Konzepte zur Materialversorgung verteilter Werkzeugumgebungen am Beispiel der Anbindung einer objektorientierten Datenbank

Studienarbeit
am
Fachbereich Informatik,
AB Softwaretechnik,
Universität Hamburg
März 1994

Betreuer:
Guido Gryczan

Martina Wulf
Grootmoor 90
22175 Hamburg
Matr.Nr.: 3 102 384

Dirk Weske
Petersdorfstr. 7b
21079 Hamburg
Matr.-Nr.: 4 060 944

Inhaltsverzeichnis

Inhaltsverzeichnis ..... **I**

Erklärung grafischer und textueller Konventionen ..... **III**

1. Einleitung **1**

2. Eine Werkzeugumgebung für die evolutionäre partizipative Systementwicklung **2**

2.1. Motivation und Zielsetzung 2

2.2. "Werkzeug und Material" als Entwurfsmetapher bei der Entwicklung 2

2.2.1. Von Werkzeugen, Materialien und Aspekten 3

2.2.2. Die Architektur von Software-Werkzeugen 3

2.2.3. Automaten 4

2.3. Der Weg vom autonomen Werkzeug zur integrierten Werkzeugumgebung 5

3. Ein Konzept zur Materialversorgung innerhalb von Werkzeugumgebungen **6**

3.1. Von Materialverwaltern, Magazinen und Versorgern 6

3.2. Das Zusammenspiel zwischen Materialverwalter und Werkzeugen 8

3.3. Über den Umgang mit gemeinsamen Materialien: Kopien und Originale als Bilder bei einer exklusiven Zugriffskontrolle 9

4. Der Übergang zu einer verteilten Werkzeugumgebung **11**

4.1. Von der Notwendigkeit eines 'zentralen' Materialkoordinators 11

4.2. Das Zusammenspiel zwischen Werkzeugumgebung und zentralem Materialkoordinator 11

5. Die Anbindung einer objektorientierten Datenbank am Beispiel von ONTOS **14**

5.1. Grundkonzepte und -begriffe des objektorientierten Datenmodells 14

5.1.1. Persistente Objekte 14

5.1.2. Abstrakte Datentypen, Vererbung 14

5.1.3. Objektidentität 15

5.1.4. Referenzen und Referentielle Integrität 15

5.1.5. Transaktionen 15

5.2. Ein verfügbares System: das Beispiel ONTOS 16

5.3. Ein Datenbankautomat als spezialisierter Materialversorger 17

6. Von der fachlichen Modellierung zum technischen Klassenentwurf **19**

6.1. Das technische Klassenmodell 19

6.1.1. Die Automatenklasse `tManager` 19

6.1.1.1. Materialien anfordern 21

6.1.1.2. Materialien als Originale reservieren 22
6.1.1.3. Materialoriginale freigeben 22
6.1.1.4. Materialoriginale löschen, einfügen und ändern 23
6.1.2. Die Behälterklasse `tDepot` 23
6.1.2.1. Über die Materialien des Behälters iterieren 23
6.1.2.2. Materialien einfügen und löschen 24
6.1.3. Die Aspektklasse `tManageable` 24
6.1.3.1. Materialien identifizieren 27
6.1.3.2. Materialien abstrakt erzeugen und zuweisen 27
6.1.3.3. Materialien typsicher konvertieren 28
6.1.4. Die Klasse `tPattern` 28
6.1.5. Die Klasse `tRequest` 28
6.1.6. Die Automatenklasse `tSupplier` 29
6.1.6.1. Materialien bei externen Datenversorgern löschen, einfügen und ändern 30
6.1.6.2. Materialien von externen Datenversorgern rekonstruieren 30
6.1.6.3. Die Materialextension eines externen Datenversorgers erfragen 31
6.1.6.4. Die Materialtypen eines externen Datenversorgers erfragen 31
6.1.7. Die Aspektklasse `tStorable` 32
6.2. Softwaretechnische Lösungen für das Problem der Verteilung in kooperativen Werkzeugumgebungen 34
6.2.1. Client/Services-Architekturen: Verteilte Werkzeugumgebungen nach dem Dienstleistungsprinzip entwerfen 34
6.2.1.1. Inter-Prozeß-Kommunikation durch Sockets realisieren 35
6.2.1.2. Von Proxyobjekten und anderen Konzepten, die die Kommunikation zwischen `Materialverwalter` und `Materialkoordinator` verständlicher machen 35
6.2.2. Verteiltes Objektmanagement - Standardisierung verteilter Anwendungen im objektorientierten Paradigma 38

7. Das SchwarzeBrett - prototypisches Werkzeug in einer verteilten Werkzeugumgebung mit objektorientiertem Datenbankanschluß 41

7.1. Das Werkzeug 41
7.2. Systemeinstieg über die Einsprungroutine `main` 42
7.3. Die Anwendungsklasse `tUmgebung` 43
7.4. Die Klasse `tEnvironment` 43
7.5. Die Werkzeugklasse `tFKNotizBrowser` 43
7.6. Die Werkzeugklasse `tFKNotizEditor` 45
7.7. Die Materialklasse `tNotiz` 46
7.7.1. Implementierung des `Object`-Protokolls 46
7.7.2. Implementierung des `tStorable`-Protokolls 48
7.7.3. Implementierung des `tManageable`-Protokolls 48

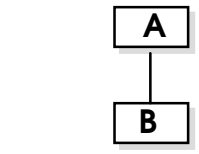
8. Schlußbemerkungen 49

Literaturverzeichnis 51

Erklärung grafischer und textueller Konventionen

`Entwurfskomponente` Fachliche und technische Entwurfskomponenten werden in Courier-Schrift dargestellt.

pManager Bezeichner von Klassenattributen werden in kursiver Schrift gedruckt.



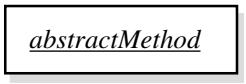
B *erbt von* A.

A *benutzt* B.

A *benutzt* und *beobachtet* B, B *meldet Ereignisse an* A.

Klassen des fachlichen und des technischen Entwurfs werden in rechteckigen Kästen dargestellt.

Objekte werden in rechteckigen Kästen mit abgerundeten Ecken dargestellt.



Klassenmethoden werden in kursiver Schrift dargestellt. Abstrakte Methoden sind darüberhinaus unterstrichen.

1. Einleitung

Die Thematik der vorliegenden Arbeit entspringt größtenteils der Beschäftigung mit Konzepten und Modellen für eine hypertextartige Werkzeugumgebung zur Unterstützung evolutionärer, partizipativer Softwareentwicklungsprozesse. Die prototypische Entwicklung einer solchen Umgebung bildete das Ziel eines zweisemestrigen studentischen Projektes, das der Arbeitsbereich Softwaretechnik im Wintersemester 1992/93 erstmalig an der Universität Hamburg durchgeführt hat.

In *Kapitel 2* stellen wir das Bild von *Werkzeugen und Materialien* als Entwurfsmetapher bei der Entwicklung objektorientierter Anwendungssysteme vor. Sie bildet eine der wesentlichen Grundlagen dieser Arbeit. Designerfahrungen, die in Zusammenhang mit der technischen Konstruktion eines fachlichen Anwendungskerns nach der *Werkzeug-Material* Metapher stehen, sind mittlerweile als Ergebnis von Studien- und Diplomarbeiten des Arbeitsbereiches in rahmenwerkartigen Klassenbibliotheken festgehalten worden (vgl. [Rie93a]).

Die vorhandenen Rahmenwerkbibliotheken decken bisher nicht den Bedarf an abstrakten Datenintegrationskonzepten. Insbesondere widmen sie sich nicht der Frage, wie der Anschluß von Werkzeugumgebungen an beliebige zentrale Datenversorger (etwa relationale oder objektorientierte Datenbanksysteme) und der Austausch von Daten zwischen Werkzeugen realisiert werden kann. In *Kapitel 3* stellen wir ein allgemeines Konzept vor, mit dem wir das Problem zunächst für Ein-Arbeitsplatz-Umgebungen zu lösen hoffen.

Kooperatives Arbeiten ist eine inhärente Eigenschaft partizipativer Softwareentwicklung und sollte bei der Unterstützung durch ein technisches System berücksichtigt werden. Daher vollziehen wir in *Kapitel 4* den Übergang zu einer verteilten Werkzeugumgebung und schlagen ein fachlich motiviertes Konzept vor, das den Umgang mit gemeinsam genutzten Daten in einer Mehr-Arbeitsplatz-Umgebung regelt, ohne dabei einen kooperativen Arbeitsstil zu verhindern.

Kapitel 5 schließlich veranschaulicht am Beispiel des kommerziell verfügbaren, objektorientierten Datenbanksystems ONTOS, wie externe Datenversorger im Rahmen des vorgestellten Materialversorgungsmodells konzeptionell an eine verteilte Werkzeugumgebung angeschlossen werden können.

In *Kapitel 6* überführen wir die entwickelten Konzepte in ein technisches Klassenmodell und zeigen, welche softwaretechnischen Lösungen sich für das Problem der *Verteilung* in kooperativen Werkzeugumgebungen finden lassen. Wir haben dieses Kapitel bewußt recht ausführlich gehalten, um potentiellen Entwicklern, die ihre Werkzeuge in das Versorgungsmodell einbinden möchten, eine technische Referenz an die Hand zu geben.

Zusätzliche Orientierungsmöglichkeit bietet darüberhinaus *Kapitel 7*, in dem wir als Anwendungsbeispiel ein prototypisches Werkzeug vorstellen, das in eine verteilte Werkzeugumgebung mit objektorientiertem Datenbankanschluß integriert worden ist.

In *Kapitel 8* schließlich geben wir im Rahmen der Schlußbemerkungen eine kurze Zusammenfassung und Einschätzung der Erfahrungen, die wir sowohl bei der *Umsetzung* als auch im *Umgang* mit dem entwickelten Materialversorgungsmodell gewonnen haben.

2. Eine Werkzeugumgebung für die evolutionäre partizipative Systementwicklung

2.1. Motivation und Zielsetzung

Die Idee einer Werkzeugumgebung für die evolutionäre partizipative Systementwicklung basiert wesentlich auf dem im Rahmen von STEPS[1] entwickelten Verständnis von Softwareentwicklungsprojekten. Im Mittelpunkt steht die Etablierung und Förderung eines Entwicklungsprozesses, der zu einem von Entwicklern und Benutzern gemeinsam getragenen Verständnis der jeweiligen Anwendung führt (vgl. [Flo91]).

Dokumenten kommt in diesem Entwicklungsprozeß eine herausragende Rolle zu, da sie das gemeinsam getragene Verständnis widerspiegeln sollen. Die Werkzeugumgebung bietet daher einen Satz von Werkzeugen für die Bearbeitung unterschiedlicher Dokumenttypen wie Szenarios, Glossare und Systemvisionen (vgl. [Gry92]) an.

Der Anspruch auf ein gemeinsam getragenes Verständnis der Anwendung kann selbstverständlich nur dann erhoben werden, wenn Erkenntnis- und Entscheidungsprozesse für alle Beteiligten transparent gemacht werden. Ein wesentliches Qualitätsmerkmal der Werkzeugumgebung als Hilfsmittel im Entwicklungsprozeß ist daher der Grad ihrer Verständlichkeit und Handhabbarkeit. Die Strukturierung in *Werkzeuge und Materialien* nimmt wegen der Analogie zu traditionellen menschlichen Arbeitsmitteln und -gegenständen einen beträchtlichen Einfluß auf die Transparenz der Werkzeugumgebung nicht nur bei der Benutzung, sondern auch bei der Herstellung selbst. Die Entwurfsmetapher von *Werkzeug und Material* bildet in dieser Hinsicht eine der wesentlichen Grundlagen dieser Arbeit und wird daher ausführlich im nächsten Abschnitt behandelt.

2.2. "Werkzeug und Material" als Entwurfsmetapher bei der Entwicklung[2]

Entwurfsmetaphern benötigen wir bei der Analyse eines Anwendungsbereiches und ihrer möglichst bruchlosen Überführung in einen Entwurf des zukünftigen Anwendungssystems. Die Metapher von *Werkzeug und Material* schafft dabei die Verbindung von den bisher am Arbeitsplatz vorhandenen und gewohnten Gegenständen zu den neuen Komponenten eines Anwendungssystems (vgl. [Kil93], S. 24) und ebnet so den Weg für die technische Unterstützung komplexer, qualifizierter Aufgaben und Arbeitsabläufe wie beispielsweise der Softwareentwicklung.

Zu den relevanten Gegenständen und Arbeitsformen der Softwareentwicklung zählen unterschiedliche Dokumente, die wir erstellen, verändern, ansehen, ablegen, usw. In einer Werkzeugumgebung für Softwareentwicklungsprojekte erwarten wir daher, Editoren, Ordner oder Browser wiederzufinden, mit denen diese Dokumente bearbeitet werden können.

In den folgenden Abschnitten geben wir einen kurzen Überblick über die wesentlichen Konzepte und Begriffe der Entwurfsmetapher. Darüberhinaus stellen wir einen Strukturierungsansatz für Werkzeuge vor, der unter Verwendung eines Beobachtermechanismus auf die vollständige Trennung von Funktionalität und Handhabung abzielt. Abschließend zeigen

---

[1] STEPS (**S**oftware**t**echnik für **E**volutionäre **P**artizipative **S**ystementwicklung) ist ein theoretischer und methodischer Ansatz der Softwaretechnik, der an der Technischen Universität Berlin entwickelt worden ist.

[2] Die in diesem Abschnitt vorgestellten Konzepte können ausführlich nachgelesen werden in [Bud90], [Bud91], [Bud92] und [Kil93].

wir, welche neuen Chancen und Möglichkeiten sich mit der Zusammenführung einst autonomer Werkzeuge in eine integrierte Werkzeugumgebung eröffnen.

2.2.1. Von Werkzeugen, Materialien und Aspekten

Im folgenden fassen wir im Sinne von Kilberth et al. (vgl. [Kil93], S.28ff.) zusammen, welches Verständnis mit den Begriffen von Werkzeug, Material und Aspekt verbunden ist.[3]

Werkzeuge

Werkzeuge dienen der interaktiven Bearbeitung von Materialien. Sie fokussieren dementsprechend auf die Handhabung von Arbeitsmitteln bzw. auf Arbeitstätigkeiten und beschreiben die Umgangsformen eines Benutzers mit der Anwendung.

Materialien

Materialien fokussieren auf die Eigenschaften von Arbeitsgegenständen. Sie sind in Begriffshierarchien organisiert, die ihre fachlich motivierten Zusammenhänge aufgreifen. Materialien sind *passiv*: sie können nur durch die aktiven Komponenten des Anwendungssystems, die Werkzeuge, verändert und sondiert werden.

Aspekte

Aspekte modellieren das Zusammenspiel von Werkzeugen und Materialien. Werkzeuge erkennen die zu ihnen *passenden* Materialien an einer Schnittstelle, die über ihre Eigenschaften das Funktionieren des Werkzeuges garantiert. Technisch wird diese Schnittstelle in Form von *abstrakten* Aspektklassen realisiert. Aspektklassen schaffen eine Entkopplung von Werkzeugen und Materialien: Werkzeuge benötigen kein Wissen über konkrete Materialien, sondern *benutzen* Aspekte, um auf sie zuzugreifen, während Materialien über die Schnittstelle ihrer ererbten Aspekte von beliebig vielen Werkzeugen bearbeitet werden können, ohne spezifische Annahmen über diese zu treffen.

2.2.2. Die Architektur von Software-Werkzeugen

Nach Kilberth et al. (vgl. [Kil93], S. 37ff.) läßt sich die Strukturierung eines Anwendungssystems in Werkzeuge und Materialien auf der Ebene von Werkzeugen durch die Trennung von Funktionalität und Handhabung noch fortführen. Ihr Architekturmuster sieht eine Zerlegung des Werkzeuges in seine Funktions- und Interaktionskomponente vor. Während die Interaktionskomponente das Werkzeug an der Benutzungsoberfläche repräsentiert, das bearbeitete Material darstellt und vom Benutzer ausgelöste Ereignisse empfängt, übernimmt die Funktionskomponente die eigentliche Sondierung und Bearbeitung des Materials.

Die Trennung von Interaktions- und Funktionskomponente wirft automatisch ein Kommunikationsproblem auf: von der Interaktionskomponente empfangene Ereignisse müssen letztlich in Methodenaufrufe der Funktionskomponente umgesetzt werden, während

[3] Die objektorientierte Analyse eines Anwendungsbereiches nach der *Werkzeug-Material* Metapher läßt sich nahezu bruchlos in einen softwaretechnischen Entwurf bestehend aus Werkzeug-, Material- und Aspektklassen überführen. Komponenten des fachlichen Entwurfs dienen als Grundlage für den technischen und heben auf diese Weise die klassische Trennung zwischen Analyse und Entwurf auf. Wir trennen daher fortan nur dann streng zwischen den fachlich motivierten Begriffen von Werkzeug, Material und Aspekt und ihren technischen Pendants, den Werkzeug-, Material- und Aspektklassen, wenn wir fachliche und technische Sicht explizit kontrastieren wollen.

umgekehrt durch die Funktionskomponente bewirkte Veränderungen am Material der Interaktionskomponente bekanntgemacht werden müssen, um eine konsistente Darstellung des Materials an der Benutzungsoberfläche zu gewährleisten. Eine tragfähige Lösung dieses Problems konnte mit dem sogenannten *Beobachter*-Konzept aufgezeigt werden. Wegen seiner grundlegenden Bedeutung für die Entwicklung des hier vorgestellten Materialversorgungskonzepts und die exemplarische Anbindung einer objektorientierten Datenbank gehen wir an dieser Stelle ausführlicher darauf ein.

Beobachter

(Software-) Werkzeuge sind ihrem Wesen nach *reaktiv*. Aktionen gehen vom Benutzer aus, werden von der Interaktionskomponente interpretiert und in sondierende und bewirkende Operationen der Funktionskomponente umgesetzt. Diesem Kontrollfluß entgegengesetzt geben Werkzeuge dem Benutzer Auskunft über ihren gegenwärtigen Zustand und den Zustand der von ihnen bearbeiteten Materialien.

Im Sinne des *Client-Server* Modells (vgl. [Wir90], S. 31ff.) verstehen wir diejenigen Komponenten, die Kontrolle abgeben, also anderswo Dienstleistungen *anfordern*, als Client und diejenigen Komponenten, die Kontrolle erhalten, also Dienstleistungen *erbringen*, als Server. Der Rückmeldefluß läuft entsprechend vom Server zum Client. Um einen Server weitestgehend unabhängig von seinem Verwendungskontext zu machen, sollte die Schnittstelle zu den anfordernden Komponenten, denen er Rückmeldung erstattet, minimal sein. Das Beobachterkonzept löst dieses Kontrollflußproblem folgendermaßen: ein Client trägt sich bei seinem Server als `Beobachter` ein. In dieser Eigenschaft erreichen ihn Rückmeldungen des Servers in Form von Ereignissen. Der `Beobachter` interpretiert das Ereignis und reagiert gegebenenfalls mit der Anforderung einer weiteren Dienstleistung. Typisches Beispiel ist eine als Client handelnde Interaktionskomponente, die die Darstellung des Werkzeuges an der Benutzungsoberfläche auffrischt, sobald von ihrer Funktionskomponente eine Änderung der Einstellung gemeldet wird. *Was* sich geändert hat, kann die Interaktionskomponente allerdings erst durch Aufruf einer entsprechenden sondierenden Operation der Funktionskomponente feststellen.

2.2.3. Automaten

Automaten ergänzen die Metapher von *Werkzeug und Material* um ein weiteres fachlich motiviertes Ausdrucksmittel. Ein Automat "ist dadurch gekennzeichnet, daß er, im Unterschied zum Werkzeug, mit einer bestimmten (veränderbaren) Einstellung im Hintergrund aktiv ist, d.h. er führt über einen längeren Zeitraum Aktionen auf der Basis eines vorgegebenen Algorithmus 'automatisch' durch" (vgl. [Kil93], S. 77).

Automaten können genau wie Werkzeuge in eine Funktions- und eine Interaktionskomponente strukturiert werden. Es ist offensichtlich, daß ein im Hintergrund aktiver Automat mit einer eher schlanken Repräsentation an der Benutzungsoberfläche auskommt. In der Regel bietet die Interaktionskomponente Benutzern die Möglichkeit, Parameterwerte zu setzen (vgl. [Bud90], S. 262f) sowie den Automaten zu starten und zu stoppen.

2.3. Der Weg vom autonomen Werkzeug zur integrierten Werkzeugumgebung

Die Entwurfsmetapher von *Werkzeug und Material* gibt uns Ausdrucksmittel für den Entwurf und die Konstruktion einzelner, autonomer Werkzeuge und ihrer Materialien an die Hand. Technische Systeme, die an qualifizierten Arbeitsplätzen komplexe Aufgaben unterstützen sollen, werden in der Regel nicht mit einem Werkzeug auskommen. Hier bedarf es einer integrierten *Werkzeugumgebung*, die dem Benutzer gleich eine ganze Reihe weitestgehend miteinander koordinierter Werkzeuge zur Verfügung stellt.

> "An integrated environment is a collection of software tools that work together, freeing the user from coordinating them manually. An integrated programming environment with tools for text editing, compiling, and debugging, for example, might ensure that when the debugger reaches a breakpoint the editor scrolls to the corresponding source statement. Integrating these tools relieves the user from the tedium of locating source files and line numbers, and supports instead the primary task, finding the bug." (vgl. [Sul92], S. 229f.)

Neben solchen Vorteilen, die sich aus der Sicht eines Anwenders ergeben, eröffnen sich mit der Integration von Werkzeugen in eine einbettende Umgebung aber auch aus softwaretechnischer Sicht ganz neue Perspektiven. Viele lästige Aufgaben, die sonst bei der Konstruktion eines jeden Werkzeuges erneut anfallen, können in einer integrierten Umgebung in werkzeugunabhängige Komponenten ausgelagert und dort gewissermaßen als Dienstleistungen den Werkzeugen zur Verfügung gestellt werden. Ein konkretes Beispiel hierfür sind etwa Komponenten, die von Datenbankintegrations- und Materialversorgungskonzepten abstrahieren und einzelne Werkzeuge so von dem Wissen um die komplexen Vorgänge bei der Beschaffung, Verwaltung und Speicherung von Materialien entlasten. Dies ist genau die Thematik, mit der wir uns im Rahmen dieser Arbeit intensiv auseinandersetzen wollen.

3. Ein Konzept zur Materialversorgung innerhalb von Werkzeugumgebungen

Im Fokus dieses Kapitels steht ein Konzept zur Versorgung der Werkzeuge einer Umgebung mit Materialien. Dabei ist es unser Hauptanliegen, die konkrete Herkunft von Materialien vor den Werkzeugen zu verbergen. Auch sollen aus Werkzeugsicht *transiente*, d.h. während der Lebensdauer einer Anwendung erzeugte und zerstörte Materialien und *persistente*, d.h. in beliebigen externen Datenspeichern über die Lebensdauer einer Anwendung hinaus abgelegte Materialien, bei der Materialbeschaffung und -bearbeitung gleichermaßen gehandhabt werden können.

Wir stellen zunächst die relevanten Komponenten des Konzeptes vor (Kapitel 3.1.) und betrachten dann ihr Zusammenspiel mit den Werkzeugen der Umgebung (Kapitel 3.2.). Anschließend überlegen wir, welche zusätzlichen Anforderungen erwachsen, wenn mehrere Werkzeuge gleichzeitig auf gemeinsame Materialien zugreifen (Kapitel 3.3.). Es wird sich zeigen, daß ähnliche Anforderungen beim Übergang von Ein- zu Mehrarbeitsplatz-Umgebungen auftreten. Diese Problematik werden wir im Zusammenhang mit verteilten Werkzeugumgebungen erneut aufgreifen.

3.1. Von Materialverwaltern, Magazinen und Versorgern

Exemplare von Werkzeugen, Automaten und Materialien können nur innerhalb einer einbettenden `Umgebung`[4] existieren, die sie erzeugt, verwaltet und zerstört. Die Aufgaben

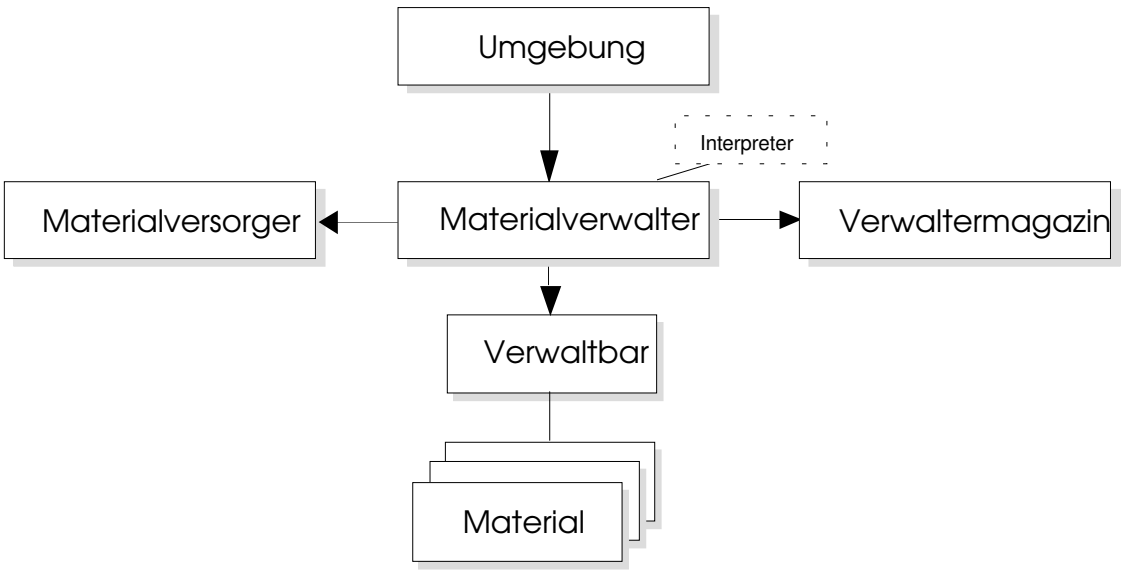


Abb. 1: Die Beziehungen zwischen den Komponenten `Umgebung`, `Materialversorger`, `Materialverwalter` und `Verwaltermagazin`.

der *Materialverwaltung* werden innerhalb von Werkzeugumgebungen mit Hilfe eines `Materialverwalters` gelöst, dem die Beschaffung, Verwahrung und Bereitstellung von Materialien obliegt.

[4] Wir verstehen die `Umgebung` hier als entwurfsmäßige Abstraktion einer integrierten Werkzeugumgebung, von der wir in Kapitel 2.3. ein skizzenhaftes Bild gezeichnet haben.

Die Beschaffung von Materialien erfolgt in der Regel am Rand der Umgebung, d.h. Materialien werden aus externen Datenspeichern (objektorientierte oder relationale Datenbanksysteme, Dateisysteme) gelesen oder bei anderen Umgebungen angefordert[5]. Der `Materialverwalter` kommuniziert dabei nicht selbst mit der Welt außerhalb seiner Umgebung, sondern *benutzt* dazu am Rande der Umgebung liegende Automaten. Mit Hilfe solcher Automaten kann der Materialverwalter von den eigentlichen Kommunikationsvorgängen - etwa dem Zugriff auf relationale oder objektorientierte Datenbanken - abstrahieren. Im folgenden werden wir diese Automaten gemäß ihrer Funktion als `Materialversorger` bezeichnen.

`Materialverwalter` und `Materialversorger` sind in der `Umgebung` verankert. Diese macht den `Materialversorger` nach seiner Erzeugung unter der vollen Schnittstelle bei dem `Materialverwalter` bekannt.

Materialversorgerklassen kapseln das Wissen um konkrete Schnittstellen zur Außenwelt. Das Protokoll ihrer gemeinsamen Oberklasse `Materialversorger` spezifiziert abstrakte Operationen, mit denen einerseits Materialien eines externen Bestandes gelöscht oder in der Umgebung rekonstruiert und andererseits Materialien der Umgebung in den externen Bestand überführt werden können. Sie werden von den abgeleiteten Klassen in Abhängigkeit von der konkreten Schnittstelle ausimplementiert. Wir wollen auf spezialisierte Materialversorgerklassen an dieser Stelle nicht weiter eingehen, sondern greifen diese Thematik in Zusammenhang mit der Anbindung einer objektorientierten Datenbank (Kapitel 5) wieder auf.

Obwohl die Anzahl der in eine Werkzeugumgebung eingebetteten Versorgerautomaten prinzipiell beliebig sein kann, gehen wir der Einfachheit halber zunächst davon aus, daß nur *ein* solcher Automat existiert. Die Erprobung des hier vorgestellten Materialversorgungskonzeptes wird aus unserer Sicht dadurch nicht eingeschränkt.

Der `Materialverwalter` bedient sich einer oder mehrerer `Magazine`, um die von ihm verwalteten Materialexemplare effizient zu verwahren. Sie können durch Bereitstellung von Iteratoren nach außen verfügbar gemacht, durch neue Versionen überschrieben, aus dem `Magazin` gelöscht oder eingefügt werden. Wir benötigen `Magazine` an mehreren Stellen des Versorgungsmodells. Auf das Magazin des `Materialverwalters` beziehen wir uns daher künftig als `Verwaltermagazin`.

Zu den Minimalanforderungen an die Funktionalität einer Materialverwaltung zählt unserem Verständnis nach die Unterstützung beliebiger Materialtypen. Insbesondere muß es möglich sein, *neue* Materialien ohne Auswirkungen auf die Schnittstelle des `Materialverwalters` in die Werkzeugumgebung einzubinden. Im Rahmen der objektorientierten Modellierung nach der *Werkzeug-Material* Metapher läßt sich diese Forderung mit Hilfe eines abstrakten Aspekts `Verwaltbar` umsetzen, dessen Schnittstelle der `Materialverwalter` benutzt, um auf einzelne Materialexemplare zuzugreifen.

Der `Materialverwalter` unterstützt eine rudimentäre "Spezifikationssprache", mittels derer Teilmengen der im Magazin enthaltenen Materialien beschrieben und aufgefunden werden können. Eine `Materialspezifikation` kann etwa durch Bildung einfacher Prädikate über Materialnamen oder Materialtypen formuliert werden - der Komplexität solcher Prädikate sind langfristig aber keine Grenzen gesetzt. Zur Auswertung solcher Spezifikationen implementiert der `Materialverwalter` ein abstraktes `Interpreter` - Protokoll, das konzeptionell aus einer Methode besteht, die auf Basis abstrakter Materialbeschreibungen konkrete Materialien liefert. Als relevante Eigenschaften

[5] Selbstverständlich können auch Werkzeuge neue Materialien erzeugen und in der Umgebung verfügbar machen.

verwaltbarer Materialien verstehen wir dementsprechend ihre *Identifizierbarkeit*. Wir gehen davon aus, daß Materialexemplare eindeutig über einen benutzerdefinierten Namen identifizierbar sind. Diese Information reicht bereits, um eine rudimentäre Materialbeschreibung zu realisieren, mittels derer Materialien aus Werkzeugsicht wiederaufgefunden werden können. Wenn wir außerdem die Möglichkeit anbieten wollen, Materialien über Typen zu *klassifizieren*, muß das von Verwaltbar abstrakt spezifizierte Protokoll zusätzlich über eine Methode verfügen, die den Materialtyp eines Exemplars liefert.

3.2. Das Zusammenspiel zwischen Materialverwalter und Werkzeugen

Den Gegenstand dieses Kapitels bildet die Betrachtung von Werkzeugen, die im Rahmen des vorgestellten Versorgungsmodells Materialien aus ihrem Kontext, d.h. im Zusammenspiel mit dem Materialverwalter beziehen möchten. Gleich zu Beginn sei darauf hingewiesen, daß wir den Materialverwalter als einen Automaten verstehen, der im Unterschied zu Werkzeugen keine Aktionen auf Anforderung eines Benutzers ausführt. Als im Hintergrund aktive Komponente stellt der Materialverwalter den Umgebungswerkzeugen vielmehr Dienste auf der Basis fester Algorithmen zur Verfügung.

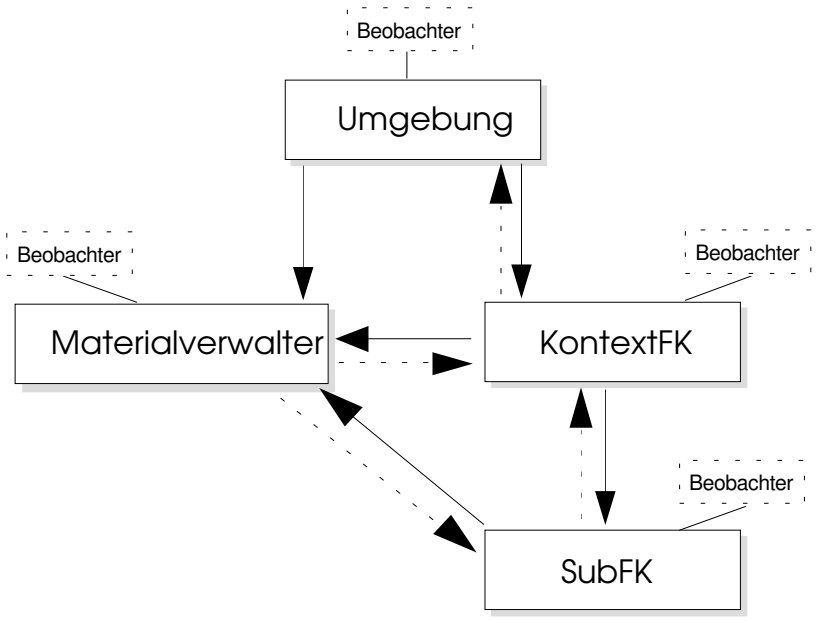


Abb.2: Das Zusammenspiel eines komplexen Werkzeuges mit dem Materialverwalter.

Wir modellieren die Interaktion zwischen diesen Werkzeugen und dem Material-verwalter als unmittelbare *Benutzt*-Beziehung und gehen davon aus, daß die Umgebung ihren Materialverwalter nach seiner Erzeugung unter der vollen Schnittstelle bei den "interessierten" Werkzeugen bekanntmacht. Diese Annahme steht in Einklang mit unserem (softwaretechnischen) Verständnis von der Umgebung, die als einbettende Komponente sowohl den Materialverwalter als auch die Werkzeuge[6] - zumindest die der oberen Hierarchieebene - kennt. Wir können den Materialverwalter über eine prinzipiell

[6] Wir differenzieren in diesem Abschnitt sprachlich nicht zwischen der Funktions- und der Interaktionskomponente von Werkzeugen. Die beschriebene Interaktion mit dem Materialverwalter wird bei der technischen Umsetzung selbstverständlich in der Werkzeug-FK realisiert.

beliebige Hierarchietiefe auch Subwerkzeugen verfügbar machen, indem wir die Aufgabe des Bekanntmachens an das jeweilige Kontextwerkzeug delegieren.

Der Umgang mit dem `Materialverwalter` ist aus der Sicht eines Werkzeuges dadurch gekennzeichnet, daß es Materialien anfordern, löschen oder neu erzeugte bzw. modifizierte Materialien in die Werkzeugumgebung übernehmen kann. Mit dem *Einfügen* neuer Materialien als Anknüpfungspunkt läßt sich anhand eines Beispiels motivieren, warum wir die Interaktion zwischen Werkzeug und `Materialverwalter` insbesondere auch als `Beobachter`-Beziehung verstehen. Vorstellbar wäre etwa ein Editor, mit dem neue Dokumente erstellt werden können, die nach ihrer Bearbeitung und Übergabe an den `Materialverwalter` allgemein in der Umgebung zur Verfügung stehen. Dem `Materialverwalter` sollte in diesem Fall die Möglichkeit gegeben sein, einen bereits gestarteten Dokumentenbrowser von der Existenz des neuen Dokuments zu *benachrichtigen*. Das Einfügen neuer Materialien ist nur *ein* möglicher Vorgang, der dem `Materialverwalter` Anlaß zur Benachrichtigung der ihn beobachtenden Werkzeuge gibt und auf den wir uns keinesfalls beschränken wollen. Als Minimalanforderung sollten zumindest noch die Änderung und das Entfernen eines bereits vorhandenen Materials in den Katalog möglicher Ereignisse aufgenommen werden. Abbildung 2 stellt die geschilderten Zusammenhänge der einzelnen Komponenten noch einmal grafisch dar.

Werkzeuge formulieren Anforderungen an den `Materialverwalter` in Form von Prädikaten über Materialnamen und -typen (vgl. Kapitel 3.1.), die gegen das `Verwaltermagazin` gehalten und ausgewertet werden. Von den Materialien, auf die das Prädikat zutrifft und die ihm selbst per Iterator zur Verfügung gestellt werden, legt der `Materialverwalter` jeweils eine Kopie an. Wir scheinen hier ganz selbstverständlich davon auszugehen, daß der Verwalter bereits alle angeforderten Materialien im Zugriff hat. Tatsächlich wird es im Regelfall so sein, daß der Verwalter die Materialien selbst erst bei einem Versorgerautomaten anfordern muß. Wir können den Punkt jedoch an dieser Stelle vernachlässigen, da er aus Sicht eines Werkzeuges unerheblich ist. Hieran läßt sich insbesondere verdeutlichen, daß Werkzeuge transiente und persistente Materialien im Rahmen des vorgestellten Versorgungskonzepts gleichermaßen handhaben können. Die Materialkopien werden dem anfordernden Werkzeug innerhalb eines eigenen, lokalen `Werkzeugmagazins` verfügbar gemacht, auf das uneingeschränkt lesend und schreibend zugegriffen werden kann. Änderungen bleiben jedoch, da sie nur an Material*kopien* vorgenommen werden, zunächst lokal. Sie werden erst durch einen expliziten Aufruf des `Materialverwalters` dauerhaft in die Umgebung übernommen. Auch die von einem Werkzeug neu erzeugten Materialien müssen dem `Materialverwalter` ausdrücklich zur weiteren Verwaltung und Bereitstellung an andere Werkzeuge übergeben werden.

3.3. Über den Umgang mit gemeinsamen Materialien: Kopien und Originale als Bilder bei einer exklusiven Zugriffskontrolle

In diesem Abschnitt untersuchen wir, welche zusätzlichen Anforderungen an die Funktionalität des `Materialverwalters` erwachsen, wenn Materialien innerhalb der Umgebung von mehreren Werkzeugen *gleichzeitig* benutzt werden. Der konkurrierende Zugriff auf gemeinsame Materialien wirft zwangsläufig Koordinationsprobleme auf, für die wir im folgenden eine mögliche Lösung vorstellen. Sie legt eine fachlich motivierte Sichtweise vom gemeinsamen Umgang mit Materialien zugrunde.

Aus unserem täglichen Leben sind wir einen bestimmten Umgang mit Arbeitsgegenständen gewohnt, die mehreren Personen gemeinsam zugänglich sind. So gehen wir beispielsweise bei einem Dokument davon aus, daß jederzeit genau ein Original, aber beliebig viele Kopien

existieren. Das Original kann nur einer Person zur Zeit vorliegen und auch nur von dieser exklusiv geändert werden. Andere Personen können sich allenfalls Kopien des Originaldokuments anfertigen, gleichgültig, ob sie das Dokument nur lesen oder auch verändern möchten. Änderungen einer Dokumentkopie wirken sich nie auf ihr Original aus. Wenn wir als Inhaber einer Dokumentkopie auch noch das Original selbst bearbeiten möchten, erwarten wir, von seiner Wiederverfügbarkeit in Kenntnis gesetzt zu werden.

Wir können das Bild von Originalen und Kopien verwenden, um bei der Verwaltung gemeinsamer Materialien innerhalb von Werkzeugumgebungen eine exklusive Zugriffskontrolle zu realisieren, die einen *kooperativen* Arbeitsstil nicht behindert.

Aus Werkzeugsicht wird der Umgang mit dem `Materialverwalter` durch einen *Checkin/Checkout*-Mechanismus erweitert. Materialien, die nicht explizit *ausgecheckt* worden sind, haben den Status einer Kopie[7] und können damit im Unterschied zu Originalen nur gelesen werden. Lokales Schreiben ist natürlich immer möglich, Änderungen können jedoch nicht in die Umgebung übernommen werden. Solange ein Werkzeug auf einem Materialoriginal arbeitet, können anderen Werkzeugen nur Kopien zur Verfügung gestellt werden. Erst wenn ein Werkzeug sein Materialoriginal explizit, also im Sinne eines *Checkin*, freigibt, ist es prinzipiell wieder für andere Werkzeuge der Umgebung verfügbar. Die Freigabe eines Materialoriginals nehmen wir daher auch in den Katalog möglicher Ereignisse auf, die ein Materialverwalter seinen beobachtenden Werkzeugen melden kann.

Werkzeuge müssen sich beim Reservieren eines Originals bis zu dessen erneuter Freigabe als `Besitzer` bei dem `Materialverwalter` bekanntmachen. Dies gilt auch für alle anderen Operationen des `Materialverwalters`, insbesondere für jene, mit denen Material des Verwalters gelöscht oder mit einer neuen Version überschrieben werden kann. Die Zulässigkeit solcher Operationen ist an die Identität von Materialbesitzer und anfragender Komponente gebunden und kann daher leicht überprüft werden.

[7] Der Begriff *Kopie* mag an dieser Stelle etwas verwirrend erscheinen, insofern wir im letzten Abschnitt bereits von Magazinkopien gesprochen haben. *Kopie* ist in diesem Abschnitt stets fachlich, also als Gegensatz zu Original zu verstehen.

4. Der Übergang zu einer verteilten Werkzeugumgebung

In diesem Kapitel soll ein Eindruck dessen vermittelt werden, was der Schritt von Ein- zu Mehrarbeitsplatz-Umgebungen in Hinblick auf die Materialversorgung bedeutet. Wir werden zeigen, inwieweit die im vorangegangenen Abschnitt vorgestellten Konzepte zur Materialversorgung *innerhalb* von Werkzeugumgebungen auch bei *verteilten* Umgebungen greifen und wo sie durch neue Konzepte ergänzt oder ersetzt werden müssen.

4.1. Von der Notwendigkeit eines 'zentralen' Materialkoordinators

Daß Materialien nicht nur innerhalb von Werkzeugumgebungen zentral verwaltet werden, sondern bei einer Mehrarbeitsplatzumgebung nun zusätzlich auch noch der Kontrolle eines zentralen Materialkoordinators unterliegen sollen, ist nicht selbstverständlich und bedarf der Motivation. Schließlich werden Materialien im Regelfall bei externen Materiallieferanten beschafft, und so wäre es nur naheliegend anzunehmen, daß Verwaltung und Zugriffskontrolle von entsprechenden Datenbankservern wahrgenommen werden.

Diese Sichtweise scheint uns in zweierlei Hinsicht an Grenzen zu stoßen. Zum einen bleibt ungeklärt, wie Materialien kontrolliert zwischen *Umgebungen* ausgetauscht werden können. Zum anderen stehen wir hier vor Koordinationsproblemen, die gleichermaßen innerhalb der einzelnen Werkzeugumgebung (vgl. Kapitel 3.3.) auftreten und die wir durch eine möglichst durchgängige Übertragung der Konzepte bewältigen möchten. Weder die gängigen Zugriffskontrollmechanismen relationaler Datenbanksysteme, noch die innovativen Lockingmechanismen objektorientierter Datenbanksysteme erlauben jedoch eine fachlich motivierte Zugriffsmodellierung nach dem Bild von Originalen und Kopien. Auch Benachrichtigungsmechanismen werden bei weitem nicht von allen Datenbankherstellern angeboten oder sind nur unter erheblichem Aufwand oder gar nicht in das von uns verwendete Beobachter-Modell einzubinden[8] .

4.2. Das Zusammenspiel zwischen Werkzeugumgebung und zentralem Materialkoordinator

Um Mißverständnissen vorzeubeugen und eine schnellere Einordnung des zentralen `Materialkoordinators` zu ermöglichen, möchten wir gleich zu Beginn dieses Abschnitts auf den wesentlichen Unterschied zu den lokalen `Materialverwaltern` der einzelnen Umgebungen hinweisen.

Von `Materialverwaltern` wissen wir, daß sie für die Beschaffung, Verwahrung und Bereitstellung von Materialien innerhalb einer Werkzeugumgebung zuständig sind, Werkzeuge von Änderungen am Materialbestand der Umgebung benachrichtigen und darüberhinaus den konkurrierenden Zugriff von *Werkzeugen* auf gemeinsame Materialien regulieren. Der `Materialkoordinator` hingegen beschafft weder Materialien, noch

[8] Objektorientierte Datenbanksysteme arbeiten wie ihre relationalen Vorgänger überwiegend transaktionsorientiert und bieten darüberhinaus - zumindest standardmäßig - noch keine Möglichkeit, um einzelne Objekte im Sinne eines *Ckeckout* zum Lesen oder Schreiben zu sperren. Der von manchen Datenbankherstellern vielgepriesene *Change-Update*-Mechanismus besteht oft nur darin, daß geänderte Objekte nach expliziter Anfrage zur Verfügung gestellt werden können. Einige Datenbankhersteller bieten allerdings auch die Möglichkeit zur Festlegung von *Callback*-Funktionen (vgl. [Bar93]).

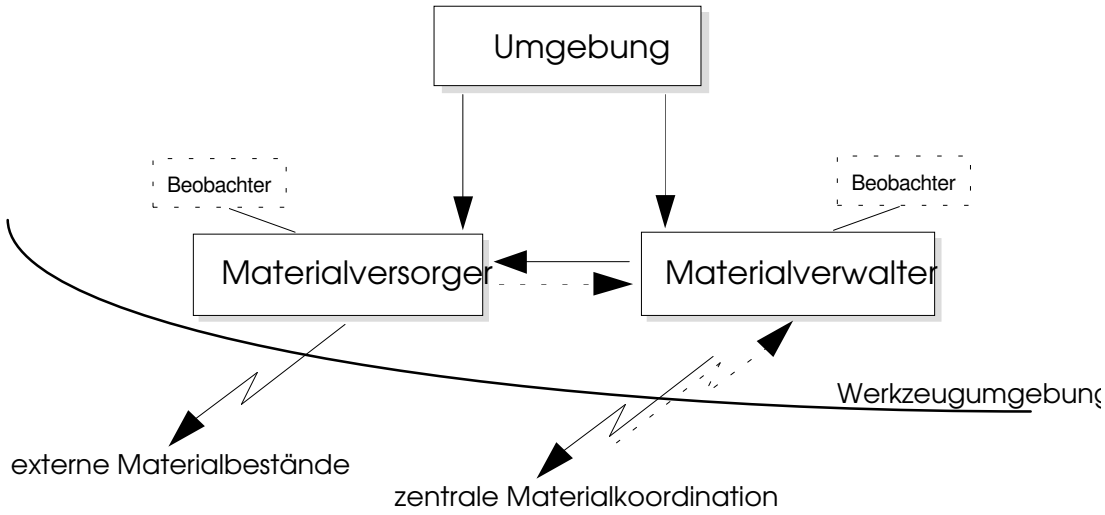


Abb. 3: Die Beziehungen zwischen den Komponenten `Umgebung`, `Materialversorger` und `Materialverwalter` beim Übergang zu einer verteilten Werkzeugumgebung.

verwahrt er sie oder macht sie den Werkzeugumgebungen verfügbar. Seine Aufgabe beschränkt sich darauf, konkurrierende Zugriffe von *Werkzeugumgebungen* auf gemeinsame Materialien zu koordinieren und diese über Änderungen am gemeinsamen Materialbestand zu informieren. Der `Materialkoordinator` soll existierende Materiallieferanten - etwa im Sinne von relationalen oder objektorientierten Datenbanken[9] - nicht *ersetzen* oder *verbergen*, sondern Werkzeugumgebungen nur die Möglichkeit geben, Zugriffe auf externe Materialien in den Kontext einer fachlich motivierten Zugriffsmodellierung nach dem Bild von Originalen und Kopien einzubetten, ohne dabei einen kooperativen Arbeitsstil zu verhindern. Die eigentliche Aktivierung, also das Laden von externen Materialien in den Addressraum der einzelnen Umgebung, und später ihre Passivierung erfolgt aus den einzelnen Umgebungen heraus über eine *direkte* Anfrage bei den jeweiligen Materiallieferanten. Im Rahmen des vorgestellten Versorgungsmodells sind hierfür die bereits erwähnten `Materialversorger`-Automaten zuständig (vgl. Kapitel 3.1.).

Aus der Sicht von Werkzeugumgebungen ist der Umgang mit dem `Materialkoordinator` dadurch charakterisiert, daß sie Materialien vor der eigentlichen Rekonstruktion aus einer Datenbank als Originale *auschecken* und diese nach ihrer Speicherung in der Datenbank wieder *einchecken* müssen - sofern sie natürlich nicht nur lesenden Zugriff darauf benötigen. Wir interpretieren die einzelnen Werkzeugumgebungen überdies als `Beobachter` des `Materialkoordinators`, denen Änderungen am gemeinsamen Materialienbestand gemeldet werden. Zum Repertoire typischer Ereignisse zählen wir das Löschen, Überschreiben und Einfügen eines Materials, sowie - im Rahmen der vorgestellten Zugriffsmodellierung - seine Freigabe als Original. In dieser Beziehung können wir also die Konzepte der Ein-Arbeitsplatz- Werkzeugumgebung (vgl. Kapitel 3.3.) unmittelbar übernehmen.

Interessant ist jedoch die Frage, auf welcher Basis der `Materialkoordinator` überhaupt entscheidet, *welches* Ereignis *wann* zu melden ist. Im Unterschied zu den lokalen `Materialverwaltern`, die aus Werkzeugsicht gewissermaßen Materiallieferanten und -

[9] Wir verstehen selbstverständlich nicht nur Datenbanken, sondern auch Werkzeugumgebungen oder andere Anwendungen, die Materialien zur Verfügung stellen, als Materiallieferanten. Auf einer konzeptuellen Ebene soll hier ja gerade nicht differenziert werden. Die Argumentation, die wir im folgenden in Bezug auf Datenbanken führen, läßt sich daher bruchlos auf andere denkbare Materiallieferanten übertragen.

koordinator in einem sind, trennen wir ja auf der umgebungsübergreifenden Ebene zwischen einem `Materialkoordinator` und den davon unabhängigen Datenversorgern. In diesem Fall müssen dem `Materialkoordinator` also erst einmal die tatsächlich ablaufenden "Materialbewegungen" zwischen Werkzeugumgebungen und Datenversorgern - das Löschen, Einfügen oder Überschreiben von Materialien - bekanntgemacht werden. Eine wesentliche Aufgabe des `Materialkoordinators` liegt daher neben der *Autorisierung* der Zugriffe in der *Protokollierung* der Materialbewegungen, die erst die Basis für die Koordination und sinnvolle Benachrichtigung der einzelnen Werkzeugumgebungen bildet. Wir schlagen daher vor, den Umgang einer Werkzeugumgebung mit dem `Materialkoordinator` so zu erweitern, daß sie, nachdem Materialien über ihren Versorgerautomaten bei einem externen Materiallieferanten gelöscht, eingefügt oder mit einer neuen Version überschrieben worden sind, die korrespondierende Operation des `Materialkoordinators` ruft, innerhalb derer der Zugriff protokolliert und gegebenenfalls durch Meldung des entsprechenden Ereignisses an alle beobachtenden Werkzeugumgebungen propagiert wird.

In Bezug auf die Interaktion mit dem `Materialkoordinator` haben wir bisher recht allgemein von "der Werkzeugumgebung" gesprochen. Bei einer differenzierteren Betrachtungsweise kehrt der `Materialverwalter` zurück ins Spiel. Er *benutzt* den zentralen `Materialkoordinator`, um seine Kommunikation mit den `Materialversorgern` - und daher indirekt auch mit den tatsächlichen Materiallieferanten - in eine fachlich motivierte Zugriffskontrolle nach dem Bild von Originalen und Kopien einzubinden. In umgekehrter Richtung ist der `Materialverwalter` Empfänger von Ereignissen des `Materialkoordinators`, die er in geeigneter Weise, etwa durch Aktualisieren oder Entfernen eines Materials aus seinem Magazin, abarbeitet und schließlich an die Umgebungswerkzeuge weiterleitet.

Wir weisen darauf hin, daß aus der Sicht einzelner Werkzeuge völlig unerheblich und nicht feststellbar ist, ob Materialmanipulationen außerhalb der Werkzeugumgebung oder innerhalb der Umgebung von anderen Werkzeugen vorgenommen worden sind. Abb. 3 zeigt die in Kapitel 3.1. bereits einmal dargestellten Beziehungen zwischen den Entwurfskomponenten `Umgebung`, `Materialversorger` und `Materialverwalter` unter Berücksichtigung der soeben geschilderten Zusammenhänge.

5. Die Anbindung einer objektorientierten Datenbank am Beispiel von ONTOS

Im folgenden wollen wir veranschaulichen, wie Datenbanken im Rahmen des vorgestellten Materialversorgungsmodells an eine verteilte Werkzeugumgebung angebunden werden können. Dabei bedienen wir uns der bisher vorgestellten Architektur als Rahmenwerk. Der Grundgedanke ist, solche Anbindungen durch Spezialisierung des `Materialversorgers` in datenbankspezifische Automaten zu realisieren. Bevor wir diesen Gedanken am Beispiel des objektorientierten Datenbanksystems ONTOS präzisieren, stellen wir einige Grundkonzepte des objektorientierten Datenmodells vor und geben einen kurzen Einblick in das verfügbare System ONTOS.

5.1. Grundkonzepte und -begriffe des objektorientierten Datenmodells

Die Standardisierung im Bereich objektorientierter Datenbanksysteme wird maßgeblich von der *Object Database Management Group* (ODMG) der OMG (*Object Management Group*) vorangetrieben, die erst kürzlich ihre neuesten Standardisierungsvorschläge als Papier vorgelegt hat[10]. Die Inhalte des ODMG-93 Standards - Definition eines Objektmodells, Schnittstellen zu objektorientierten Sprachen, Integration mit OMG-Modellen wie beispielsweise dem *Object Request Broker* (vgl. Kapitel 6.2.2.), und vieles mehr - ragen weit über das hinaus, was für das Verständnis dieser Arbeit notwendig ist. Wir beschränken uns daher im folgenden auf die Darlegung einiger Grundkonzepte und -begriffe des objektorientierten Datenmodells, die wir, soweit es uns sinnvoll erscheint, gegen die relationale Sicht abgrenzen. Eine detaillierte Unterscheidung von objektorientiertem und relationalem Datenmodell findet sich z.B. in [Cat91].

5.1.1. Persistente Objekte

Die wesentliche Erweiterung der objektorientierten Programmierung um ein objektorientiertes Datenmodell besteht in der Einführung von *persistenten* Objekten. Persistente Objekte können - im Gegensatz zu herkömmlichen transienten Objekten - den sie erzeugenden Prozeß "überleben" und jederzeit als normale Programmvariable in den Addressraum eines neuen Prozesses reaktiviert werden.

5.1.2. Abstrakte Datentypen, Vererbung

Im Unterschied zum relationalen Modell erlaubt das objektorientierte Datenmodell die Realisierung benutzerdefinierter abstrakter Datentypen. Die Komplexität persistenter Objektattribute unterliegt prinzipiell keinerlei Beschränkungen, und Methoden, die den Umgang mit einem Objekt formalisieren, können zusammen mit diesem gespeichert werden. Diese Eigenschaften sind unverkennbar dem Klassenkonzept objektorientierter Programmiersprachen entlehnt, und so lag es nahe, im objektorientierten Datenmodell auch das Konzept der Vererbung zu unterstützen.

5.1.3. Objektidentität

[10] vgl. Rick Cattell (Hrsg.): *The ODMG-93 Standard*. Entwurf, Version 0.9, Revision 7/1/1993.

Datensätze können im relationalen Datenmodell nur über die Werte ihrer (Primär-) *Schlüssel* voneinander unterschieden, d.h. identifiziert werden. Im objektorientierten Datenmodell werden Objekte dagegen über eindeutige, vom System generierte Bezeichner (Object Identifier, kurz OIDs) identifiziert. Dies erleichtert die Referenzierung der Objekte untereinander und entkoppelt, im Unterschied zum relationalen Modell, die Identität eines Objektes von seinen Werten. OIDs sind i.d.R. nicht durch die Anwendung manipulierbar.

5.1.4. Referenzen und Referentielle Integrität

Das objektorientierte Datenmodell hebt die Trennung zwischen Attributen und Beziehungen bei der Datenmodellierung vollständig auf. Der Bezug eines Objektes auf ein anderes wird als Referenz bezeichnet und drückt sich technisch in Form von komplexen Attributwerten aus. Der Attributwert eines Objektes kann m.a.W. selbst wieder ein Objekt oder eine Sammlung von Objekten sein. Fälschlicherweise werden Referenzen oft mit Zeigern gleichgesetzt. Tatsächlich verbirgt sich hinter ihnen ein weitaus mächtigeres Konzept einer höheren Abstraktionsebene: Referenzen sind im Unterschied zu Zeigern keine Speicheradressen, sondern Verweise auf OIDs. Das Konzept der Referenz erleichtert durch die Definition sogenannter *inverser Referenzen* die Aufrechterhaltung der *referentiellen Integrität*. Mit diesem Terminus wird die permanente Sicherstellung der Gültigkeit aller Referenzen durch die jeweilige Datenbank bezeichnet.

5.1.5. Transaktionen

Das aus dem relationalen Datenmodell bereits wohlbekannte Transaktionskonzept (vgl. beispielsweise [Gra81]) kommt gleichermaßen in objektorientierten Modellen zum Einsatz. Objektorientierte Anwendungssysteme stellen i.d.R. allerdings ganz andere Anforderungen an das Transaktionsverhalten, da die Bearbeitung von Objekten hier nicht auf der Ebene von einzelnen Datensätzen im Millisekundenbereich stattfindet, sondern häufig über Stunden oder gar Tage erfolgt (vgl. [Cat91]). Dieser Sachverhalt wird durch den Begriff des *Langzeit-Checkout* geprägt. Die meisten objektorientierten Datenbanksysteme sehen daher die Möglichkeit vor, Veränderungen mit Hilfe sogenannter Checkpoints zwischenzuspeichern, um allzugroßen Datenverlust durch den Abbruch einer langen Transaktion zu vermeiden. Eng verzahnt mit der Problematik langer Transaktionen ist die Forderung nach innovativen Zugriffskontrollmechanismen. In diesem Kontext werden beispielsweise *optimistische* Zugriffsprotokolle diskutiert, die prinzipiell einen parallelen lesenden und schreibenden Zugriff auf Daten ermöglichen und erst beim Abschluß der Transaktion auf Serialisierbarkeit der Zugriffe prüfen. Es sei angemerkt, daß sich das Transaktionskonzept für die von uns gewählte Zugriffskontrolle nach dem Bild von Original und Kopie als nicht brauchbar erwiesen hat, da Transaktionen Zugriffskontrolle *implizit* auf der Basis von technischen Gesichtspunkten (Serialisierbarkeit) ausüben, während wir *explizite*, fachlich motivierte Mechanismen favorisieren.

5.2. Ein verfügbares System: das Beispiel ONTOS

ONTOS ist eine kommerziell verfügbare objektorientierte Datenbankprogrammiersprache[11] mit *Client-Server* Architektur. Eine auf der Client-Seite laufende Anwendung greift nicht direkt auf die Datenbank zu, sondern stellt ihre Anforderung an einen Datenbankserver. Da eine Datenbank physikalisch über mehrere Hosts verteilt sein kann, übernimmt es der Datenbankserver, bei Bedarf mit weiteren Servern zu kommunizieren. Dieser Vorgang bleibt vor der Anwendung verborgen.

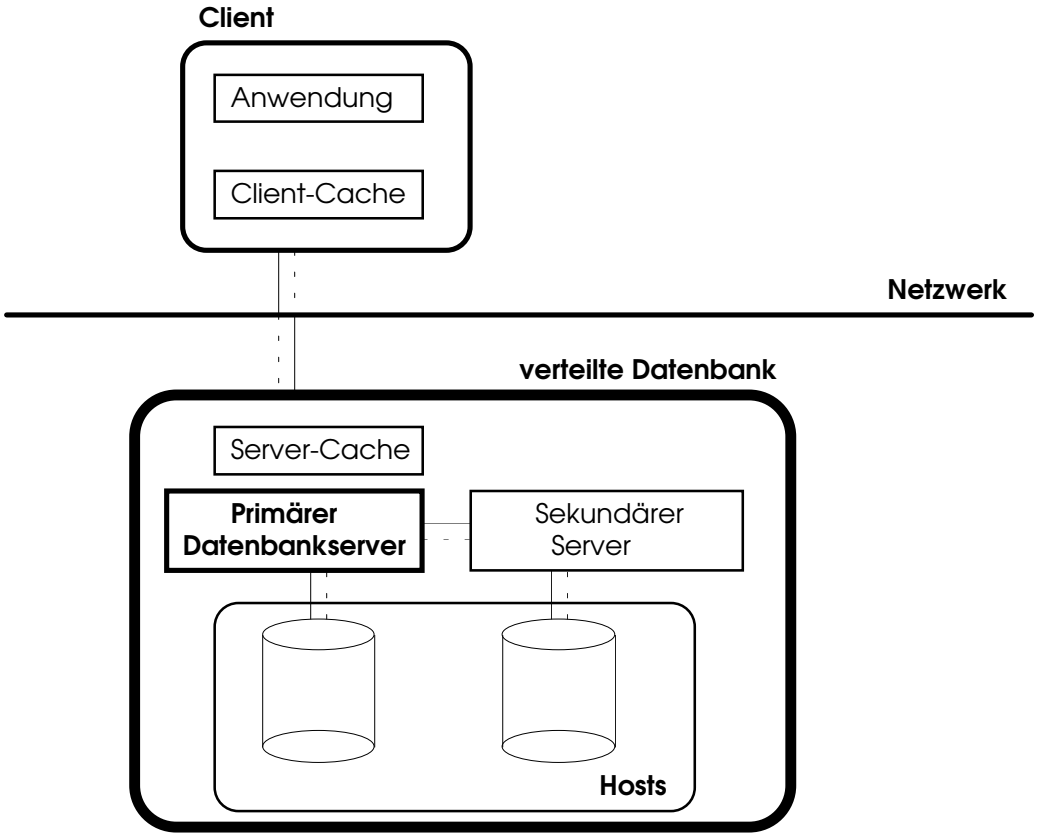


Abb. 4: Die Architektur von ONTOS (nach [ONT92])

Ein beim Datenbankserver angefordertes Objekt wird zunächst in den Server-Cache gelesen, um dann von dort in den Client-Cache *aktiviert* zu werden. Dabei wird seine Objektidentität in eine Speicheradresse des virtuellen Adreßraumes der Anwendung überführt. Manipulationen auf dem Objekt verändern zunächst ausschließlich den Client-Cache. Erst durch explizite *Deaktivierung* wird das geänderte Objekt in den Server-Cache zurückgeschrieben. Der Datenbankserver prüft auf konfliktfreien Zugriff und sorgt gegebenenfalls für das Zurückschreiben des Objektes in die Datenbank.

Als Programmierschnittstelle bietet ONTOS die Sprache C++ an. Eine Anwendung erwirbt sich ihre Datenbankfunktionalität durch das Einbinden einer Klassenbibliothek, die u.a. die Basisklasse aller persistenten Objekte, `Object`, zur Verfügung stellt. Nur solche Objekte einer Anwendung, die Exemplare einer von `Object` abgeleiteten Klasse sind, können in ONTOS-Datenbanken gespeichert werden. Eine Anwendung greift auf ein persistentes

[11] Wir sprechen hier von einer Datenbank*programmiersprache* im Unterschied zu einem Datenbank*system*, da der von ONTOS realisierte Datenbankkern auf C++-Basis einige klassische Aufgaben von Datenbanksystemen in die Verantwortlichkeit des Programmierers überträgt (vgl. [Heu92], S.485)

Objekt entweder über seinen (benutzerdefinierten) Namen oder über das Verfolgen einer Referenz zu. Eine ungültige Referenz auf ein gelöschtes Objekt wird in ONTOS durch einen Ausnahmebehandlung-Mechanismus abgefangen, der erst zum Zeitpunkt der Referenzverfolgung ausgelöst wird. Wir möchten an dieser Stelle auf die *Passivität* des Verfahrens hinweisen. Integritätsverletzungen werden zwar erkannt und sind auch behandelbar, können jedoch nicht vermieden werden. Obendrein obliegt es dem Programmierer, die Ausnahme in geeigneter Weise zu behandeln.

5.3. Ein Datenbankautomat als spezialisierter Materialversorger

Wir haben bereits mehrfach angedeutet, daß die Anbindung von Datenbanken durch Spezialisierung des `Materialversorgers` in datenbankspezifische Automaten realisiert werden kann. Wir werden diesen Grundgedanken im folgenden detaillierter ausführen.

Materialversorgerklassen kapseln bekanntlich das Wissen um konkrete Schnittstellen zu externen Materiallieferanten und gewährleisten damit unter anderem auch ein hohes Maß an Unabhängigkeit von konkreten Datenbanksystemen. Sie implementieren in Abhängigkeit von ihrer je spezifischen Schnittstelle Routinen, mit denen externe Materialien gelöscht oder innerhalb einer Werkzeugumgebung rekonstruiert werden können, und die in umgekehrter Richtung die (Rück-) Überführung eines Materials der Umgebung zu den externen Materiallieferanten ermöglichen.

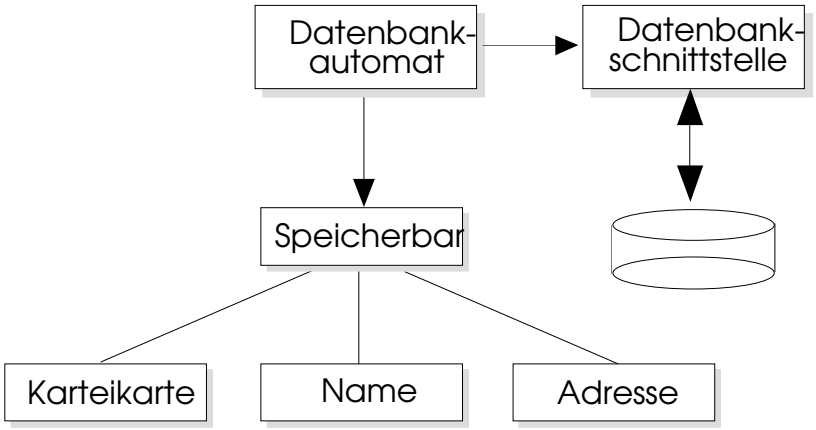


Abb. 5: Das Zusammenspiel zwischen Datenbankautomat und Materialien (nach [Kil93], S. 77)

Mit Kilberth et al. (vgl. [Kil93], S. 76f.] verstehen wir Datenbanken als Automaten, dessen Zweck in der *Speicherung* und *Wiedergewinnung* von Materialien besteht. Im Unterschied dazu sind etwa Materialversorgerautomaten denkbar, deren Anliegen der *Austausch* von Materialien zwischen Umgebungen ist.

Materialien werden dem Datenbankautomaten erwartungsgemäß über einen Aspekt verfügbar gemacht: nur Exemplare einer von `Speicherbar` erbenden Materialklasse können von dem Automaten aus der Datenbank in die Umgebung geladen bzw. aus der Umgebung in die Datenbank überführt werden.

Die Modellierung als *Automat* ist fachlich motiviert und zeigt, daß die Datenbank im Unterschied zu den *Werkzeugen* der Umgebung nicht als ein für den Benutzer sichtbares Arbeitsmittel verstanden wird. So wie Benutzer beim Umgang mit Werkzeugen ohne expliziten Befehl von der Darstellung des je spezifischen Leistungsumfanges und der

bearbeiteten Materialien durch eine *Werkzeugoberfläche* ausgehen dürfen, soll auch die Persistenz von Materialien ohne Einsatz eines vom Benutzer zu bedienenden Speicherwerkzeuges gewährleistet sein. Da die Speicherbarkeit von Materialien im Unterschied zu ihrer Darstellbarkeit jedoch eine von der konkreten Werkzeugausprägung unabhängige *Materialeigenschaft* ist, werden Rekonstruktion und Speicherung von Materialien durch eine unabhängige Komponente, eben den `Datenbankautomaten`, modelliert und nicht etwa in das einzelne Werkzeug integriert.

Um dem Anspruch des vorgestellten Versorgungskonzeptes gerecht zu werden und die konkrete Herkunft von Materialien vor den Werkzeugen der Umgebung zu verbergen, ist es gerade entscheidend, daß die Routinen des Datenbankautomaten ausschließlich auf Veranlassung des `Materialverwalters` und niemals direkt von einem Werkzeug ausgelöst werden. Jeglicher Umgang eines Werkzeuges mit Materialien, der über einen rein fachlichen Aspekt hinausgeht, erschöpft sich in der Kommunikation mit dem `Materialverwalter`. Nur er kann entscheiden, ob etwa ein neu erzeugtes Material, nachdem es in der Umgebung verfügbar gemacht worden ist, auch in einen angebunden Datenspeicher übernommen werden soll und falls ja, in welchen.

6. Von der fachlichen Modellierung zum technischen Klassenentwurf

Im Fokus dieses Kapitels steht die *technische Umsetzung* des entwickelten Materialversorgungsmodells für verteilte Werkzeugumgebungen.

Im ersten Teil richten wir uns insbesondere an solche Leser, die Werkzeuge und Automaten nach der *Werkzeug-Material* Metapher konstruieren und in das Versorgungsmodell einbinden möchten. Die Aufbereitung der einzelnen Abschnitte zeigt in diesem Sinne deutlich den Charakter einer technischen Dokumentation. Zusätzliche Referenz und vertiefte Orientierungsmöglichkeit bietet Kapitel 7, in dem exemplarisch der Entwurf und die Implementierung eines prototypischen Werkzeuges beschrieben wird, das wir zur Verifikation der Tragfähigkeit des Modells entwickelt und in eine Werkzeugumgebung mit objektorientiertem Datenbankanschluß integriert haben.

Im zweiten Teil dieses Kapitels beschäftigen wir uns mit der softwaretechnischen Lösung von Problemen, die im Zusammenhang mit der *Verteilung* kooperativer Werkzeugumgebungen stehen. Wir runden dieses Thema mit einem Blick auf die gegenwärtige Diskussion um Standards für verteilte Anwendungen im Rahmen des objektorientierten Paradigmas ab.

6.1. Das technische Klassenmodell

Wir übernehmen das Begriffsgerüst des fachlichen Klassenentwurfs nahezu vollständig in das technische Klassenmodell (vgl. [Kil93], S. 20f.). Fachliche Komponenten, die als konkreter Ausgangspunkt für die Konstruktion einer technischen Klasse dienen, machen wir als solche kenntlich. Auf technische Klassen beziehen wir uns durchgängig mit Hilfe englischer Bezeichner. Dies erleichtert zum einen schon auf sprachlicher Ebene die Unterscheidung zwischen fachlichen und technischen Komponenten, und entspricht zum anderen dem informellen Standard bereits vorhandener Klassenbibliotheken des Arbeitsbereiches.

Eine detaillierte Beschreibung *aller* Komponenten des technischen Klassenmodells würde vermutlich den Umfang dieser Arbeit sprengen. Wir haben uns daher auf diejenigen Klassen beschränkt, die von potentiellen Werkzeugentwicklern wiederverwendet oder sinnvoll durch Ableitung spezialisiert werden können. Zu ihnen exsitiert eine entsprechende Schnittstellenspezifikation in der Notation der objektorientierten Programmiersprache C++. Soweit sich die Bedeutung einzelner Parameter oder Rückgabewerte nicht unmittelbar aus dem Zusammenhang erschließen läßt und sie nicht bereits an anderer Stelle erklärt worden sind, fügen wir jeweils eine kurze Beschreibung an.

Um einen möglichst verständlichen Eindruck der *Gesamtarchitektur* zu vermitteln, beschreiben wir darüberhinaus den jeweiligen Kontext einer Klasse.

6.1.1. Die Automatenklasse `tManager`

Als fachliche Entwurfskomponente bildet der `Materialverwalter` den unmittelbaren Ausgangspunkt für die Konstruktion der Automatenklasse `tManager`. Im Sinne von [Gam92] verstehen wir `tManager` als *zentrale* Komponente einer Gruppe von Klassen, die bei der Realisierung des Materialversorgungsmodells kooperieren - zentral deshalb, weil sie

aus der Sicht von Werkzeugklienten, die das Klassenteam[12] bei der Materialbeschaffung zwar als Einheit benutzen, aber nie direkt auf einzelne Mitglieder zugreifen, die Rolle eines Vermittlers übernimmt.

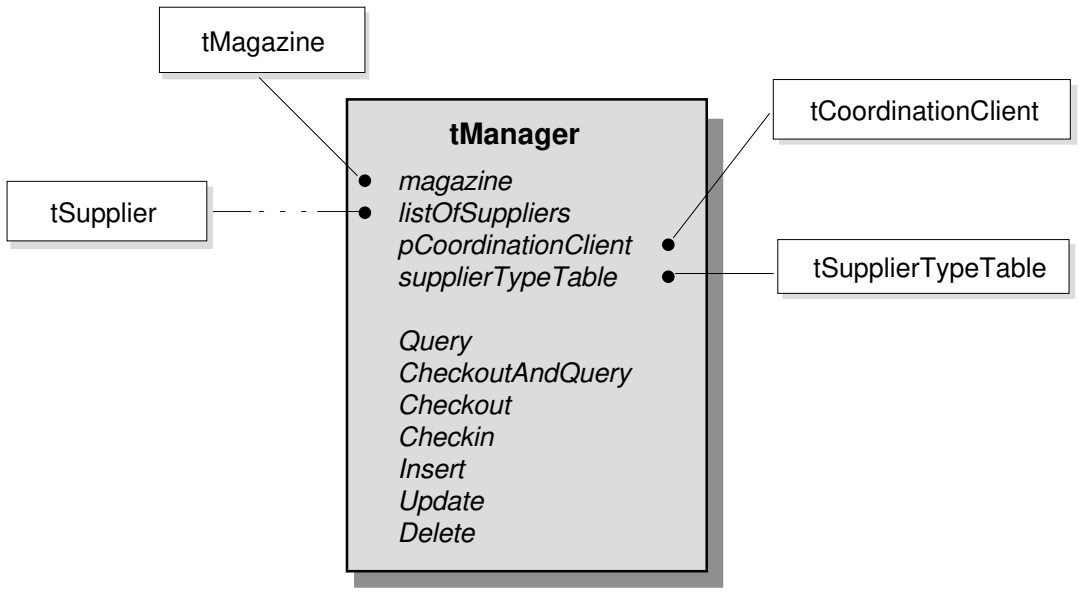


Abb. 6: Technischer Entwurf der Automatenklasse `tManager`.

Die Verantwortlichkeiten, die wir bereits bei der fachlichen Modellierung herausgearbeitet haben, sind gleichmäßig auf die kooperierenden Klassen des Teams verteilt:

• Materialverwalter bedienen sich bekanntlich einer oder mehrerer `Verwaltermagazine`, um die von ihnen verwalteten Materialexemplare effizient zu verwahren (vgl. Kapitel 3.1.). Innerhalb des Klassenteams wird diese Aufgabe von der Klasse `tMagazine` implementiert (*magazine*).

• Der Materialtransport zwischen `tManager`-Objekten und beliebigen externen Datenversorgern, der konzeptionell mit Hilfe von `Materialversorgern` realisiert werden kann (vgl. Kapitel 3.1.), gehört zu den Verantwortlichkeiten der Klasse `tSupplier`. Als abstrakte Klasse bildet sie die Wurzel einer möglichen Klassenfamilie, deren spezialisierte Materialversorgerklassen ein gemeinsames Protokoll für die Kommunikation mit externen Versorgern wie beispielsweise relationalen oder objektorientierten Datenbanken unterstützen. `tManager` ist mit diesen spezialisierten Materialversorgerklassen abstrakt gekoppelt, d.h. Objektkommunikation findet ausschließlich auf der Ebene des abstrakten Protokolls statt (*listOfSuppliers*).

• Damit ein `tManager`-Objekt auf der Ebene dieses abstrakten Protokolls den für ein Materialexemplar je zuständigen Materialversorger ausfindig machen kann, implementiert die Klasse `tSupplierTypeTable` eine eindeutige Abbildung von Materialtypen auf `tSupplier`-Objekte (*pSupplierTypeTable*). Wir legen hier implizit die Annahme zugrunde, daß die Materialien einer Werkzeugumgebung zwar insgesamt von verschiedenen, externen Datenversorgern bezogen werden können, Materialexemplare desselben Typs aber stets nur von einem Versorger verwaltet werden.

---

[12] Detaillierte Informationen über Klassenteams und andere Arten von Klassengruppen können in [Gam92], S. 96ff. nachgelesen werden.

• Die Verwaltung gemeinsamer Materialien nach dem Bild von Originalen und Kopien kann in verteilten Werkzeugumgebungen konzeptionell mittels eines `Materialkoordinators` realisiert werden (vgl. Kapitel 4.2.). Wir werden später noch genauer darauf eingehen, daß Materialkoordination über Prozeßränder hinaus im Sinne einer *Client-Server*-Kommunikation vollzogen wird und der Materialkoordinator deshalb nicht 1:1 in einen technischen Klassenentwurf überführt werden kann (vgl. Kapitel 6.2.). Vielmehr verfügt jede Werkzeugumgebung über ein *Proxyobjekt* vom Typ `tCoordinationClient`, das die Koordinationskomponente des zentralen Serverprozesses auf der Clientseite repräsentiert und einem `tManager`-Objekt im Zuge seiner Initialisierung bekannt gemacht wird (*pCoordinationClient*).

6.1.1.1. Materialien anfordern

```
tDepot* Query(tOwner* anOwner_p, tRequest aRequest, tLock aLock);
```

Kontext

Werkzeug-FKs benutzen die Methode `Query()`, um Materialien mittels einer abstrakten Materialbeschreibung anzufordern. Im Sinne einer kooperativen Zugriffskontrolle (vgl. Kapitel 3.3.) müssen sie dabei explizit machen, ob die betreffenden Materialien als fachliche Originale oder als Kopien bereitgestellt werden sollen.

Parameter

`anOwner_p` bezeichnet die Adresse der anfragenden Werkzeug-FK. Beim Aufruf der Methode wird als aktueller Parameter `this` übergeben:[13]

```
myManager->Query(this, myRequest, cWriteLock);
```

`aRequest` ist ein Prädikat über Materialnamen und -typen. Die Werkzeug-FK spezifiziert über diesen Parameter beispielsweise, daß alle mit dem Buchstaben "A" beginnenden Dokumente für sie bereitgestellt werden sollen.

`aLock` spezifiziert, ob die angeforderten Materialien als Originale (`cWriteLock`) oder als Kopien (`cReadLock`) bereitgestellt werden sollen. Originale können nur dann angefordert werden, wenn die anfragende Komponente sich vorher im Sinne eines *Checkout* als Besitzer der betreffenden Materialien hat registrieren lassen.

Rückgabewert

`Query()` liefert als Rückgabewert ein Behälterobjekt des Typs `tDepot`, in dem der Werkzeug-FK die angeforderten Materialexemplare bereitgestellt werden.

6.1.1.2. Materialien als Originale reservieren

```
tManageableIterator* Checkout(   tOwner* anOwner_p,
                                 tRequest aRequest );
```

[13] Die Adresse einer anfragenden Komponente wird generell benötig um zu prüfen, ob die Durchführung der Operation zulässig ist. Der Materialverwalter darf etwa ein Material nicht als Original für ein Werkzeug bereitstellen, wenn es sich nicht vorher im Sinne eines *Checkout* als Besitzer des betreffenden Materials hat registrieren lassen.

```
tDepot* CheckoutAndQuery(    tOwner* anOwner_p,
                             tRequest aRequest );
```

Kontext

Werkzeug-FKs benutzen die Methode `Checkout()` bzw. `CheckoutAndQuery()`, um sich mittels abstrakter Materialbeschreibung als neue Besitzer bestimmter Materialien registrieren zu lassen. Sie erwerben damit die Berechtigung, die betreffenden Materialien im Sinne eines Originals zu manipulieren.

Rückgabewerte

`Checkout()` liefert als Rückgabewert einen Iterator, mit dem die anfragende Werkzeug-FK *lesend* auf die soeben für sie reservierten Materialien zugreifen kann. Die Verwendung der Methode erscheint in solchen Kontexten sinnvoll, bei denen zwar eine Vielzahl von Materialien als Original reserviert, aber der anfragenden Komponente nicht sofort zur Bearbeitung verfügbar gemacht werden soll. In solchen Fällen ist das eigentlich zu bearbeitende Material gesondert, also über eine explizite `Query`-Anfrage, anzufordern.

Diese Vorgehensweise erscheint uns in solchen Zusammenhängen umständlich, bei denen eine Vielzahl von Materialien nicht nur als Original reserviert, sondern auch unmittelbar verfügbar gemacht werden muß. `CheckoutAndQuery()` liefert der anfragenden Komponente daher als Rückgabewert ein entsprechendes Behälterobjekt.

6.1.1.3. Materialoriginale freigeben

```
tState Checkin(tOwner* anOwner_p, tManageable* aManageable_p);
```

Kontext

Werkzeug-FKs machen einzelne Materialoriginale mit Hilfe der Methode `Checkin()` wieder frei in der Werkzeugumgebung verfügbar.

Parameter

`aManageable_p`	ist eine Referenz auf das betreffende Materialoriginal. Materialien werden der Klasse `tManager` stets unter der allgemeinen Transportschnittstelle `tManageable` bekannt gemacht (vgl. Kapitel 6.1.3.).

Rückgabewert

Die Methode `Checkin()` liefert als Rückgabewert ein Element des Aufzählungtyps `tState`, das Auskunft über den Erfolg (`cSuccess`) bzw. Mißerfolg (`cFail`) der Operation gibt.

6.1.1.4. Materialoriginale löschen, einfügen und ändern

```
tState Delete( tOwner* anOwner_p, tManageable* aManageable_p );
tState Insert( tOwner* anOwner_p, tManageable* aManageable_p );
tState Update( tOwner* anOwner_p, tManageable* aManageable_p );
```

Kontext

Werkzeuge können Materialien in ihren lokalen Behälterobjekten löschen, verändern oder neu erzeugen. Durch explizite Anfrage bei ihrem `tManager`-Objekt lassen sich solche lokalen Änderungen in die Werkzeugumgebung übernehmen. An der Klassenschnittstelle von `tManager` werden daher die Methoden `Insert()`, `Update()` und `Delete()` nach außen verfügbar gemacht.

6.1.2. Die Behälterklasse `tDepot`

Als fachliche Entwurfskomponente bildet das `Werkzeugmagazin` den unmittelbaren Ausgangspunkt für die techischen Konstruktion der Behälterklasse `tDepot`.

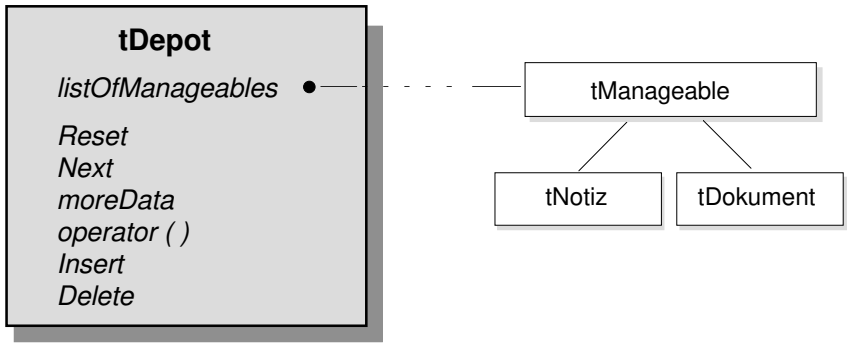


Abb. 7: Technischer Entwurf der Behälterklasse `tDepot`.

`tDepot`-Objekte sind Behälter, die Sammlungen von Materialexemplaren verwalten. Sie werden insbesondere von `tManager`-Objekten bei der Zusammenstellung von Materialsammlungen für Werkzeugklienten benutzt (vgl. Kapitel 6.1.1.1.). Die Klasse `tDepot` unterstützt die Verwaltung heterogener Materialien, d.h. ein `tDepot`-Objekt kann gleichzeitig Materialien unterschiedlicher Typen verwalten, vorausgesetzt, sie erben von der Aspektklasse `tManageable`. Unter dieser abstrakten Schnittstelle werden sie einem `tDepot`-Objekt bekannt gemacht. Die Aspektklasse `tManageable` spielt eine zentrale Rolle innerhalb des Materialversorgungsmodells, so daß wir ihr einen gesonderten Abschnitt widmen (vgl. Kapitel 6.1.3.). Wie für Behälterklassen üblich, unterstützt `tDepot` die Iteration über seine Elemente, ohne daß Klienten auf die interne Datenrepräsentation zugreifen müssen.

6.1.2.1. Über die Materialien des Behälters iterieren

Werkzeugklienten können ausschließlich *vorwärts-iterativ* auf Materialien eines ihnen zur Verfügung gestellten `tDepot`-Objektes zugreifen. Zu diesem Zweck stellt die Klasse `tDepot` an seiner Schnittstelle Methoden zur Verfügung, mit denen

- Werkzeuge an den Anfang des Materialbehälters gehen können

```
void Reset ();
```

- innerhalb des Behälters um eine Position vorrücken können, falls der Behälter noch weitere Materialien enthält

```
void Next ();

Bool moreData ();
```

- Materialien entnehmen können

```
tManageable * operator() ();
```

6.1.2.2. Materialien einfügen und löschen

```
tState Insert( tManageable* aManageable_p );
tState Delete( );
```

Kontext

Werkzeugklienten können auf ein ihnen zur Verfügung gestelltes `tDepot`-Objekt uneingeschränkt lesend und schreibend zugreifen. Typischerweise werden Werkzeuge neue Materialexemplare einfügen oder bereits vorhandene Materialexemplare löschen wollen. An der Klassenschnittstelle sind daher die Methoden `Insert()` und `Delete()` sichtbar. Die Änderung eines bereits existierenden Materials braucht hingegen nicht explizit ausgeführt zu werden, da Werkzeugen bei der Entnahme eines Materials keine Kopie, sondern eine direkte Referenz auf das Material im `tDepot`-Objekt geliefert wird.

6.1.3. Die Aspektklasse `tManageable`

Im Rahmen der fachlichen Modellierung haben wir den Aspekt `Verwaltbar` eingeführt, unter dem ein `Materialverwalter` auf die von ihm bearbeiteten Materialien zugreift (vgl. Kapitel 3.1.). Er dient bei der technischen Umsetzung als unmittelbarer Ausgangspunkt für die Spezifikation der abstrakten Aspektklasse `tManageable`, der im Materialversorgungsmodell ein zentraler Stellenwert zukommt.

Im Rahmen der fachlichen Modellierung konnten wir bislang offen lassen, unter welcher Schnittstelle Materialien *zwischen* dem `Materialverwalter` und den Werkzeugen der Umgebung transportiert werden. Die Beschäftigung mit Modellen zur Materialversorgung ist ja in erster Linie softwaretechnisch begründet und aus fachlicher Sicht wenig relevant. Beim Übergang vom fachlichen zum technischen Entwurf wird sich jedoch an der Materialschnittstelle eines Werkzeuges manifestieren müssen, ob hier - im Sinne der beschriebenen Materialverwaltung - mit *verwaltbarem* Material umgegangen wird oder nicht. `tManageable` ist in diesem Sinne offensichtlich mehr als eine "reine" Aspektklasse, wenn wir darunter nur werkzeug- bzw. automaten*spezifische* Materialschnittstellen verstehen wollen. Sie ist vielmehr gleichzeitig *Materialschnittstelle* für die Klasse `tManager` und *Transportschnittstelle* für solche Werkzeugklassen, die Materialien künftig aus ihrem Kontext beziehen möchten.

Eine Anbindung an das Materialversorgungsmodell setzt aus Werkzeugsicht also voraus, daß rein fachlich motivierte Aspekte weiter gefaßt werden müssen als zuvor. Denn nur so können

Werkzeuge gleichzeitig Materialien unter der `tManageable`-Schnittstelle aus ihrem Kontext beziehen und weiterhin über den je werkzeugspezifischen Aspekt auf sie zugreifen.[14]

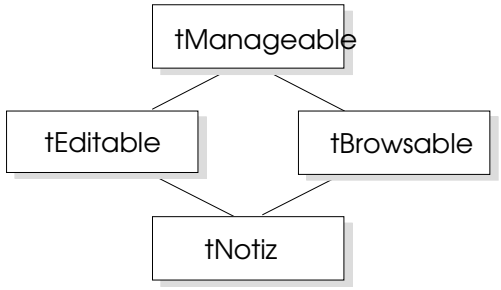


Abb. 8: Die fachlich motivierten Aspekte `tEditable` und `tBrowsable` müssen weit genug gefaßt werden, um die Anbindung eines Werkzeuges an das Materialversorgungsmodells zu ermöglichen.

Softwaretechnisch läßt sich dies mit Hilfe von Vererbung realisieren, d.h. werkzeugspezifische Aspektklassen wie `tBrowsable` oder `tEditable` werden von der Klasse `tManageable` abgeleitet. Die Modellierung durch Spezialisierung erzwingt die Einführung virtueller Basisklassen (hier: `tManageable`), da ein Material andernfalls über seine sämtlichen Aspektklassen mehrfach deren gemeinsame Basisklasse `tManageable` erbt.

Aus technischen Gründen müssen wir auf solche Strukturen verzichten. Zum einen ist es in der Sprache C++ nicht möglich, von einer virtuellen Basisklasse abgeleitete Klassen per *Typecasting* zu konvertieren (vgl. Kapitel 6.1.3.3.), zum anderen unterstützt die von uns

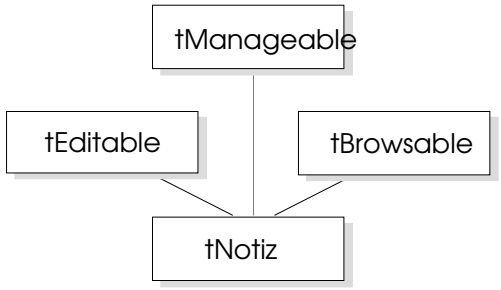


Abb. 9: Materialklassen müssen aus technischen Gründen direkt von der Aspektklasse `tManageable` abgeleitet werden.

verwendete, objektorientierte Datenbankprogrammiersprache ONTOS überhaupt keine virtuellen Basisklassen. Wir arbeiten daher mit dem Klassenmodell aus Abb. 9. Dieses Klassenmodell ist ein rein technisch bedingtes Zugeständnis an die Restriktionen von

[14] Es ist uns durchaus bewußt, daß wir implizit dafür argumentieren, Werkzeuge auf der softwaretechnischen Ebene nicht mehr ausschließlich nur über werkzeugspezifische Aspekte auf ihre Materialien zugreifen zu lassen. Hingegen ist es langfristig sicherlich wünschenswert, Werkzeugen ihre Materialien in *werkzeugspezifisch typisierten*, lokalen Magazinen zur Verfügung zu stellen. In Anbetracht der Unmöglichkeit, virtuelle Methoden in C++ kovariant zu redefinieren, setzt eine Umsetzung dieses Anspruches derzeit leider voraus, daß die Klasse `tManager` statisch, d.h. bereits zur Übersetzungszeit, die Aspektklassen sämtlicher Werkzeuge kennt, die in das Materialversorgungsmodell eingebunden werden sollen. Wir haben bei der Implementierung daher auf werkzeugspezifisch typisierte Magazine verzichtet.

ONTOS und C++ und soll nicht ausdrücken, daß in der Klasse `tManageable` von Eigenschaften des Materials abstrahiert wird, die - im fachlichen Sinne - für das richtige "Funktionieren" des Werkzeuges relevant sind.

Es sei darauf hingewiesen, daß das von der Klasse `tManageable` spezifizierte Protokoll zwar die für das richtige Funktionieren der Materialverwaltung notwendigen *Identifikationsdienste* bereitstellt, diese allein aber zumindest in zweierlei Hinsicht nicht ausreichend sind:

• Bei der fachlichen Modellierung haben wir die Unterstützung eines Interpreterprotokolls zur Auswertung von Prädikaten über Materialnamen und -typen seitens des `Materialverwalters` gefordert. Deskriptive Materialspezifikationen lassen sich aber überhaupt nur dann auswerten, wenn Wissen über die gesamte Menge der existierenden Materialexemplare vorhanden ist. Insofern es in verteilten Werkzeugumgebungen

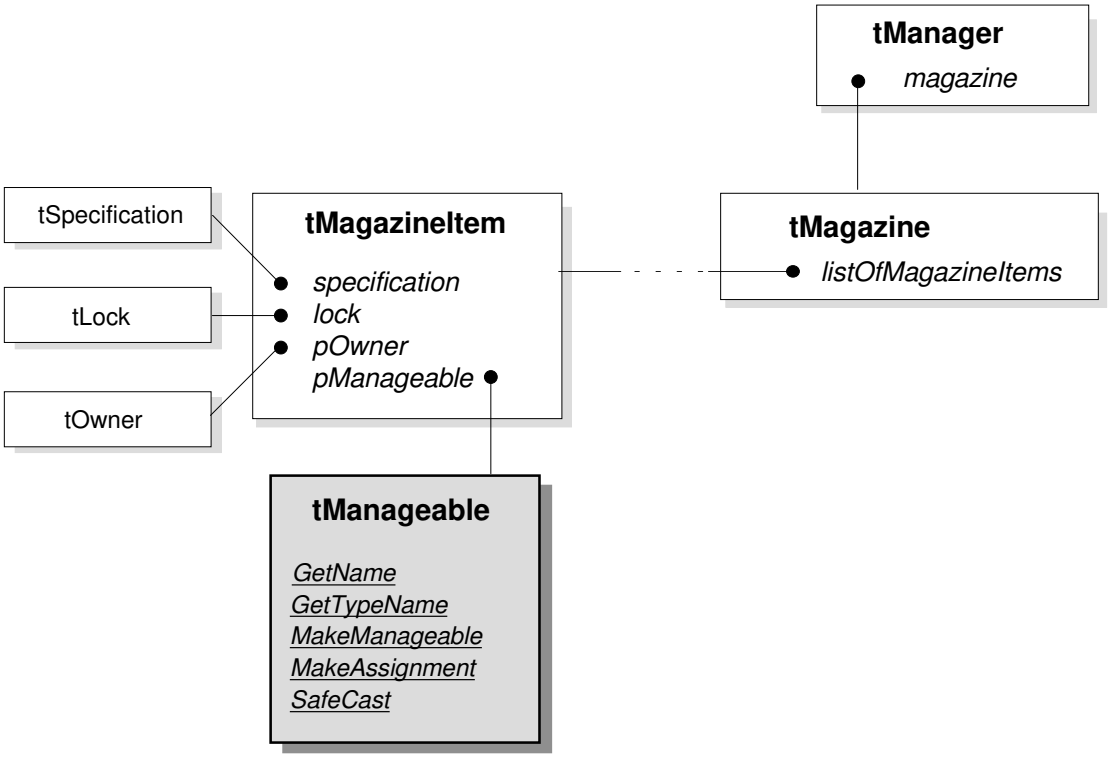


Abb. 10: Technischer Entwurf der Klasse `tManageable`.

unmöglich ist, ständig *alle* Materialexemplare vorrätig zu halten, muß ein `Materialverwalter` unabhängig davon, welche Materialexemplare gerade in der Werkzeumgebung aktiviert sind, auf für die Auswertung notwendige Informationen zugreifen können. Im technischen Entwurf haben wir zu diesem Zweck die Klasse `tMagazineItem` eingeführt. Bei ihrer Erzeugung initialisieren `tManager`-Objekte ihre Magazine mit einer Vielzahl von `tMagazineItem`-Objekten, von denen jedes einzelne die Spezifikation eines Materialexemplars (*specification*) enthält und die zusammen ein vollständiges Bild aller existierenden Materialien geben.[15] Materialexemplare werden dann

[15] Hier deutet sich bereits an, daß der Komplexität von Prädikaten aus pragmatischen Erwägungen heraus Grenzen gesetzt sind. Für jedes zusätzliche Attribut, das bei der Materialspezifikation von Werkzeugklienten verwendet werden darf, muß eine entsprechende Information innerhalb der `tMagazineItem`-Objekte verfügbar gemacht werden. SQL-artige Prädikate etwa setzen dann gar komplette Materialinformationen voraus.

nach ihrer Aktivierung in die Werkzeugumgebung in das zu ihnen passende `tMagazineItem`-Objekt eingebettet und können von dort aus Werkzeugklienten zur Bearbeitung verfügbar gemacht werden.

• Die Entkopplung der Klassen `tManageable` und `tMagazineItem` erweist sich auch bei der Realisierung einer kooperativen Materialzugriffskontrolle nach dem Bild von Original und Kopie als vorteilhaft. Die Information, ob ein Material bereits als Original für einen Werkzeugklienten reserviert worden ist (*lock*), muß beispielsweise auch dann verfügbar sein, wenn das betreffende Materialexemplar noch nicht aktiviert ist - ein ähnliches Problem also wie im Falle der Materialspezifikationen. Darüberhinaus ermöglicht Entkopplung, Wissen über einen Originalinhaber (*pOwner*) lokal zu halten und nicht unnötigerweise in Form eines Attributs in die Materialien hineinzuimplementieren.

6.1.3.1. Materialien identifizieren

```
virtual tString    GetName        () const = 0;
virtual tString    GetTypeName    () const = 0;
```

Kontext

Materialexemplare sollen innerhalb von verteilten Werkzeugumgebungen eindeutig über einen benutzerdefinierten Namen identifiziert werden. Darüberhinaus können sie nach ihrem Typ klassifiziert werden. Materialklassen, die in das Versorgungsmodell eingebunden werden sollen, verpflichten sich daher zur Implementierung der Methoden `GetName()` und `GetTypeName()`.

6.1.3.2. Materialien abstrakt erzeugen und zuweisen

```
virtual tManageable* MakeManageable () const  = 0;
virtual void MakeAssignment( tManageable* aManageable_p) = 0;
```

`MakeManageable()` zählt im weitesten Sinne zum Genre der Fabrikmethoden[16]. Die Methode wird von `tManager`-Objekten verwendet, um die in ihrem Magazin verwahrten Materialexemplare nicht als direkte Referenzen, sondern als Speicherkopien nach außen verfügbar machen zu können. Über die Methode `MakeManageable()` fordern sie ein Materialexemplar abstrakt dazu auf, eine Kopie von sich anzulegen. Abgeleitete Materialklassen implementieren die Methode, indem sie ein neues Exemplar ihres Typs per Kopierkonstruktor erzeugen und als direkte Referenz zurückliefern.

In der umgekehrten Richtung verwenden `tManager`-Objekte die Methode `MakeAssignment()`, um Änderungen, die ein Werkzeugklient lokal an einer Materialspeicherkopie vorgenommen hat, an dem entsprechenden Materialexemplar ihres eigenen Magazins nachzuziehen und damit global in der Werkzeugumgebung verfügbar zu machen. Dabei fordern sie ihr "privates" Materialexemplar abstrakt dazu auf, das als Parameter übergebene, modifizierte Material des Werkzeugklienten auf sich selbst zuzuweisen. In abgeleiteten Materialklassen kann dies als attributweise Zuordnung implementiert werden.

---

[16] Detaillierte Informationen über abstrakte Fabriken und Fabrikmethoden finden sich in [Gam92], S. 104ff. und S. 148f..

6.1.3.3. Materialien typsicher konvertieren

```
virtual void* SafeCast( tString aTypeName ) const = 0;
```

Bei dem Klassenmodell aus Abb. 9 müssen Materialexemplare gegebenenfalls zwischen einem fachlich motivierten Aspekt und ihrer `tManageable`-Schnittstelle konvertiert werden. In `tDepot`-Objekten etwa sind sie unter ihrer Schnittstelle `tManageable` bekannt, fachlich aber werden sie unter dem je werkzeugspezifischen Aspekt eines Werkzeugklienten bearbeitet. Damit Materialien *typsicher* zwischen Aspekten konvertiert werden können, muß jede Materialklasse die Methode `SafeCast()` implementieren, die ein Exemplar der Klasse, sofern zulässig, in den per Stringparameter spezifizierten Aspekt konvertiert.[17]

6.1.4. Die Klasse `tPattern`

```
tPattern( const char* s );
tPattern( const tString& s );

friend Bool operator==(     const tPattern& aPattern,
                            const tString&  aString );
```

Kontext

`tPattern` ist eine spezialisierte `tString`-Klasse und ermöglicht die Bildung und Interpretation von Stringausdrücken, die Wildcard- bzw. Jokerzeichen enthalten. Exemplare dieses Typs werden bei der Formulierung von `Materialspezifikationen`, also Prädikaten über Materialnamen und -typen, benötigt.

Beispiel `tPattern myPattern( "Notiz?");`

6.1.5. Die Klasse `tRequest`

```
tRequest( tPattern aNamePattern, tString aTypeName );

tPattern  GetNamePattern();
tString   GetTypeName   ();
```

[17] Sprachen mit statischer Typprüfung erzwingen, daß die von der Methode `SafeCast()` gelieferten Materialreferenzen von einem eindeutigen, an der Schnittstelle deklarierten Typ sind. Da Aspektklassen kein gemeinsames, von einer Oberklasse geerbtes Protokoll implementieren, bietet sich nur `void*` als Typ des Rückgabewertes an.

Kontext

`Materialspezifikationen` werden mit Hilfe von `tRequest`-Objekten implementiert. An der Klassenschnittstelle wird neben dem Materialtyp ein `Pattern` für den Materialnamen erwartet.

Beispiel

```
tRequest myRequest( tPattern( "Notiz?"), tString( "tNotiz") );
```

6.1.6. Die Automatenklasse `tSupplier`

Bei der fachlichen Modellierung haben wir `Materialversorger` eingeführt, die als Kommunikationsautomaten am Rand einer Werkzeugumgebung angesiedelt sind und das Wissen um konkrete Schnittstellen zur Außenwelt kapseln (vgl. Kapitel 3.1.).

Zu ihren Verantwortlichkeiten zählt insbesondere der Transport von Materialien zwischen dem `Materialverwalter` und beliebigen externen Datenversorgern. Der `Material-versorger` dient uns als Ausgangspunkt bei der Konstruktion seines technischen Pendants, der Klasse `tSupplier`.

In Kapitel 6.1.1. haben wir bereits angesprochen, daß `tSupplier` als abstrakte Klasse die Wurzel einer möglichen Klassenfamilie bildet, deren spezialisierte Materialversorgerklassen ein gemeinsames Protokoll für die Kommunikation mit externen Versorgern wie beispielsweise relationalen oder objektorientierten Datenbanken unterstützen. Die Klasse `tManager` ist mit diesen spezialisierten Materialversorgerklassen abstrakt gekoppelt, d.h. Objektkommunikation findet ausschließlich auf der Ebene des abstrakten Protokolls statt.

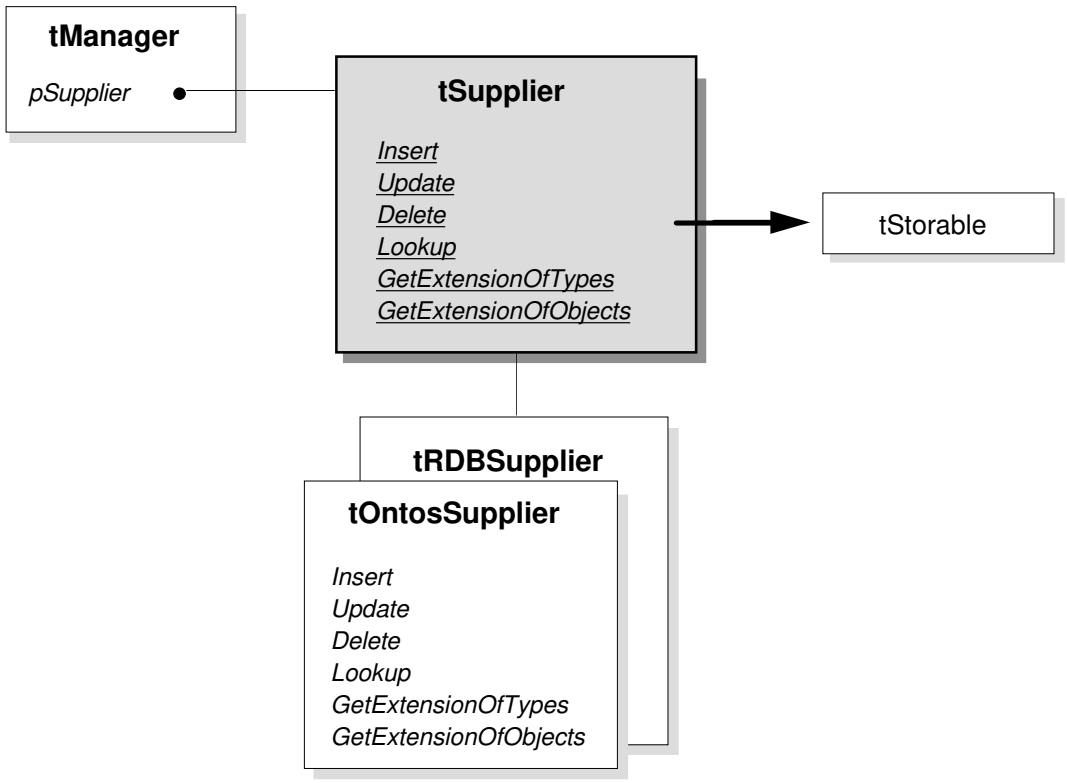


Abb. 11: Technischer Entwurf der abstrakten Automatenklasse `tSupplier`.

Das in objektorientierten Programmiersprachen übliche Konzept der *späten Bindung* (vgl. [Kil93], S. 206) ermöglicht, die Zugriffe zur Laufzeit dynamisch an den je spezialisierten Automaten zu binden. Werkzeugumgebungen können auf diese Weise flexibel um beliebige Materialversorgeranschlüsse erweitert werden.

6.1.6.1. Materialien bei externen Datenversorgern löschen, einfügen und ändern

```
tState Insert( tStorable* aStorable_p ) = 0;
tState Update( tStorable* aStorable_p ) = 0;
tState Delete( tStorable* aStorable_p ) = 0;
```

Kontext

Wenn Herkunft und Lebensdauer von Materialien vor den Werkzeugen einer Umgebung verborgen bleiben sollen, müssen sich Änderungen am persistenten Materialbestand für Werkzeuge *transparent* bis zu ihrem externen Ursprung fortpflanzen. Im Rahmen des vorgeschlagenen Versorgungsmodells sind hierfür die `tManager`-Objekte als unmittelbare Ansprechpartner von Werkzeugen zuständig. Sie sorgen dafür, daß mit persistenten Materialien, die auf explizite Anfrage eines Werkzeugklienten in der Umgebung gelöscht, eingefügt oder geändert werden, auf gleiche Weise auch in dem je zuständigen, externen Datenversorger verfahren wird. Hierzu bedienen sie sich der Methoden `Insert()`, `Update()` und `Delete()`.

Parameter

`aStorable_p`	ist ein Verweis auf das neu eingefügte, geänderte oder gelöschte Materialexemplar. Materialien werden der Klasse `tSupplier` stets unter ihrer abstrakten Schnittstelle als `tStorable` bekannt gemacht (vgl. Kapitel 6.1.7.)

6.1.6.2. Materialien von externen Datenversorgern rekonstruieren

```
tStorable* Lookup( tSpecification aSpecification ) = 0;
tListOfStorables* Lookup( tListOfSpecifications ) = 0;
```

Kontext

`tManager`-Objekte können eine an sie gerichtete Materialanforderung gegebenenfalls erst durch Anfrage bei einem ihrer Versorgerautomaten befriedigen. Die Klasse `tSupplier` bietet an ihrer Schnittstelle zu diesem Zweck verschiedene `Lookup()`-Methoden an, mit denen je nach Verwendungskontext ein einzelnes oder eine Vielzahl von Materialexemplaren in die Werkzeugumgebung aktiviert werden kann. Materialien werden dabei über benutzersichtbare Namen und Typen spezifiziert.

Parameter

`aSpecification`	ist eine abstrakte Materialbeschreibung bestehend aus dem Namen und dem Typ eines Materials. Im Unterschied zu Exemplaren des Typs `tRequest` sind keine Wildcard- oder Jokerzeichen erlaubt: `tSpecification specification( tString("Notiz1"), tString("tNotiz"));`
`aListOfSpecification_p`	ist ein Verweis auf eine Liste abstrakter Materialbeschreibungen des Typs `tSpecification`.

6.1.6.3. Die Materialextension eines externen Datenversorgers erfragen

```
tListOfSpecifications* GetExtensionOfTypes () = 0;
```

Kontext

Bei seiner Erzeugung initialisiert ein `tManager`-Objekt sein Magazin mit einer Vielzahl von `tMagazineItem`-Objekten, von denen jedes einzelne die Spezifikation eines Materialexemplars enthält und die zusammen ein vollständiges Bild aller innerhalb der verteilten Werkzeugumgebung existierenden Materialien ergeben (vgl. Kapitel 6.1.3.). Die notwendigen Informationen erhält das `tManager`-Objekt, indem es über seine sämtlichen Materialversorgerautomaten iteriert und sich von jedem einzelnen eine aus Name und Typ bestehende Beschreibung aller von ihm verwahrten Materialexemplare liefern läßt. Spezialisierte Materialversorgerklassen müssen zu diesem Zweck die in ihrer gemeinsamen Oberklasse `tSupplier` abstrakt spezifizierte Methode `GetExtensionOfObjects()` implementieren.

6.1.6.4. Die Materialtypen eines externen Datenversorgers erfragen

```
tListOfTypeNames* GetExtensionOfTypes () = 0;
```

Kontext

`tManager`-Objekte sind mit ihren Materialversorgerobjekten bekanntlich abstrakt gekoppelt, d.h. ihre Kommunikation findet ausschließlich auf der Ebene des `tSupplier`-Protokolls statt. Die Typeninformation einzelner Materialexemplare garantiert in Verbindung mit einer eindeutigen Abbildung von Materialtypen auf Versorgerautomaten, daß ein `tManager`-Objekt stets den für die Durchführung einer Operation zuständigen Materialversorgerautomaten ermitteln kann. Im Zuge seiner Erzeugung initialisiert jedes `tManager`-Objekt daher eine entsprechende Zuordnungstabelle (*supplierTypeTable*), indem es über seine sämtlichen Materialversorgerautomaten iteriert und sich von jedem einzelnen eine Liste der von ihm verwalteten Materialtypen liefern läßt. Spezialisierte Materialversorgerklassen müssen zu diesem Zweck die in ihrer gemeinsamen Oberklasse `tSupplier` abstrakt spezifizierte Methode `GetExtensionOfTypes()` implementieren.

6.1.7. Die Aspektklasse `tStorable`

In Kapitel 5.3. haben wir gezeigt, daß Materialien einem Datenbankautomaten konzeptionell über den Aspekt `Speicherbar` verfügbar gemacht und auf diese Weise aus einer Datenbank in die Umgebung geladen bzw. aus der Umgebung zurück in die Datenbank überführt werden können. Ein entsprechendes Protokoll wird bei der technischen Umsetzung von der abstrakten Aspektklasse `tStorable` spezifiziert, deren abgeleitete Materialklassen sich zur Implementation der beiden Methoden `Store()` und `Delete()` verpflichten.

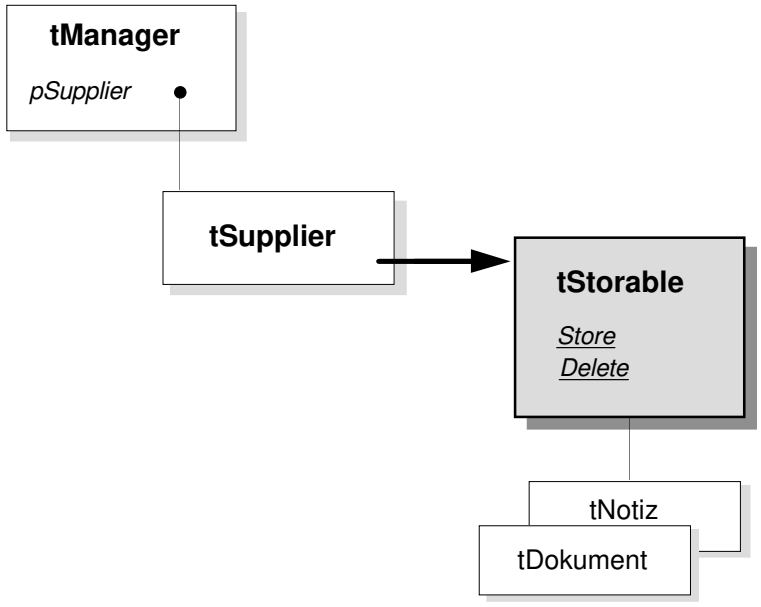


Abb. 12: Technischer Entwurf der Aspektklasse `tStorable`.

Das resultierende Klassenmodell impliziert ein Verständnis von Materialien, das sich nicht ohne weiteres mit der hier zugrundegelegten Materialphilosophie (vgl. Kapitel 2.2.1) in Einklang bringen läßt, gleichzeitig aber typisch für objektorientierte Datenbankprogrammiersprachen ist.

Bei der Kurzvorstellung der objektorientierten Datenbank ONTOS (vgl. Kapitel 5.2.) haben wir am Rande erwähnt, daß Anwendungen ihre Datenbankfunktionalität durch Einbinden einer Klassenbibliothek erwerben, die u.a. die Basisklasse aller persistenten Objekte, `Object`, zur Verfügung stellt. Materialexemplare einer verteilten Werkzeugumgebung können nur dann in ONTOS-Datenbanken gespeichert und gelöscht werden, wenn ihre Materialklassen von `Object` die entsprechenden Methoden `putObject()` bzw. `deleteObject()` erben.

In diesem Sinne erscheint die Schnittstelle der Klasse `tStorable` durchaus plausibel: Materialklassen, die dem Datenbankautomaten über den Aspekt `tStorable` verfügbar gemacht werden sollen, verpflichten sich zur Implementation von `Store()` und `Delete()`, deren Code sie in Abhängigkeit von dem gerade verwendeten Datenbankprodukt erben.

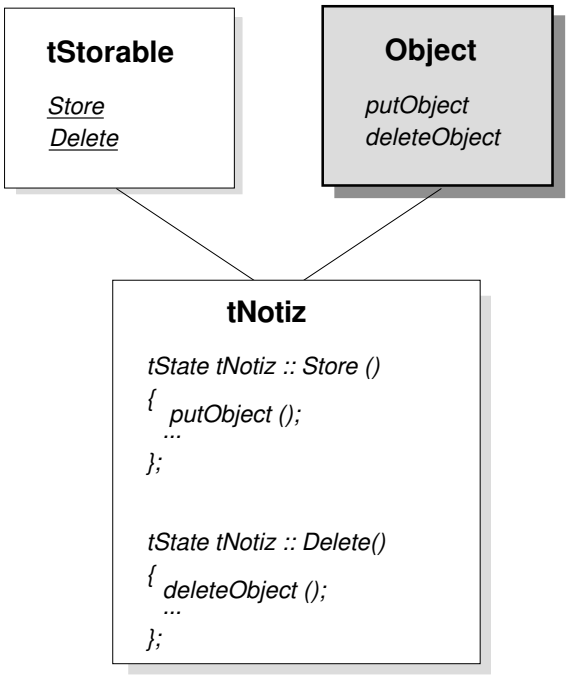


Abb. 13: Bei der Verwendung objektorientierter Datenbankprogrammiersprachen müssen Materialklassen ihre Datenbankfunktionalität i.d.R. von einer gemeinsamen Oberklasse (hier: `Object`) *erben*.

Die Analogie der Schnittstellen von `tStorable` und `Object` ist keinesfalls unumstritten. Kilberth et al., die die konzeptionelle Anbindung einer *relationalen* Datenbank erörtern, kommen zu dem Schluß, daß das von `Speicherbar` spezifizierte Protokoll "konzeptionell aus einer Routine, die eine Überführung eines Objektes in eine an Datenbanktypen orientierte Repräsentation vornimmt", besteht (vgl. [Kil93], S. 77). Der Datenbankautomat ist dann dafür zuständig, das Objekt auf der Basis dieser Repräsentation zu speichern, in der Regel, indem er einzelne Attributwerte rekursiv bis zu den Basistypen in die Felder entsprechender Relationen einträgt. Auf diese Weise kann in Materialklassen Wissen über ihre Speicherbarkeit modelliert werden, ohne konkrete Annahmen darüber zu treffen, in welchen Arbeitszusammenhängen Materialexemplare wo von wem gespeichert werden.

Eine solche Abbildung vom Anwendungsschema in das Datenbankschema eines Materials erübrigt sich bei objektorientierten Datenbankprogrammiersprachen, die sich ja gerade dadurch auszeichnen, daß der Strukturbruch zwischen den Datenstrukturen des Hauptspeichers und denen der Datenbank überwunden wird. Der Preis, den man hierfür zahlen muß, drückt sich technisch dann gerade in solchen Methoden wie `Store()` und `Delete()` aus, die dem Materialbegriff aus Kapitel 2.2.1. in zwei wesentlichen Punkten zuwiderlaufen:

• Materialklassen werden zwangsläufig abhängig von konkreten Systemplattformen. Die Umstellung auf eine neue Datenbank wirkt sich unmittelbar auf die Implementierung einer gegebenenfalls großen Anzahl unterschiedlicher Materialklassen aus. Darüberhinaus wird die Speicherung von Materialexemplaren desselben Typs in verschiedenen Datenbanken ausgeschlossen, was die langsame Migration von Objekten *zwischen* Datenbanken, etwa bei einer Umstellung, praktisch unmöglich macht.

• Die Implementierung der Methoden `Store()` und `Delete()` läßt sich nicht immer so geradlinig von der Basisklasse `Object` übernehmen, wie es die vorige Abbildung suggeriert. In der Regel müssen persistente Objekte die gesamte Speicherverwaltung für

eingebettete bzw. referenzierte Objekte selbst übernehmen. Gerade bei hypertextartigen Werkzeugumgebungen mit der für sie typischen Verknüpfung einzelner Dokumente zu komplexen Netzen müssen sich Materialklassen daher in großem Stile damit auseinandersetzen, wann beispielsweise Materialexemplare ihre Referenzen verfolgen sollen, ob das hinter einer Referenz stehende Objekt beim Speichern des Materialexemplars mitgespeichert werden muß, usw. Sobald solche Entscheidungen auf der Ebene von Materialklassen getroffen und implementiert werden, haben Werkzeuge aber keinerlei Kontrolle mehr darüber, in welchen Arbeitszusammenhängen welche Materialexemplare wo gespeichert werden, so daß sich das Rollenverhältnis der "aktiven" Werkzeuge zu ihren "passiven" Materialien genau umkehrt.

Die beschriebenen Eigenarten objektorientierter Datenbankprogrammiersprachen ließen sich möglicherweise umgehen, wenn man - wie im relationalen Fall - eine (künstliche) Trennung zwischen dem Anwendungsschema und dem Datenbankschema einer Materialklasse vollzieht und die Speicherung und Rekonstruktion von Objekten damit wieder voll in die Hände eines Datenbankautomaten legt. Wir haben davon Abstand genommen, weil es uns hochgradig paradox erscheint und die Frage nahelegt, warum hier überhaupt mit einem objektorientierten Datenbankanschluß gearbeitet wird. Wir schlagen vielmehr vor, die Schnittstelle der Aspektklasse `tStorable` um die abstrakte Methode `ErzeugeDatenbankRepräsentation` zu erweitern (vgl. [Kil93], S. 78), um die Anbindung relationaler Datenbanken im Sinne von Kilberth et al. zu ermöglichen.

6.2. Softwaretechnische Lösungen für das Problem der Verteilung in kooperativen Werkzeugumgebungen

Die Verwaltung gemeinsamer Materialien nach dem Bild von Originalen und Kopien kann in verteilten Werkzeugumgebungen bekanntlich mit Hilfe eines `Materialkoordinators` realisiert werden. Bei der *konzeptuellen* Darstellung seines Zusammenspiels mit den `Materialverwaltern` der einzelnen Werkzeugumgebungen (vgl. Kapitel 4.2.) konnten wir stillschweigend darüber hinwegsehen, daß ihre korrespondierenden technischen Komponenten zur Laufzeit in voneinander unabhängigen Prozessen angesiedelt sind. Wir wollen in diesem Abschnitt darlegen, wie die damit verbundenen Kommunikationsprobleme softwaretechnisch gelöst werden können.

6.2.1. Client/Services-Architekturen: Verteilte Werkzeugumgebungen nach dem Dienstleistungsprinzip entwerfen

Das Dienstleistungsprinzip[18] (*Client/Services*-Prinzip) ist eine tragfähige Technik bei der Konstruktion verteilter Softwarearchitekturen. Einzelne Architekturkomponenten realisieren eine festgelegte, implementations- und ortsunabhängige Dienstleistung, deren Schnittstelle sie als Dienstleistungsanbieter nach außen exportieren und so beliebigen Leistungsabnehmern verfügbar machen. `Materialverwalter` und `Materialkoordinator` verstehen wir in diesem Sinne als Leistungsabnehmer bzw. Leistungsanbieter von Koordinationsdiensten.

Im folgenden beschäftigen wir uns schwerpunktmäßig damit, *wie* der Leistungsaustausch zwischen den beiden Komponenten realisiert werden kann. Bei der technischen Umsetzung haben wir uns insbesondere von dem Vorsatz leiten lassen, notwendige Kommunikationsvorgänge, die auf der untersten Ebene durch Socketverbindungen realisiert

[18] Wir beziehen uns hier auf das *Dienstleistungsprinzip*, wie es beispielsweise in [Kil93], S. 49ff. dargestellt wird.

werden, möglichst frühzeitig zu kapseln, um davon auf höheren Ebenen abstrahieren und Konzepte gewissermaßen über "Prozeßränder" hinaus modellieren zu können. Der Grundgedanke ist, den `Materialkoordinator` nicht 1:1 in einen technischen Klassenentwurf zu überführen, sondern auf der Seite der Leistungsabnehmer mit Stellvertreterobjekten (*Proxies*) zu arbeiten, die den Leistungsanbieter repräsentieren. Bevor wir diese Idee im Detail ausführen, verlieren wir einige grundsätzliche Worte über die Realisierung von Inter-Prozeß-Kommunikation auf der Basis von Socketverbindungen.

6.2.1.1. Inter-Prozeß-Kommunikation durch Sockets realisieren

Voneinander unabhängige Prozesse können über den Austausch von Nachrichten miteinander kommunizieren. Eine mögliche und zudem recht einfache Implementierung läßt sich mit Hilfe sogenannter *Sockets* realisieren[19]. Sockets zeichnen sich durch hohe Übertragungsgeschwindigkeiten sowie eine geringe Netzbelastung aus und werden im UNIX-Sprachgebrauch als *Endpunkte* einer Verbindung zwischen zwei unabhängigen Prozessen aufgefaßt. Bei der Socketprogrammierung wird zwischen unterschiedlichen *Typen* und *Domänen* eines Sockets differenziert. In unserem Kontext verwenden wir einen übertragungssicheren, permanenten und streamorientierten Sockettyp innerhalb der *Internet*-Domäne. Verbindungen können also zwischen beliebigen Maschinen mit gültiger Internet-Adresse aufgebaut werden, die zusammen mit einer lokalen Portnummer die eindeutige Identifizierung eines Verbindungsendpunktes gestattet. Der verwendete Sockettyp garantiert, daß Nachrichten sicher und in der korrekten Reihenfolge verschickt und empfangen werden. Darüberhinaus wird ein allzu großer Laufzeitoverhead durch ständigen Auf- und Abbau von Socketverbindungen vermieden.

Wir beschreiben Kommunikationsendpunkte und die für sie typische Funktionalität in der Klasse `tSocket`. Der Auf- und Abbau eines Kommunikationsendpunktes erfolgt mit der Erzeugung bzw. Zerstörung eines Socketobjekts. Die Verbindung zweier Socketobjekte muß explizit herbeigeführt werden und ist stets asymmetrisch: nur wenn einer der beiden Endpunkte explizit benannt und in diesem Sinne auch adressierbar ist, kann ein zweiter sich mit ihm verbinden lassen. Voraussetzung ist in jedem Fall, daß das addressierbare Socketobjekt bereit ist, eine Verbindung zu akzeptieren. Die Anzahl der Verbindungen, die ein Kommunikationsendpunkt eingehen kann, ist prinzipiell beliebig und wird nur durch das Betriebssystem nach oben begrenzt. Nachrichten können nach dem Aufbau einer Verbindung als Pakete in beiden Richtungen gesendet und empfangen werden.

6.2.1.2. Von Proxyobjekten und anderen Konzepten, die die Kommunikation zwischen `Materialverwalter` und `Materialkoordinator` verständlicher machen

`Materialverwalter` benutzen den zentralen `Materialkoordinator` bekanntlich, um die Kommunikation mit ihren Materialversorgerautomaten in eine fachlich motivierte Zugriffskontrolle nach dem Bild von Originalen und Kopien einzubinden (vgl. Kapitel 4.2.). Unser Anliegen ist, dem `Materialverwalter` einen *transparenten* Umgang mit dem `Materialkoordinator` zu ermöglichen, der ihn unberührt von Einzelheiten der Inter-Prozeß-Kommunikation läßt.

---

[19] Ausführliche Erläuterungen zu den beiden Themen *Inter-Prozeß-Kommunikation* und *Sockets* können in [Cur91] und [Sun92] nachgelesen werden, die auch die Grundlage dieses Abschnitts bilden.

Wir haben zu diesem Zweck alle Kommunikationsvorgänge auf der Ebene von Socketverbindungen in einer gesonderten Klasse, `tCoordinationClient`, gekapselt. Exemplare dieses Typs werden innerhalb von Werkzeugumgebungen - und hier insbesondere innerhalb der Klasse `tManager` - als Proxyobjekte für den zentralen Materialkoordinator benutzt, der technisch mit Hilfe der Klasse `tCoordinationServer` umgesetzt wird. In Abb.14 haben wir diese Zusammenhänge grafisch dargestellt. Auffallend ist die Gleichheit der Klassenschnittstellen von `tCoordinationClient` und `tCoordinationServer`, die überhaupt erst ermöglicht, ein Clientobjekt transparent anstelle eines Serverobjektes zu verwenden.

Proxies verwenden Socket-Objekte (*requestSocket*), um die Verbindung zu dem zentralen `tCoordinationServer`-Objekt aufzubauen und seine Dienste in Anspruch zu nehmen. Die Klasse `tSocket` stellt zu diesem Zweck an ihrer Schnittstelle eine Operation zur Verfügung, die nach dem Abschicken eines Paketes auf den Empfang eines entsprechenden Antwortpaketes wartet und dieses als Rückgabewert liefert, also gewissermaßen synchrone Methodenaufrufe über Prozeßränder hinweg *simuliert*.

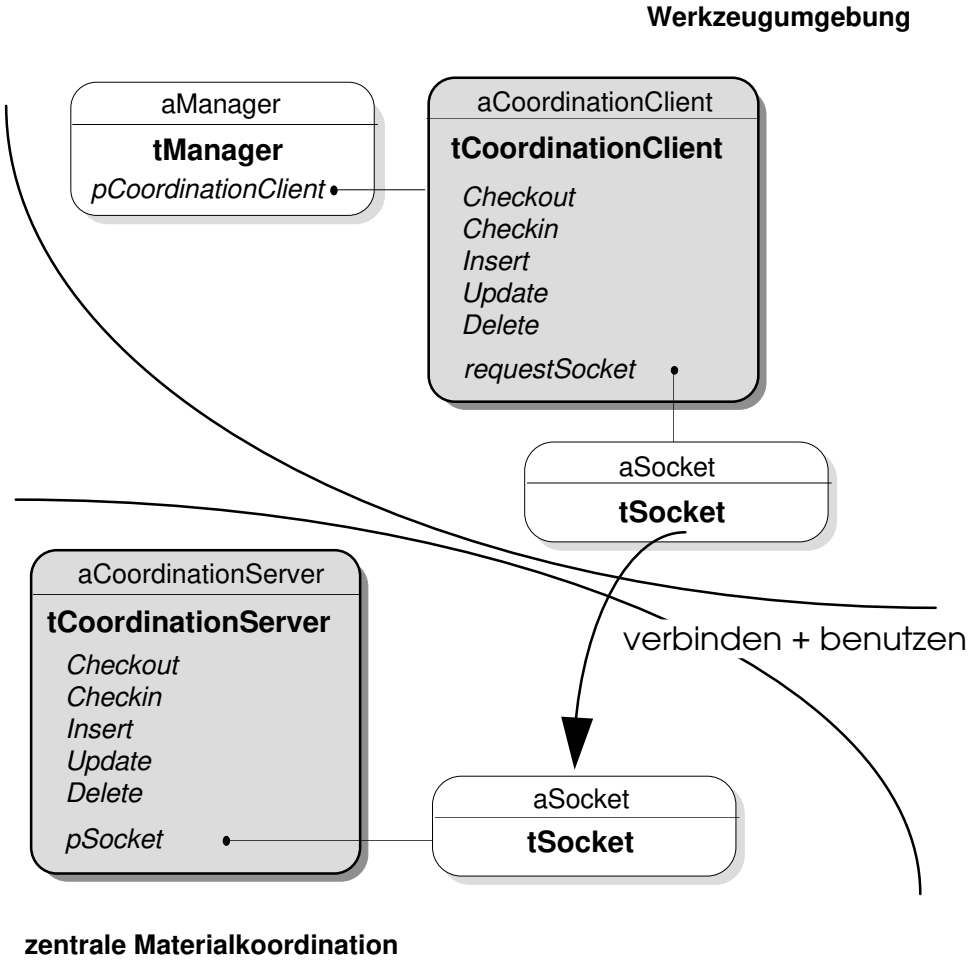


Abb. 14: Aufbau einer Prozeßverbindung zwischen einem `tCoordinationClient`- und `tCoordinationServer`-Objekt

Bei der Implementierung der Klasse `tCoordinationClient` haben wir ausgiebig von dieser Technik Gebrauch gemacht: Operationenaufrufe der Klasse `tCoordinationServer` werden zusammen mit ihren Argumenten, den Materialspezifikationen, in Form von Paketen kodiert, von dem jeweiligen Proxy per

Socketverbindung an das zentrale `tCoordinationServer`-Objekt gesendet, von diesem interpretiert und ausgeführt. Das Ergebnis der ausgeführten Operation wird ebenfalls als Paket kodiert, über die Verbindungsendpunkte zurück an das anfragende Clientobjekt gesendet und von diesem als Rückgabewert der gerufenen Operation interpretiert.

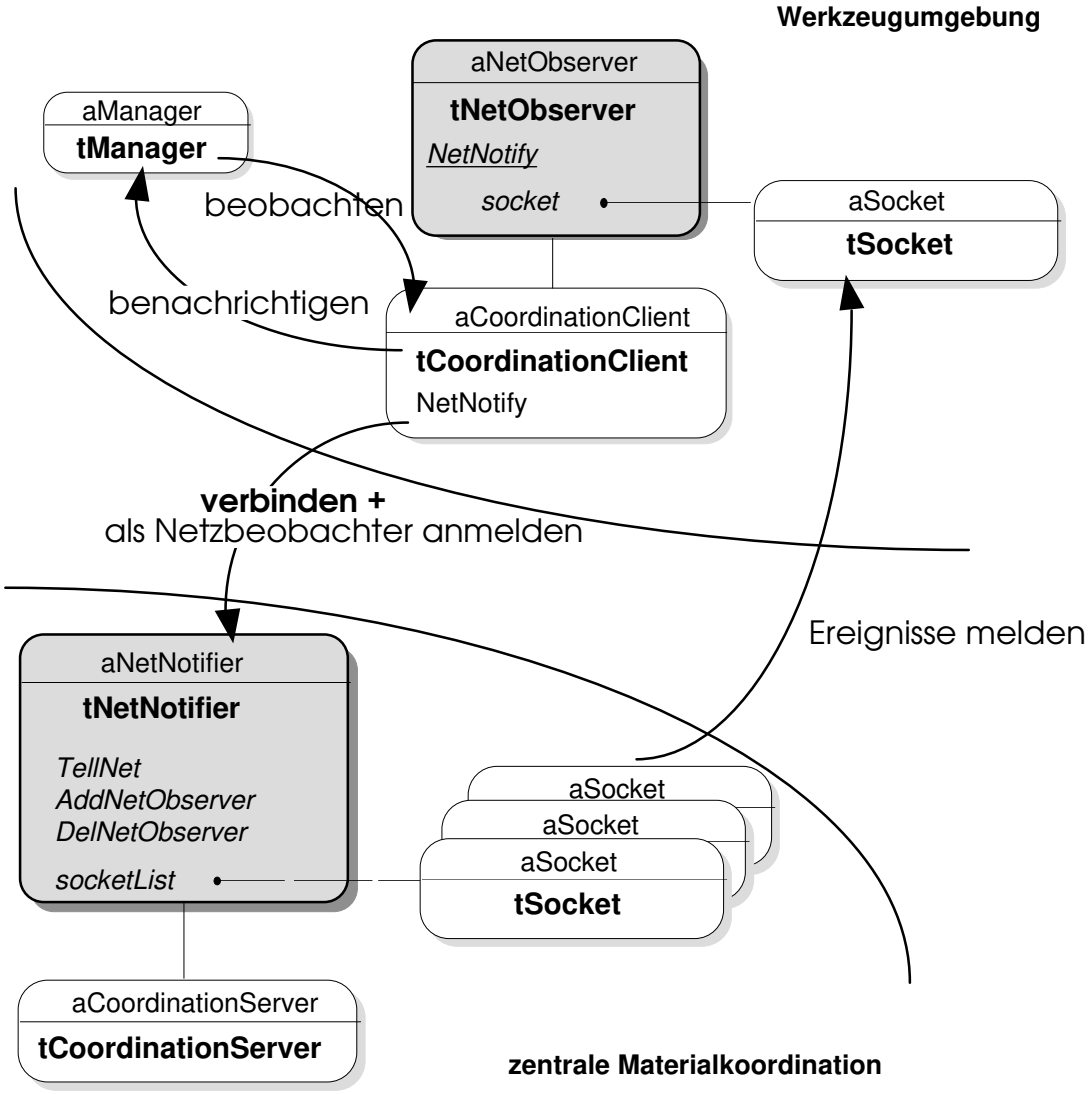


Abb. 15: Modellierung des Beobachtermechanismus bei einer verteilten Werkzeugumgebung.

Auf ganz ähnliche Weise läßt sich auch das `Beobachter`-Konzept (vgl. Kapitel 2.2.2.) über Prozeßränder hinaus modellieren. `tManager`-Objekte können transparent, d.h. über den für *nicht-verteilte* Anwendungen bereits implementierten "Klingelmechanismus"[20], von Änderungen am gemeinsamen Materialbestand benachrichtigt werden und bleiben weiterhin unabhängig von Einzelheiten der Inter-Prozeß-Kommunikation. Der Grundgedanke ist, daß `tCoordinationServer`-Objekte Ereignisse in Form von Paketen kodieren und über ein `tNetNotifier`-Protokoll an zu benachrichtigende Werkzeugumgebungen schicken, die es über ihre Proxyobjekte empfangen und auswerten. Die Klasse `tCoordinationClient`

[20] Die technische Umsetzung des *Klingelmechanismus* in nicht-verteilten Anwendungen wird in [Rie93b] ausführlich dokumentiert. Wir gehen daher an dieser Stelle nicht näher darauf ein, sondern verweisen auf Kapitel 3 der genannten Arbeit.

erbt zu diesem Zweck ein entsprechendes Protokoll von ihrer abstrakten Oberklasse `tNetObserver`.

`tNetNotifier` und `tNetObserver` ergänzen sich zu einem Netzbenachrichtigungsprotokoll, das sich *konzeptuell* nicht von dem bereits in anderen Klassenbibliotheken des Arbeitsbereiches verwendeten `tNotifier`-Protokoll[21] unterscheidet (vgl. [Rie93b]). `tCoordinationClient`-Objekte melden sich bei ihrer Erzeugung als neue Netzbeobachter bei der zentralen Materialkoordination an (`AddNetNotifier`) und werden damit künftig über jedes stattfindende Ereignis informiert. Bei ihrer Zerstörung melden sie sich dementsprechend auch wieder ab (`DelNetNotifier`).

Zu den typischen Ereignissen, die das zentrale `tCoordinationServer`-Objekt seinen Netzbeobachtern meldet (`TellNet`), zählen das Einfügen eines neuen Materials (`cEventInsert`), das Löschen (`cEventDelete`) bzw. Überschreiben (`cEventUpdate`) eines vorhandenen Materials und die Freigabe eines zuvor als Original reservierten Materials (`cEventCheckin`). Netzbeobachter verpflichten sich zur Implementierung einer in `tNetObserver` abstrakt spezifizierten Methode, innerhalb derer sie spezifisch auf gemeldete Netzereignisse reagieren (`NetNotify`). Vor dem Hintergrund, daß `tManager`-Objekte transparent, d.h. über das für *nicht-verteilte* Anwendungen bereits implementierte `tNotifier`-Protokoll von Änderungen am gemeinsamen Materialbestand benachrichtigt werden sollen, propagiert ein `tCoordinationClient`-Objekt Ereignisse hier sinnvollerweise an das ihn beobachtende `tManager`-Objekt[22] .

6.2.2. Verteiltes Objektmanagement - Standardisierung verteilter Anwendungen im objektorientierten Paradigma

Wie eingangs bereits angekündigt, möchten wir dieses Kapitel mit einem Blick auf die gegenwärtige Diskussion um Standards für verteilte Anwendungen im Rahmen des objektorientierten Paradigmas abschließen. Diese Diskussion wird maßgeblich von der Arbeit der *Object Management Group* (OMG) beeinflußt, die sich die Schaffung eines Industriestandards für die Verwaltung und den Zugriff auf netzwerkweit verteilte Objekte zum Ziel gesetzt hat und damit u.a. langfristig den derzeitigen Mangel an Interoperabilität von verteilter Software zu beheben hofft.[23]

Wesentliche Konzepte hat die OMG mit ihrer Spezifikation der *Common Object Request Broker Architecture*[24] (CORBA) vorgestellt. Für CORBA-basierte Systeme hat sich in der Literatur mittlerweile recht durchgängig die Bezeichnung des 'Distributed Object Management System' (DOMS) durchgesetzt. Zentrales Element der CORBA-Spezifikation ist der sogenannte *Object Request Broker* (ORB), der Mechanismen zur Verfügung stellt, die einen aus Anwendungssicht transparenten, von Herkunft und Lebensdauer unabhängigen Zugriff auf Objekte ermöglichen.[25] Die CORBA-Spezifikation ist das Ergebnis der

---

[21] Technisch ist der Unterschied natürlich erheblich. Beispielsweise können `tNetObserver`-Objekte Ereignisse überhaupt nur durch *Polling* der Prozeßverbindung zur zentralen Materialkoordination empfangen.

[22] Die Klasse `tManager` und `tCoordinationClient` erben dazu von der Beobachterklasse `tNotifier`. Der Übersichtlichkeit halber haben wir in Abbildung 15 davon Abstand genommen, die entsprechenden Objekte von *aManager* und *aCoordinationClient* einzuzeichnen.

[23] Wir beziehen uns im folgenden auf die Artikel von [Bey93], [Hor93], [Mow93] und [Str93]. Die Originaltexte der OMG stehen uns derzeit leider nicht zur Verfügung, ihre Titel werden jedoch im jeweils geeigneten Kontext zitiert.

[24] vgl. *The Common Object Request Broker: Architecture and Specification.* OMG Document Number 91.12.1, Revision 1.1.

[25] Transparenz im Umgang mit Materialien, Unabhängigkeit von ihrer Herkunft und Lebensdauer haben wir gleichfalls als die aus Werkzeugsicht entscheidenden Kriterien bei der Entwicklung eines verteilten

Zusammenführung von Entwürfen einzelner Hersteller, die in ihrem Bemühen, ein Rahmenwerk für verteilte Anwendungen zu schaffen, jeweils eigene, ORB-ähnliche Instanz entwickelt haben.

Das folgende Beispiel verdeutlicht die hinter CORBA stehenden Konzepte und zeigt gleichzeitig Perspektiven auf, mit denen das bereits diskutierte Verteilungsproblem kooperativer Werkzeugumgebungen (vgl. Kapitel 6.2.1.) alternativ gelöst werden könnte.

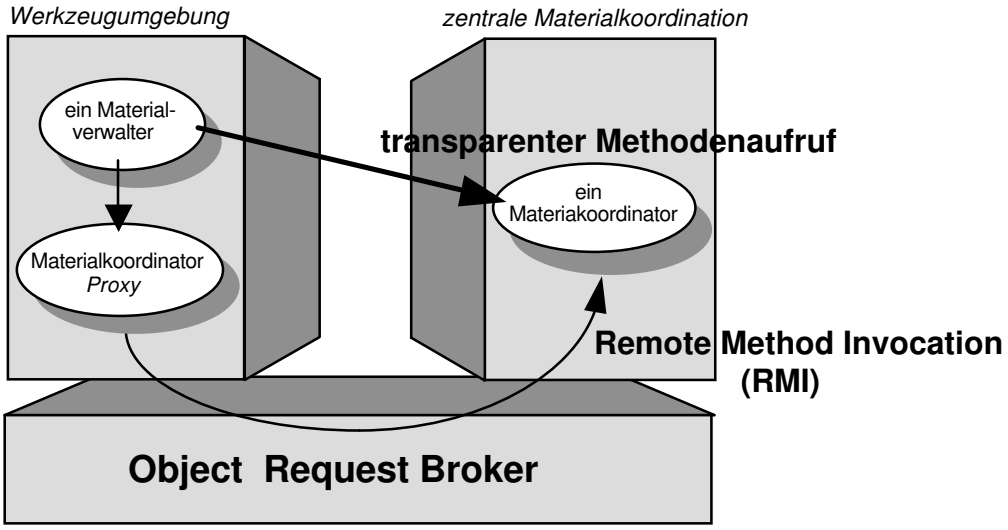


Abb. 16: Transparenter Zugriff auf ein verteiltes `Materialkoordinator` - Exemplar (in Anlehnung an [Gut93]).

Object Request Broker sind Serverprozesse auf Betriebssystemebene, deren Dienste von Clientanwendungen, etwa einer Werkzeugumgebung und einem zentralen Koordinationsprozeß, in Anspruch genommen werden können. Eine Werkzeugumgebung, die mit einem beliebig im Netz angesiedelten `Materialkoordinator` kommunizieren möchte, beschafft sich zunächst mit Hilfe des Object Request Brokers ein Proxyobjekt, das aus ihrer Sicht den "realen" `Materialkoordinator` repräsentiert. Die Werkzeugumgebung ruft daher auch ausschließlich Methoden des `Materialkoordinator`-Proxies, die intern in Anforderungen (requests) - bestehend aus dem Methodennamen und der geforderten Anzahl Parameter - umgesetzt, über den Object Request Broker an das reale, verteilte Objekt weitergeleiet und dort bearbeitet werden. Dieser in der Literatur teilweise als *Remote Method Invocation* (RMI) zitierte Vorgang und damit zusammenhängende Aufgaben, wie etwa das Auffinden des realen `Materialkoordinators` im Netz, werden innerhalb des Object Request Brokers gekapselt und bleiben damit vollständig vor den Architekturkomponenten der Werkzeugumgebung verborgen. Im Unterschied etwa zu RPC- oder Socket-Anwendungen, die die gesamte Verteilungssemantik unmittelbar im Applikationscode implementieren (vgl. Kapitel 6.2.1.) und damit insbesondere Annahmen darüber treffen, wo der gewünschte Kommunikationspartner residiert, werden mit dem ORB integrierte Anwendungen von solchem Detailwissen entlastet und können transparent, also in der gewohnten Programmiersprachensyntax, auch auf verteilte Objekte zugreifen. Als angenehme

---

Materialversorgungsmodells herausgearbeitet (vgl. Kapitel 3 und 4). In diesem Sinne handelt es sich klar um ein CORBA-verwandtes Konzept. Materialversorgung einerseits und CORBA-basiertes verteiltes Objektmanagement andererseits sind jedoch Diskussionsgegenstände voneinander unabhängiger Ebenen (fachlich vs. technisch) und können insofern gegeneinander abgegrenzt werden.

Konsequenz lassen sich Objekte bzw. die sie einbettenden Prozesse daher beliebig im Netz umverteilen, ohne daß Änderungen am Applikationscode notwendig werden.

Für Klassen, deren Exemplare im Sinne des `Materialkoordinators` nach einer Integration mit dem Object Request Broker Dienste im Netz zur Verfügung stellen sollen, müssen Schnittstellen in einer abstrakten Beschreibungssprache, der *Interface Definiton Language* (IDL), definiert werden. Mit Hilfe eines IDL-Übersetzers werden aus diesen abstrakten Schnittstellen sogenannte *Interface-Klassen* in einer konkreten Programmiersprachensyntax generiert. Exemplare solcher Interface-Klassen entsprechen den bereits erwähnten Proxyobjekten. Der vom IDL-Übersetzer generierte Code besteht im wesentlichen aus der Zusammenstellung von Requests, ihrem Versand an den Object Request Broker, sowie der Entgegennahme der Resultate. Exemplare sind nicht *per se* entweder Clientobjekte oder Serverobjekte, sondern können je nach Kontext Dienste in Anspruch nehmen oder anbieten. Die vom IDL-Übersetzer generierten *Implementationsklassen* sind daher in der Regel von den *Interfaceklassen* abgeleitet und haben somit auch Zugriff auf den "Clientcode". Implementationsklassen kapseln Einzelheiten der Inter-Prozeß-Kommunikation und dienen als (abstrakte) Basis für diejenigen Klassen, die die eigentliche Dienstleistungssemantik, etwa die eines `Materialkoordinators`, implementieren.

Vor dem Hintergrund der im letzten Abschnitt geführten Diskussion wirkt sich die Integration einer verteilten Werkzeugumgebung in ein CORBA-basiertes System insbesondere dadurch aus, daß manuell implementierte Klassen wie `tCoordinationClient`, die ja gewissermaßen mit primitiven Mitteln die beschriebenen ORB-Techniken *simulieren*, beim Entwurf der verteilten Softwarearchitektur vollständig entfallen. Fachliche Entwurfskomponenten wie der `Materialkoordinator` können nun prinzipiell 1:1 in das technische Klassenmodell überführt werden.

7. Das *SchwarzeBrett* - prototypisches Werkzeug in einer verteilten Werkzeugumgebung mit objektorientiertem Datenbankanschluß

Im Mittelpunkt dieses Kapitels stehen der Entwurf und die Implementierung eines prototypischen Werkzeuges, das wir in eine verteilte Werkzeugumgebung mit objektorientiertem Datenbankanschluß integriert haben. Für die Entwicklung dieses Werkzeuges gibt es zumindest zwei gute Gründe. Zum einen sehen wir darin die Chance, Lesern, die selbst Werkzeuge konstruieren und in das Versorgungsmodell einbinden möchten, eine beispielhafte Orientierungsmöglichkeit an die Hand zu geben. Zum anderen ermöglicht sie uns, das im Rahmen dieser Arbeit entwickelte Materialversorgungsmodell im Hinblick auf seine Tragfähigkeit zu validieren. Wir werden daher in den Schlußbemerkungen zu der vorliegenden Arbeit unter anderem über unsere Erfahrungen bei der Konstruktion des Werkzeuges berichten und sie einer kritischen Beurteilung unterziehen.

7.1. Das Werkzeug

Das *SchwarzeBrett* ist ein Browserwerkzeug mit integriertem Editor zur Bearbeitung von Notizen. Über seine grafische Benutzungsoberfläche können Notizen mit dem Browser durchblättert und ausgewählt, durch einfache Mausmanipulation in den Editor geladen, dort verändert und schließlich gespeichert werden. Notizen können mit dem Editor neu erzeugt werden und sind mit ihrem ersten Speichern unmittelbar in allen Browserwerkzeugen der verteilten Umgebung sichtbar.

Das *SchwarzeBrett* scheint uns in mehrfacher Hinsicht als Anwendungsbeispiel geeignet. Notizen werden erfahrungsgemäß nur dann an ein Schwarzes Brett geheftet, wenn sie einer breiten Leserschaft zugänglich gemacht werden sollen. Innerhalb eines technischen Systems kann das *SchwarzeBrett* also überhaupt nur dann zu einem sinnvollen Einsatz kommen, wenn es in eine verteilte Werkzeugumgebung eingebettet ist. Wer schon öfters vor einem Schwarzen Brett mit völlig überalteten Anschlägen gestanden hat, wird darüberhinaus leicht zu überzeugen sein, das ein entsprechendes Softwarewerkzeug die Aktualität seiner Browseranzeige über die Teilnahme an einem kooperativen Benachrichtigungsprotokoll sicherstellen sollte. Nicht zuletzt kann man an dem Browser-Editor-Paar deutlich machen, warum Notizen in einer verteilten Werkzeugumgebung als Originale oder als Kopien zur Verfügung gestellt werden.

In Anlehnung an das bereits vorgestellte Architekturmuster für Softwarewerkzeuge (vgl. Kapitel 2.2.2.) haben wir Browser und Editor jeweils in Funktions- und Interaktionskomponente strukturiert und ihr Zusammenspiel mit den Notizen über werkzeugspezifische Aspekte modelliert. Abbildung 17 greift die geschilderten Zusammenhänge grafisch auf. Einzelne Komponenten werden dabei mit ihren technischen Klassennamen bezeichnet.

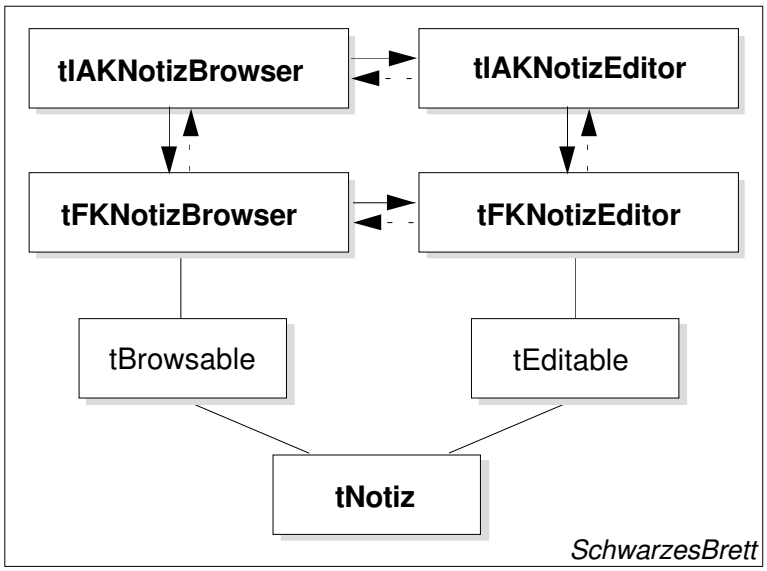


Abb. 17: Das Zusammenspiel von Browser, Editor und Notizen über werkzeugspezifische Aspekte.

Wir werden im folgenden schrittweise die Konstruktion von Browser und Editor nachvollziehen und am Beispiel der *Erzeugung einer neuen Notiz* zeigen, wie sich die Anbindung an das Materialversorgungsmodell auf die Entwicklung von Werkzeug- und Materialklassen auswirkt. Die Erläuterungen werden jeweils durch passende Ausschnitte aus den Klassenimplementationen veranschaulicht.[26]

7.2. Systemeinstieg über die Einsprungroutine `main`

In C++ ist die Routine `main()` Einsprungspunkt für eine Anwendung. In unserem Fall enthält sie so gut wie keinen prozeduralen Code, sondern erzeugt und startet lediglich ein Anwendungsobjekt der Klasse `tUmgebung`, die wir als softwaretechnische Abstraktion der Werkzeugumgebung verstehen. Dem Anwendungsobjekt obliegt die Erzeugung aller weiteren benötigten Objekte. Mit dem Verlassen der Methode `Run()` wird gleichzeitig auch die Anwendung beendet.

```
void main( int argc, char* argv[] )
{
      tUmgebung umgebung( argc, argv );
      umgebung.Run ();
};
```

[26] Wir werden sowohl die Interaktionskomponenten von Browser und Editor als auch ihre werkzeugspezifischen Aspekte im folgenden vernachlässigen, da sie für die Einbindung der Werkzeuge in das Materialversorgungsmodell keine relevante Rolle spielen.

7.3. Die Anwendungsklasse `tUmgebung`

Die Anwendungsklasse `tUmgebung` wird benötigt, um das erste und einzige Anwendungsobjekt zu liefern. Bei seiner Instantiierung in der Einsprungroutine `main()` wird die Initialisierungsmethode `Init()` gerufen, die ein neues Browserobjekt erzeugt.

```
void tUmgebung :: Init( int argc, char* argv[], ... )
{
      // ... alle Notizen, die in der Umgebung verfügbar sind
      tRequest request( '*', "tNotiz" );

      pFKNotizBrowser = new tFKNotizBrowser(this, pManager, request);
}
```

Das Browserobjekt wird mit einem Verweis auf den Materialverwalter der Werkzeugumgebung und einer abstrakten Materialbeschreibung initialisiert. Die folgenden Abschnitte werden zeigen, wofür das Browserobjekt diese Informationen benötigt.

7.4. Die Klasse `tEnvironment`

Neben der Funktionalität zum Öffnen und Schließen des zugrundeliegenden grafischen Systems erbt unsere Anwendungsklasse von `tEnvironment`[27] die Methode `Run()`. Sie startet eine Anwendung und verteilt bis zu ihrer Beendigung die Kontrolle gleichmäßig zwischen dem über Benutzereingaben gesteuerten Fenstersystem und einer *Polling*-Komponente (*pCoordinationClient*), die eintreffende Ereignisse des im Netz liegenden Materialkoordinators empfängt. `Run()` wird unmittelbar nach der Instantiierung des Anwendungsobjekts in der Einsprungroutine `main()` angestoßen (vgl. Kapitel 7.2.).

Neben der Pollingkomponente sind ein Materialversorgerautomat, der die Kommunikation mit der angeschlossenen objektorientierten Datenbank regelt (*pSupplier*) und der Materialverwalter der Werkzeugumgebung (*pManager*) in der Klasse `tEnvironment` verankert.

```
void tEnvironment :: Init( int argc, char* argv[], ... )
{
      ...
      pCoordinationClient = new tCoordinationClient;
      pSupplier = new tOntosSupplier;
      pManager = new tManager;
      ...
}
```

Aus Werkzeugsicht ist ausschließlich der Materialverwalter von Interesse. Er wird sowohl Browser- als auch Editorobjekten bei ihrer Erzeugung über einen entsprechenden Verweis bekannt gemacht (vgl. Kapitel 7.3. und 7.5.).

7.5. Die Werkzeugklasse `tFKNotizBrowser`

Bei ihrer Instantiierung in der `Init()`-Methode des Anwendungsobjekts (vgl. Kapitel 7.3.) wird der Konstruktor des Browserwerkzeuges gerufen.

[27] vgl. [Rie93a].

```
tFKNotizBrowser :: tFKNotizBrowser    (    tNotifier* aParent_p,
                                           tManager* aManager_p,
                                           tRequest aRequest)
          : pFKNotizEditor( NULL ), pDepot( NULL ), ...
{
      if (! (pDepot = pManager -> Query( this, aRequest, cReadLock )))
        pDepot = new tDepot;
      ...
}
```

Browserobjekte initialisieren im Konstruktor ihr lokales Werkzeugmagazin (*pDepot*) mit Notizen - in unserem Beispiel mit allen, die innerhalb der verteilten Werkzeugumgebung verfügbar sind. Ein Browser stellt zu diesem Zweck eine entsprechende Materialanforderung an den Materialverwalter (*pManager*). Da die Notizen im Browser nur angezeigt, nicht aber verändert werden, fordert er sie als Kopien an (`cReadLock`). Sie stehen damit weiterhin als potentielle Originale - etwa für einen Editor (vgl. Kapitel 7.6.) - in der verteilten Werkzeugumgebung zur Verfügung. Falls der von dem Materialverwalter gelieferte Behälter leer ist, also keine Notizen in der Umgebung vorhanden sind, erzeugt der Browser sich ein leeres Werkzeugmagazin, in das nun neue Notizen eingefügt werden können.

```
void tFKNotizBrowser :: Neu ()
{
      pFKNotizEditor = new tFKNotizEditor( this, pManager );
}
```

Neue Notizen werden mit Hilfe des Editors angelegt, der die notwendige Funktionalität zum Erzeugen, Editieren und Speichern von Notizen bereitstellt. In seiner Methode `Neu()` erzeugt der Browser daher lediglich ein neues Objekt der Klasse `tFKNotizEditor`. Sobald der Editor die neue Notiz speichert (vgl. Kapitel 7.6.), erreicht den Browser ein entsprechendes Ereignis (`cEventInsert`), auf das er in seiner `Notify()`-Methode geeignet reagieren kann.

```
void tFKNotizBrowser :: Notify( tNotifier* aNotifier_p, tEvent anEvent )
{
      ...
  case cEventInsert:    pInsertDepot = pManager -> Query(this,
                                                         specification,
                                                         cReadLock );
                        ...
                        if (pManageable = (*pInsertDepot)())
                        {
                          // Speicherkopie der Notiz anlegen und
                          // einfügen
                          pDepot -> Insert( pManageable->MakeManageable());
                             ...
                        }
                        delete pInsertDepot;
                        ...
}
```

In unserem Beispiel läßt sich der Browser durch eine erneute Anfrage beim Materialverwalter einen temporären Behälter (*pInsertDepot*) mit der neuen Notiz (*pManageable*) liefern, deren Spezifikation er vorher beim Materialverwalter erfragen kann. Bevor er den temporären Behälter löscht, legt sich der Browser eine Speicherkopie der Notiz an und fügt sie in sein lokales Werkzeugmagazin ein, dessen Repräsentation an der Benutzungsoberfläche von der zuständigen Interaktionskomponente (`tIAKNotizBrowser`) entsprechend aktualisiert werden muß.

Wir möchten darauf hinweisen, daß unser Beispielbrowser nicht notwendigerweise das einzige Objekt ist, dem das Einfügen der neuen Notiz gemeldet wird. Im Sinne eines

kooperativen Benachrichtigungsmechanismus werden vielmehr *alle* Browser, die in der verteilten Werkzeugumgebung zu diesem Zeitpunkt bereits gestartet worden sind, über das Ereignis informiert[28] .

7.6. Die Werkzeugklasse `tFKNotizEditor`

Die Funktionskomponente `tFKNotizEditor` stellt die gesamte Funktionalität bereit, die zur Bearbeitung von Notizen benötigt wird.

```
void tFKNotizEditor :: Speichern ()
{
      tManageable* pManageable = pEditable -> SafeCast( "tManageable" );

      // die neue Notiz bei dem Materialverwalter der Umgebung einfügen
      pManager -> Insert( this, pManageable );
      ...
}
```

Bei seiner Instantiierung in der `Neu()`-Methode des Browsers wird das Editorobjekt mit einem Verweis auf den Materialverwalter der Werkzeugumgebung (*pManager*) initialisiert, die es beispielsweise dazu benutzen kann, eine neu angelegte Notiz einzufügen. Wir weisen darauf hin, daß der Editor sich nicht darum kümmern muß, *ob* die Notiz darüberhinaus in einem externen Datenbestand gespeichert wird, und falls ja, *in welchem*. Nach dem Einfügen steht die Notiz allen Werkzeugen der verteilten Umgebung zunächst als potentielle Kopie zur Verfügung. Erst wenn ein Anwender den Editor verläßt und der Programmfluß in den Destruktor des Editorobjekts gelangt, wird die Notiz beim Materialverwalter als Original freigegeben. Ein entsprechendes Ereignis - `cEventCheckin` - wird an alle bereits gestarteten Werkzeuge der verteilten Umgebung propagiert.

```
tFKNotizEditor :: ~tFKNotizEditor ()
{
      tManageable* pManageable = pEditable -> SafeCast( "tManageable" );

      pManager -> Checkin( this, pManageable );
      ...
}
```

Während Editorobjekte ihre Notizen unter dem werkzeugspezifischen Aspekt `tEditable` bearbeiten, kann der Materialverwalter bekanntlich nur über den Aspekt `tManageable` auf sie zugreifen (vgl. Kapitel 6.1.3.3.). Die von einem Editorobjekt gerade bearbeitete Notiz (*pEditable*) muß daher typsicher auf den richtigen Aspekt konvertiert werden, bevor sie dem Materialverwalter übergeben werden kann.

[28] Genaugenommen werden natürlich nicht nur die Browser, sondern auch alle anderen Werkzeuge, die derzeit aktiv sind und das kooperative Benachrichtigungsprotokoll implementieren, von dem Ereignis unterrichtet.

7.7. Die Materialklasse `tNotiz`

Als letztes Glied der Kette betrachten wir die Materialklasse `tNotiz`. Uns geht es insbesondere darum, exemplarisch zu demonstrieren, was bei der Entwicklung einer Materialklasse konkret zu berücksichtigen ist, wenn ihre Exemplare über einen Materialverwalter administriert und darüberhinaus in der objektorientierten Datenbank ONTOS gespeichert werden sollen. Wir werden daher die *fachliche* Seite von Notizen in unseren Ausführungen vernachlässigen.

Ein Blick auf die Klassenschnittstelle von `tNotiz` läßt sich vor diesem Hintergrund auf drei wesentliche Protokolle einschränken: `Object`, `tStorable`, und `tManageable`. Wie die Materialklasse diese Protokolle implementiert, zeigen die folgenden Abschnitte.

```
class tNotiz :     public Object,
                   public tStorable,
                   public tManageable
{
  private:
      char* name;
      // ... fachliche Attribute

  public:

      // Object - Protokoll
      tNotiz   ( APL* theAPL );
      virtual Type* getDirectType ();
      virtual void* startAdress  ();
      virtual void Destroy   ( OC_Boolean aborted = FALSE );


      // tStorable - Protokoll
      tState Store ();
      tState Delete ();

      // tManageable - Protokoll
      virtual tString   GetName       () const { return tString(name); };
      virtual tString   GetTypeName   () const { return tString("tNotiz");
};

      virtual tManageable* MakeManageable  () const;
      virtual void      MakeAssignment     (tManageable* aManageable_p);
      virtual void      SafeCast           (tString aTypeName) const;
      ...
};
```

7.7.1. Implementierung des `Object`-Protokolls

Materialklassen, deren Exemplare in der objektorientierten Datenbank ONTOS gespeichert werden sollen, müssen die hierfür notwendige, datenbankspezifische Funktionalität bekanntlich von der Klasse `Object` erben (vgl. Kapitel 6.1.7.).

```
// Aktivierungskonstruktor
tNotiz :: tNotiz ( APL* theAPL ) : Object( theAPL )
{
}
```

Sie verpflichten sich damit gleichzeitig zur Reimplementierung bestimmter virtueller Methoden ihrer Basisklasse. Darüberhinaus müssen sie einen im ONTOS Sprachgebrauch als *activation constructor* bezeichneten Konstruktor implementieren, der bei der Aktivierung von Materialexemplaren aus der Datenbank in den Client-Cache (vgl. Kapitel 5.2.) benötigt wird. Im Standardfall kann er mit einem leeren Konstruktorkörper implementiert werden.

```
// beliebiger Klassenkonstruktor
tNotiz :: tNotiz ()
{
      initDirectType( getDirectType() );
      ...
}
```

Ein weitere Besonderheit, die *alle* anderen Konstruktoren betrifft, ist die Methode `initDirectType()`. Sie muß von der Materialklasse `tNotiz` zwar nicht überschrieben, dafür aber innerhalb sämtlicher Klassenkonstruktoren aufgerufen werden und liefert ONTOS die nötigen Metainformationen, die es braucht, um ein innerhalb der Anwendung erzeugtes Materialexemplar in der Datenbank speichern und später auch wieder daraus rekonstruieren zu können.

```
Type* tNotiz :: getDirectType ()
{
      // das Metaobjekt "tNotiz" aus der Datenbank laden
      return (Type *) OC_lookup( "tNotiz" );
}
```

Softwaretechnisch wird von diesen Metainformationen in der ONTOS-Klasse `Type` abstrahiert. Notizspezifische Metainformationen werden als Objekt der Klasse `Type` verwaltet, das über den Namen der Materialklasse ("tNotiz") in der Datenbank referenziert werden kann und als Rückgabewert von der zu reimplementierenden Methode `getDirectType()` geliefert werden muß.

```
void* tNotiz :: startAddress ()
{
      return this;
}
```

Mehrfaches Hineinvererben von Aspekten in Materialklassen ist gewissermaßen die Regel, kann von ONTOS aber nur dann bewältigt werden, wenn die Methode `startAddress()` reimplementiert wird.

```
void tNotiz :: Destroy( OC_Boolean aborted )
{
      ...      // hier steht der normale Destruktorcode
               // ( Deallokation dynamisch angelegten
               // Speichers )

      if (aborted)
        Object :: Destroy( aborted );
}
```

Nicht zuletzt muß die Methode `Destroy()` überschrieben werden, die ONTOS im Rahmen seines *Exception Handling* benutzt, um den dynamisch alloziierten Speicher eines (gegebenenfalls hierarchischen) Objektes abzubauen, ohne das Objekt selbst dabei zu zerstören. Der Destruktorcode vereinfacht sich nun zu einem Aufruf der Methode `Destroy()`.

```
tNotiz :: ~tNotiz ()
{
      Destroy( FALSE );
}
```

7.7.2. Implementierung des `tStorable`-Protokolls

Damit Notizexemplare von einem datenbankspezifischen Automaten aus einer ONTOS-Datenbank in die Umgebung geladen bzw. aus der Umgebung zurück in die Datenbank überführt werden können, implementiert `tNotiz` die Methoden `Store()` und `Delete()` (vgl. Kapitel 6.1.7.), deren Code sie praktisch vollständig von der Klasse `Object` übernehmen kann.

```
tState tNotiz :: Store ()
{
      putObject();           // Object::putObject
      ...
}


tState tNotiz :: Delete ()
{
      deleteObject();        // Object::deleteObject
      ...
}
```

7.7.3. Implementierung des `tManageable`-Protokolls

Welche Rolle das `tManageable`-Protokoll innerhalb des Materialversorgungsmodells für verteilte Werkzeugumgebungen spielt, haben wir bereits ausführlich in Kapitel 6.1.3. deutlich gemacht. An dieser Stelle sollen als Ergänzung Implementierungsvorschläge für die Methoden `MakeManageable()`, `MakeAssignment()` und `SafeCast()` unterbreitet werden.

```
// ... eine Speicherkopie des Notizexemplars abstrakt erzeugen
tManageable* tNotiz :: MakeManageable () const
{
      return (tManageable *) new tNotiz(*this);
}


// ... ein anderes Notizexemplar abstrakt zuweisen
void tNotiz :: MakeAssignment( tManageable* aManageable_p )
{
      tNotiz* pNotiz = (tNotiz *) aManageable_p -> SafeCast("tNotiz");
      if (pNotiz)
      {
          strcpy( name, pNotiz -> GetName());
          // ... fachliche Attribute kopieren
      } ...
}


// ... ein Notizexemplar typsicher zwischen Aspekten
// und auf Aspekte konvertieren
void* tNotiz :: SafeCast( tString aTypeName ) const
{
      if (aTypeName == "tManageable")
          return (void *) (tManageable *) this;
      ...

      else if (aTypeName == "tNotiz")
          return (void *) this;
      ...

      else return NULL;
}
```

8. Schlußbemerkungen

Wir möchten den Raum dieser Schlußbemerkungen für eine kurze Zusammenfassung und Einschätzung der Erfahrungen nutzen, die wir sowohl bei der *Umsetzung* als auch im *Umgang* mit dem entwickelten Materialversorgungsmodell gesammelt haben.

Bei der Konstruktion eines prototypischen Werkzeuges und seiner Integration in eine verteilte Werkzeugumgebung mit objektorientierter Datenbankanbindung hat sich das Versorgungskonzept als solches aus unserer Sicht als tragfähig erwiesen.[29] Der Implementierungsaufwand im Bereich Materialbeschaffung und -verwaltung wird insbesondere für *Werkzeugklassen*, die alternativ ihre eigene Materialverwaltung implementieren müßten, deutlich geringer. Dies gilt um so mehr natürlich in Hinblick auf die in der Materialverwaltung gekapselte Verteilungs- und Benachrichtigungssemantik.

Weniger gute Erfahrungen haben wir bei der Integration unserer *Materialklassen* mit dem objektorientierten Datenbanksystem ONTOS gemacht. Materialien müssen neben ihren fachlichen Aspekten eine erstaunliche Vielzahl an Methoden und Methodenaufrufen implementieren, ohne die ONTOS nicht in der Lage wäre, ein einfaches, unvernetztes Materialexemplar in seinen Datenbanken zu verwalten. Der Hersteller hätte hier mittels Sourcecode-Generator jedenfalls für Standardfälle Abhilfe schaffen können.

Als weitaus schwerwiegender empfinden wir aber, daß Materialklassen gewissermaßen "von unten" verdrahtet werden, indem sie nicht nur systemplattformabhängige Informationen, sondern gegebenenfalls auch das komplette Speichermanagement für referenzierte Objekte implementieren müssen und damit an der Kontrolle der bearbeitenden Werkzeuge vorbeilaufen.

Bei der Implementierung eines ONTOS-spezifischen Datenbankautomaten haben wir uns insgesamt nur sehr wenige objektorientierte Datenbankkonzepte zunutze machen können. Die in Herstellerkreisen vielgepriesenen *optimistischen Zugriffsprotokolle* ermöglichen zwar prinzipiell einen parallelen lesenden und schreibenden Zugriff auf Daten, führen aber in der Praxis zu häufig zum Abbruch von Transaktionen, weil konkurrierende Zugriffe nicht serialisiert werden können. Für eine verteilte Werkzeugumgebung zur Unterstützung hochgradig *kooperativer* Arbeitsprozesse haben sich vielmehr fachlich motivierte, von technischen Gesichtspunkten wie Serialisierbarkeit unabhängige Zugriffsprotokolle als brauchbar erwiesen. Was wir aus der Sicht von Werkzeugen benötigen, sind im buchstäblichen Sinne *objekt*orientierte Sperrmechanismen (*Checkin/Checkout*), die auf der Ebene einzelner Materialexemplare, also noch weitaus granularer als Transaktionen, operieren.

Im Unterschied zu den ereignisgesteuerten Implementierungen anderer Datenbankhersteller macht darüberhinaus auch der pollingbasierte *Change-Update* Mechanismus von ONTOS seinem Namen nicht gerade alle Ehre. Durch die Implementierung eines eigenen, von eventuellen Datenbankanschlüssen unabhängigen Benachrichtigungsprotokolls sind wir allerdings von den damit verbundenen Widrigkeiten unberührt geblieben.

Eine weitere Erprobung und Konsolidierung der im Rahmen dieser Arbeit vorgestellten Konzepte und Modelle erachten wir in jedem Fall als lohnenswert. Wir denken dabei vor allem in die Richtung eines datenbankunabhängigen Austausches von Materialien *zwischen*

[29] Wir sind uns bewußt, daß eine aussagekräftige Modellvalidierung im Rahmen dieser Arbeit nicht geleistet werden kann und sich überhaupt erst nach intensiver Rückkopplung mit Benutzenden auf festen Boden stellen läßt. Als nächstes Ziel haben wir daher vor Augen, die Klassen unseres technischen Modells in die bereits vorhandene FIAK-Klassenbibliothek (vgl. [Rie93a]) zu integrieren, die von StudentInnen des Arbeitsbereiches standardmäßig bei der Konstruktion von Werkzeugen verwendet wird.

verteilten Werkzeugumgebungen und an den Umgang mit komplexen, hypertextartig verknüpften Materialobjekten.

Literaturverzeichnis

[Bar93] D. Bartels: *Eventdriven Concepts for object oriented Client/Server Database Systems*. Papier zur "ObjectWorld" Konferenz, Frankfurt, 1993.

[Bey93] T. Beyer: *CORBA - OMG-Standard für verteilte Objekte*. iX-Magazin, Februar 1993, S. 24-33.

[Bud90] R. Budde, H. Züllighoven: *Software-Werkzeuge in einer Programmierwerkstatt*. Berichte der Gesellschaft für Mathematik und Datenverarbeitung, Nr. 182, Oldenbourg, München, Wien, 1990.

[Bud91] R. Budde, H. Züllighoven: *Software Tools in a Programming Workshop*. In: C. Floyd et al. (Hrsg.) Software Development and Reality Construction, Springer, Berlin, Heidelberg, New York, 1991, S. 252-268.

[Bud92] R. Budde , G. Gryczan, K.-H. Sylla, H. Züllighoven: *Construction of Tools in the "Tools and Material" Metaphor*. Unveröffentlichtes Papier, Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1992.

[Cat91] R.G.G. Cattell: *Object Data Management - Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, Reading, Mass., 1991.

[Cur91] D.A. Curry: *Using C on the Unix System - A Guide to System Programming*. O'Reilly & Associates, Sebastopol, Ca., 1991.

[Flo91] C. Floyd: *Einführung in die Softwaretechnik*. Skriptum zur gleichnamigen Vorlesung, Fachbereich Informatik, Universität Hamburg, 1991.

[Gam92] E. Gamma: *Objektorientierte Softwareentwicklung am Beispiel von ET++*. Design-Muster, Klassenbibliothek, Werkzeuge. Springer, Berlin, Heidelberg, New York, 1992.

[Gra81] Gray, J.N.: *The Transaction Concept - Virtues and Limitations*. Proceedings of the Seventh International Conference on Very Large Databases, Cannes, 1981.

[Gry92] G. Gryczan, H. Züllighoven: *Objektorientierte Systementwicklung. Leitbilder und Entwicklungsdokumente*. In: Informatik-Spektrum , Band 15, Heft 5, Oktober 1992, S. 264-272.

[Gut93] M. Guttman, J.A. King: *A Methodology for Developing Distributed Applications*. Object Magazine, Januar/Februar 1993, S. 55-59.

[Heu92] A. Heuer: *Objektorientierte Datenbanken*. Addison-Wesley, Bonn, München, Paris, 1992.

[Hor93] C. Horn: *Standardizing your Object Interface. What will the ORB technology be like in practice?* Object Magazine, September/Oktober 1993, S. 56-64.

[Kil93] K. Kilberth, G. Gryczan, H. Züllighoven: *Objektorientierte Anwendungsentwicklung. Konzepte, Strategien, Erfahrungen*. Vieweg, Braunschweig, Wiesbaden, 1993.

[Mow93] T. Mowbray, T. Brando: *Interoperability and CORBA-based Open Systems*. Object Magazine, September/Oktober 1993, S. 50-54.

[ONT92] *ONTOS DB 2.2 Developers Guide*. Handbuch. ONTOS Inc., Burlington, Mass., 1992.

[Rie93a] D. Riehle: *Dokumentation zur FIAK-Bibliothek*. Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1993.

[Rie93b] D. Riehle: *Interaktionstypen Dokumentation zur IATMotif v1.1. Bibliothek*. Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1993.

[Str93] H. Strickland: *ODMG-93. The Object Database Standard for C++*. C++ Report, Oktober 1993, S. 45-52.

[Sul92] K.J. Sullivan, D. Notkin: *Reconciling Environment Integration and Software Evolution*. ACM Transactions on Software Engineering and Methodolgy. Vol. 1, No. 3, 1992, S. 229-268.

[Sun92] Sun Microsystems: *Solaris 1.1 Answer Book*. Sun Microsystems Computer Corporation, Mountain View, Ca., 1992.

[Wir90] R.J. Wirfs-Brock, B. Wilkerson, L. Wiener: *Designing Object-Oriented Software*. Prentice Hall, New Jersey,1990.