden. Deshalb – und nicht aus Gründen der Persistenz – wird er als persistentes Wurzel-Objekt markiert.

```
pjs.newPRoot("Archivist", _archivist);
```

Die Anwendung `runStore` in ihrer ersten Fassung muß keinerlei Funktionalität besitzen. Sie realisiert nur eine Endlosschleife, die dafür sorgt, daß der mit ihr gestartete Store ständig zur Verfügung steht. Denn mit dem Store wird auch die persistente Registry gestartet, und der verteilte aufrufbare Archivar steht für Anfragen ebenfalls zur Verfügung.

6.5.1 Erweiterungen zur Datensicherung und Beendigung des Stores

Damit der Store in einer Form von Serverbetrieb laufen kann, wurde der Konstruktion ein verteilt aufrufbares Objekt namens `adminServer` hinzugefügt, welches Dienste zur Administration des Stores zur Verfügung stellt.

An der verteilt aufrufbaren Schnittstelle von `adminServer` stehen zwei Methoden zur Verfügung.

```
public void stabilize() throws java.rmi.RemoteException;
public void suspendAndQuit() throws java.rmi.RemoteException;
```

Die Methode `stabilize` bewirkt eine Aufruf von `stabilizeAll` am Store-Objekt, so daß der Store ohne Beenden der Server-Applikation im Betrieb gesichert werden kann.

Die Methode `suspendAndQuit` implementiert eine sauberes Herunterfahren des Stores und damit des Archiv-Servers. Statt einfach nur den Store zu beenden, wird ein Thread gestartet, der unabhängig von dem verteilten Aufruf weiterläuft und den Store nach einer kurzen Pause herunterfährt. Diese Pause wird benötigt, um den verteilten Netz-Aufruf am persistenten Administrationsobjekt noch sauber abzuschließen.

Damit die Administration über `adminServer` eingesetzt werden kann, erweitert man die Klasse `initStore` um die Erzeugung und Registrierung des `adminServer`-Objekts. Dieses ist, wie oben begründet, implizit persistent.

```
adminServer _adminServer = new adminServerImpl();
UnicastRemoteObject.exportObject(_adminServer);
Naming.rebind("adminServer", _adminServer);
```

Mit zwei kleinen Applikation, die über RMI am verteilt aufrufbaren Administrationsobjekt nur die gewünschte Methode aufrufen, können die Administrationsfunktionen dann eingesetzt werden.

6.5.2 Zusammenspiel mit Messaging-Mechanismus des Frameworks JWAM

Zur Koordination zwischen den Benutzern des Archivs wird der im Framework JWAM vorhandene Kommunikationsdienst, verkörpert durch den MessageBroker, eingesetzt.

Zum Verschicken von Nachrichten aus einem Puffer heraus setzt das Framework jedoch einen Thread ein, so daß dieser Dienst mit der jetzigen PJama-Version, die keine persistenten Threads unterstützt, nicht ohne weiteres persistent gemacht werden kann.

Der transienten Benutzung des Dienstes steht andererseits entgegen, daß einige Klassen des Kommunikationsdienstes nach dem Singleton-Muster gebaut wurden. Da im Singleton-Muster das einzige Exemplar einer Klasse in einer statischen Variable referenziert wird, wird dadurch bei Persistenz der Klasse auch das Singleton-Exemplar persistent, was es aber zu vermeiden gilt. Unglücklicherweise reichte durch die transitive Persistenz die Verwendung einer einzigen Klasse aus dem Framework-Kommunikationsdienst in einer persistenten Klasse des Archivs aus, um alle in Frage kommenden Singleton-Klassen als persistent zu markieren.

Um diese Konstruktionsproblematik zu lösen, gibt es zur Zeit als Lösung die Möglichkeit, die statischen Exemplar-Felder aus diesen Klassen des Kommunikationsdienstes explizit bei der Initialisierung des Stores in `initStore` als transient zu kennzeichnen, obwohl die Felder privat sind und eigentlich den außenstehendem Klassen nicht bekannt sein sollten. Hierbei handelt es sich letztlich um eine sehr unbefriedigende Lösung, da sie das Archiv eng mit dem Framework verzahnt, obwohl dies eigentlich unnötig sein sollte.

```
PJSystem.markTransient(wam.toolconstruction.coordination.
        ToolAutomatonMessageCoordinator.class, "_instance");
PJSystem.markTransient(wam.distribution.messaging.MessageBroker.class,
        "_instance");
```

Bei der Markierung transienter Attribute ist wichtig, daß diese *vor* einer Verwendung durch andere Objekte markiert werden, also in diesem Fall bevor der Archivar, der sie verwendet, instanziiert wird, da die Markierung sonst nicht funktioniert[10].

Da der Kommunikationsdienst des Frameworks intern auf RMI aufsetzt, muß zu seiner Verwendung ein `MessageBrokerServer` als verteilt aufrufbares Objekt verfügbar sein, der von allen Clients, die Nachrichten verschicken wollen, aufgerufen werden kann. Die Erzeugung und Registrierung an der persistenten Registry geschieht in der vorliegenden Konstruktion in `runStore`, da es sich ja um ein transientes Objekt handeln soll, welches bei jedem Starten des Stores neu erzeugt werden muß.

```
public static MessageBrokerServerImpl _mbs;
...
PJSystem.markTransient(runStore.class, "_mbs");
_mbs = new MessageBrokerServerImpl();
Naming.bind ("WAM-Message-Broker-Server", _mbs);
```

Eine weitere Lösungsmöglichkeit könnte das Einbinden jeweils eines sogenannten `PJActionHandler` für eine Klasse nach Singleton-Muster sein. Diese Routinen können durch den `PJActionManager` beim Starten des Stores automatisch aufgerufen werden und die statischen Exemplar-Felder der Singleton-Klassen jeweils neu initialisieren. Auch bei dieser Lösung wäre PJama aber eng mit dem Framework verbunden, da sonst auch diese Callback-Routinen nicht auf die privaten Felder der Framework-Klassen zugreifen können.

6.5.3 Probleme mit einer nicht persistenten Registry

Um dem Problem der nicht möglichen Persistenz des Kommunikationsdienstes des Frameworks zu entgehen, wurden Tests durchgeführt, die mit einer nicht persistenten Registry arbeiten. Im Store sollte hierbei lediglich das Archiv und das Administrations-Objekt gespeichert werden. Der `MessageBrokerServer` könnte dann in einem eigenen Prozeßraum laufen und an derselben nicht persistenten Registry angemeldet sein.

Eine externe Registry für persistente verteilt aufrufbare Objekte im Store zu benutzen, ist in der gegebenen Version von PJRMI aber nicht möglich:

Um die Illusion von persistenten Referenzen zu verteilt aufrufbaren Objekten zu erhalten, ist es notwendig, einen Service in der Implementation von PJRMI zur Verfügung zu stellen, der Referenzen zu persistenten verteilt aufrufbaren Objekten erneuert, wenn sie das erste Mal nach dem Neustarten des Stores benutzt werden. Die verteilt aufrufbaren Objekte werden nach einem Neustart erneut exportiert, um die Illusion eines durch sie gegebenen persistenten Service zu erhalten. Die entfernten Referenzen auf sie werden ebenfalls erneuert, wobei ein persistenter PJRMI Service verwendet wird, um die neue Portnummer für ein verteilt aufrufbares Objekt nach einem Neustart zu ermitteln.

---

[10] PJama erzeugt eine Exception und bewirkt einen Abbruch des Programms.

Um den PJRMI Service, der die entfernten Referenzen auf verteilt aufrufbare Objekte erneuert, zu bilden, wird ein verteilt aufrufbares Objekt, welches diesen Service unterstützt, von `PJamaPJExported` erzeugt und an einer persistenten lokalen Registry angemeldet. Auf diese Weise wird der Service persistent gemacht, und alle Clients können ihn immer über die persistente Registry abfragen.

Wenn dieser Service nicht über eine Registry, sondern als komplett eigener Dienst angeboten würde, also wie eine Registry einen eigenen Port hätte, bräuchte man keine persistente Registry. Die Autoren von PJRMI haben sich aber dagegen entschieden, neben der bestehenden Registry noch einen anderen in seiner Art sehr ähnlichen Dienst einzuführen und somit noch weitere Konventionen in RMI einzuflechten.

Daher wird in der jetzigen Version von PJRMI der oben beschriebene persistente Service zur Aufrechterhaltung der Illusion von persistenten Referenzen auf persistente verteilt aufrufbare Objekte unterstützt. Dieser Service kann nur wirklich persistent zur Verfügung stehen, wenn es für einen Client immer möglich sein soll, eine Referenz auf ihn zu erhalten. Somit muß die Registry, die den Zugriff auf den Service unterstützt, ebenfalls persistent sein. Mit einer externen Registry wäre diese Konstruktion von PJRMI nicht möglich.

Eine letzte Alternative bildet eine Lösung mit zwei Registries, die auf unterschiedlichen Ports laufen; eine persistente Registry im Store für den Archivar und die Administration des Stores und eine Registry für den Kommunikationsdienst des Frameworks. Diese Lösung wurde aber nicht weiter durchkonstruiert, da zwei Registries einen unverhältnismäßigen Aufwand für das Archiv bilden.

6.6 Beurteilung der Verwendbarkeit von PJama

Bislang scheint die Kombination von RMI und PJama noch nicht ganz optimal. Probleme wie die Persistenz von Threads, die noch nicht durchgeführte Integration von PJama in das Standard-JDK, die notwendige explizite Markierung von Feldern als transient und nicht zuletzt gelegentliche Abstürze beim Speichern des Stores lassen die Verwendbarkeit von PJama noch nicht so sicher und einfach erscheinen, wie ein System mit orthogonaler Persistenz sein sollte. Auch eine Kapselung der Persistenz in der Konstruktion mit PJama und RMI ist wie gesehen noch nicht befriedigend möglich.

6.7 Beurteilung des hierArchive bezüglich der Entwurfsziele

In diesem Kapitel sollen noch einmal die in Kapitel 4.8 aufgestellten Entwurfsziele für ein softwaretechnisches Archivs aufgegriffen werden und auf die beschriebene Implementation angewandt werden.

- Die Konstruktion des Archivs soll möglichst autark sein, aber trotzdem leicht in eine Anwendung einzubetten.

An den Schnittstellen des Archivs werden nur zu Java gehörende oder selbst konstruierte Objekte verwendet, um das Archiv nicht vom verwendeten Framework abhängig zu machen. Da es jedoch intern die jConLib des Frameworks und dessen Kommunikationsdienst verwendet, ist das Entwurfsziel zugunsten einer einfacheren Implementation und Kompatibilität zum Framework eingeschränkt worden.

- Der Aufwand zur Anpassung von Materialien zur Speicherung im Archiv soll minimal sein. Am besten wäre, wenn jedes Material ohne Anpassung im Archiv gespeichert werden könnte.

Das Ziel, Materialien ganz ohne Anpassung archivieren zu können, konnte nicht erfüllt werden, da das Archiv eine allgemeine Schnittstelle zur Handhabung der Materialien braucht. Diese Schnittstelle wurde aber so klein und einfach[11] wie möglich gehalten. Durch die Konstruktion nach dem Composite-Muster müssen nur die Behälter eine komplexere Schnittstelle erfüllen, während die einfachen Materialien und fachlichen Anteile mit der einfachen Schnittstelle auskommen.

- Das Archiv soll eine Unterstützung von Kooperation gewährleisten. Am besten wäre die Unterstützung aller Kooperationsmodelle und aller Formen des konkurrierenden Zugriffs zur Wahl durch den Anwender.

Das Archiv unterstützt in seiner Konstruktion bei den Formen des konkurrierenden Zugriffs nur die nur-Kopie-Logik, um die bei den anderen Zugriffsarten in Kapitel 4.1 aufgezeigten Probleme zu vermeiden.

- Das Archiv soll persistent sein mit einem leistungsfähigen Persistenzmechanismus, der auch den schnellen Zugriff auf Daten und die Speicherung größerer Datenmengen erlaubt.

Mit PJama steht ein recht guter Mechanismus für Persistenz zur Verfügung, der jetzt schon eine schnellen Datenzugriff und die Möglichkeit der Speicherung größerer Datenmengen erlaubt. In zukünftigen Versionen soll dies noch weiter verbessert werden, damit PJama sich auch mit herkömmlichen Datenbanksystemen messen kann, zumindest was Zugriff und Datenmenge betrifft. Dieses Entwurfsziel kann somit als erreicht gelten.

- Die Verwendbarkeit des Archivs über ein Netzwerk soll gewährleistet sein, wobei die verwendete Netztechnologie keine Rolle spielen sollte oder sichtbar sein soll.

[11] Die Schnittstelle wurde im Vergleich zu einer älteren Archiv-Konstruktion der Autoren stark verkürzt.

Mit der konstruierten RMI-Kapsel um das Archiv ist die Verwendung des Archivs im Netz ohne Einschränkung möglich und fast unsichtbar – sie wird nur bei Netzfehlern spürbar. Die Kapsel kann bei Bedarf gegen einen anderen Verteilungsmechanismus ausgetauscht werden. Jedoch ist fraglich, ob dieser bessere Eigenschaften als RMI bezüglich Sichtbarkeit aufweisen kann. Das Entwurfsziel konnte somit erfüllt werden.

- Die Handhabbarkeit von strukturierten (komplexen) Materialien im Archiv soll gegeben sein.

Mit der Entwicklung einer Schnittstelle für archivierbare Materialien können diese im Archiv gehandhabt werden. Bis auf Einschränkungen, wie den Verlust der fachlichen Funktionalität bei der Nutzung über die Schnittstelle des Archivs ohne Herausnahme des Materials, ist dieses Entwurfsziel erfüllt worden.

Als Ergebnis der Betrachtung sei zusammengefaßt, daß die reale Konstruktionen des Archivs viele Entwurfsziele erfolgreich implementieren konnte. Die Nichterfüllung der übrigen Ziele ist zum Teil zugunsten der einfacheren Implementation bewußt in Kauf genommen worden, zum Teil waren die Ziele nicht erfüllbar, da sie in Konflikt zu anderen Zielen standen, wie beispielsweise bei der Speicherung unangepaßter Materialien gegen die strukturierte Handhabbarkeit durch das Archiv.

Vor dem Hintergrund der Tatsache, daß mit dieser Arbeit einiges Neuland beschritten wurde – die Arbeit mit Archiven, der Umgang mit behälterartigen Materialien und nicht zuletzt PJama in praktischer Anwendung – kann trotz der genannten Einschränkungen gesagt werden, daß die Arbeit ingesamt das gewünschte Ziel erreicht hat – die Konstruktion eines einsatzfähigen persistenten Archivs zur kooperationsunterstützenden Verwaltung behälterartiger Materialien.

7 Anhang

7.1 Literaturverzeichnis

[Atkinson et al. 90] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott und R. Morrison. An approach to persistent programming. In Readings in Object-Oriented Database Systems, Seiten 141-146. Morgan-Kaufmann, 1990.

[Atkinson 91] M.P. Atkinson. A vision of persistent systems. In Proceedings of the 2nd International Conference on Deductive and Object-Oriented Databases, Seiten 453-459, Berlin, 1991. Springer, Lecture Notes in Computer Science 566

[Atkinson et al. 95] M.P. Atkinson, R. Morrison. Orthogonally Persistent Object Systens. In VLDB Journal 4 (3), Seiten 319-401, ISSN: 1066-8888

[Atkinson et al. 96a] M.P. Atkinson, M.J. Jordan, L. Daynès, S. Spence. Design Issues for Persistent Java: a type-safe, object-oriented, orthogonally persistent system. Proceedings of POS7, Cape May, New Jersey, Mai 1996.

[Atkinson et al. 96b] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis, S. Spence. An Orthogonally Persistent Java. SIGMOD Record, 25 (4), Seiten 68-75, Dezember 1996.

[Bäumer 98] Dirk Bäumer. Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme. Disserationsschrift zur Vorlage am Fachbereich Informatik der Universität Hamburg, Januar 1998.

[Bohlmann 98] Holger Bohlmann. Realisierung einer Behälterbibliothek in Java: die jConLib. Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1998.

[Danielsen 98] Asbjørn Danielsen. The Evolution of Data Models and Approaches to Persistence in Database Systems, Mai 1998. http://www.stud.ifi.uio.no/~asbjornd/Essay.htm

[Floyd 97] Christiane Floyd. Einführung in die Softwaretechnik. Scriptum zur gleichnamigen Vorlesung, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Hamburg, 1997.

[Fricke 98] Niels Fricke. Der Umgang mit einem Server für WAM-Entwicklungsdokumente (vorläufiger Titel). Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, voraussichtlich 1998.

[Gamma et al. 96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Entwurfsmuster: Elemente wiederverwendbarer objektoriertierter Software. Bonn, 1996. Addison-Wesley.

[Heuer 92] Andreas Heuer. Objektorientierte Datenbanken: Konzepte, Modelle, Systeme, Seiten 125f, 451-453, Bonn; München; Paris [u.a.], 1992. Addison-Wesley.

[Jordan 96] Mick Jordan. Early Experiences with Persistent Java. Sun Microsystems Laboratories, Mountain View, 1996.

[JWAM] JWAM-Homepage. Einführung, Überblick und Dokumentation. http://swt-www.informatik.uni-hamburg.de/Software/JWAM/

[Lippert 97] Martin Lippert. Konzeption und Realisierung eines GUI-Frameworks in Java nach der WAM-Metapher. Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universistät Hamburg, 1997.

[Mashburn et al. 94] H.H. Mashburn, Satyanarayanan. Recoverable Virtual Memory. RVM Release 1.3, CMU, Januar 1994.

[Meyer 90] Bertrand Meyer. Objektorientierte Softwareentwicklung. Prentice-Hall, London, 1990.

[Stonebraker et al. 90] M. Stonebraker, L.A. Rowe, B. Lindsey, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech. Third-generation Database System Manifesto. SIGMOD Record, 19 (3), September 1990.

[Sun PJama] Sun Microsystems Laboratories, Mountain View. PJama Tutorial. http://www.sunlabs.com/research/forest/opj.tutorial.tutorial.html

[Sun PJRMI] Sun Microsystems Laboratories, Mountain View. PJRMI Tutorial. http://www.sunlabs.com/research/forest/opj.tutorial.pjrmidoc.html

[Sun RMI] Sun Microsystems Laboratories, Mountain View. RMI Documentation and Reference. http://www.javasoft.com/products/jdk/rmi/index.html

[Weiß 97] Ulfert Weiß. Konzeption und technische Weiterentwicklung eines objektorientierten Frameworks nach der Werkzeug-Material-Metapher. Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997.

[Züllighoven 98] Heinz Züllighoven. Das objektorientierte Konstruktionshandbuch. Nach dem Werkzeug & Material-Ansatz. Dpunkt-Verlag, Heidelberg, 1998.