Objektorientierung und Persistenz:

Ein Anforderungskatalog an die Verwaltung von Materialien in Einzelplatz-Arbeitsumgebungen aus fachlicher und technischer Sicht

Studienarbeit
am
Fachbereich Informatik,
Arbeitsbereich Softwaretechnik,
Universität Hamburg

Betreuerin:
Dr. Ingrid Wetzel

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

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

26. Juli 1996

Zusammenfassung

Zur Zeit steht dem *qualifizierten Benutzer* von Anwendungssystemen, die nach der Methode *WAM* entwickelt wurden, abgesehen von den Arbeiten [Wul+94] und [Gry95] keine fachlich motivierten Konzepte für die Verwaltung persistenter Materialien zur Verfügung. Materialien stehen nur transient zur Laufzeit innerhalb eines Arbeitsumgebungsprozesses bereit.

Diese Arbeit stellt vergleichend mehrere persistente Spracherweiterungen gegenüber und stellt Forderungen für ein technisches Konzept eines *Objektmanagementsystems* (*OMeS*) auf. Dabei wird die Problematik der komplexen Vernetzung und damit die der Identitäten und der Referenzen von Objekten beleuchtet.

Vergleichend werden die Materialversorgungskonzepte aus [Wul+94] und [Gry95] und ein Konzept von [Kil+93] gegenübergestellt. Daraus werden Forderungen für ein rein fachlich motiviertes *Materialmanagementsystem* (*MMeS*) abgeleitet und die Grundlage für einen integrierten Ansatz von OMeS und MMeS zu einem *Persistenten objektorientierten Materialmanagementsystem* (*PooMMeS*) geschaffen.

„Fakten, Fakten, Fakten...
und an die Leser denken."
(Helmut Markwort, Chefredakteur FOCUS)

Danksagung

Wir danken allen, die uns bei der Erstellung dieser Studienarbeit unterstützt haben. Unser besonderer Dank gilt unserer Betreuerin Dr. Ingrid Wetzel, Eckhard Stolberg für seine Hilfe bei unseren TeX- und UNIX-Fragen, Christian Langmann und Dr. Florian Matthes für ihren gut gemeinten Ratschlag, ObjectStore zu verwenden, und Sheetal V. Kakkad von der University of Texas in Austin/Texas für ihre Hinweise aus der Ferne.

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Hintergrund 3
1.2 Methodenrahmen STEPS 3
1.3 Methode WAM 4
1.4 Aufgabenstellung 5
1.5 Übersicht 6
1.6 Schreibweisen 7
1.6.1 Textuelle Konventionen 7
1.6.2 Grafische Konventionen 8

2 Begriffe **9**
2.1 Das Objekt 9
2.2 Die Identität 9
2.3 Die Vererbung und Beziehungen zwischen Objekten 10
2.4 Die Referenz 10
2.5 Die indirekte Referenz (smart pointer) 11
2.6 Die Lebensdauer 12
2.7 Die Repräsentation eines Objektes im transienten Speicher 12

3 Forderungen an ein Materialmanagementsystem MMeS **15**
3.1 Einleitung 15
3.2 Konzeptionelle Anbindung einer relationalen Datenbank ([Kil+93]) 15
3.2.1 Einleitung 15
3.2.2 Der Datenbankautomat 16
3.3 Das Materialversorgungskonzept ([Wul+94]) 17
3.3.1 Einleitung 17
3.3.2 Der Materialverwalter 18
3.3.3 Der Materialversorger 19
3.3.4 Der Materialkoordinator 19
3.4 Die Materialverwaltung ([Gry95]) 20
3.4.1 Einleitung 20
3.4.2 Wesentliche Unterschiede zum Materialversorgungskonzept ([Wul+94]) 20

3.5 Kritische Würdigung 21
3.6 Forderungskatalog 22

4 Forderungen an ein Objektmanagementsystem OMeS 25
4.1 Motivation 25
4.2 Einleitung 25
4.3 Darstellung persistenter C++-Erweiterungen 28
4.3.1 Arjuna 28
4.3.2 E 31
4.3.3 O++ 36
4.3.4 Texas 40
4.4 Übersicht der persistenten C++-Erweiterungen 50
4.5 Kritische Würdigung und Forderungskatalog 54
4.5.1 Die Problematik der Bewertung 54
4.5.2 Der Forderungskatalog 57
4.5.3 Die Bewertung der Ansätze 66

5 Standortbestimmung und Ausblick 69
5.1 Standortbestimmung 69
5.2 Richtungsbestimmung für die weiterführende Arbeit 70

A Entwicklungsprozeß unserer Studienarbeit 73
A.1 Unser Hintergrund 73
A.2 Die Zusammenarbeit 74

Kapitel 1

Einleitung

1.1 Hintergrund

Das Softwaretechnikprojekt am Fachbereich Informatik der Universität Hamburg befaßt sich mit dem Entwurf und der Konstruktion interaktiver verteilter Softwaresysteme. Unter Einsatz der Methode *WAM*[1] im Methodenrahmen[2] *STEPS*[3] fokusiert der evolutionäre, partizipative Softwareentwicklungsprozeß des diesjährigen Projektes auf die Erstellung eines Systems zur computergestützten Koordination universitärer Lehrveranstaltungen.

1.2 Methodenrahmen STEPS

Der Methodenrahmen STEPS begreift die Softwareentwicklung als ein zyklisches Vorgehen. Der Lern- und Kommunikationsprozeß zwischen Anwendern und Entwicklern steht im Mittelpunkt und wird unterstützt (vgl. [Flo91]).

Da STEPS keinen umfassenden Satz von Werkzeugen und Darstellungsmitteln zur Entwicklung von Software bereitstellt, muß der Methodenrahmen durch eine zur Sichtweise von STEPS passende Methode gefüllt werden. Eine solche ist die im folgenden vorgestellte Methode WAM.

[1] WAM ist eine an der Universität Hamburg entstandene Ausprägung einer objektorientierten Sichtweise, siehe [Kil+93]

[2] vgl. [Gry95]

[3] STEPS (**S**oftware**t**echnik für **E**volutionäre **P**artizipative **S**ystementwicklung) ist ein an der Technischen Universität Berlin entwickelter theoretischer und methodischer Ansatz der Softwaretechnik, siehe [Flo91]

1.3 Methode WAM

Der Methode WAM liegt die *Entwurfsmetapher* „Umgang mit Werkzeug und Material" zugrunde. WAM verfolgt das Ziel, einen fachlichen Entwurf ohne Modellbruch in einen technischen zu überführen. Dabei bildet die Entwurfsmetapher von den gewohnten Gegenständen der Arbeitsdomäne auf die Komponenten des zu entwerfenden Anwendungssystems ab. Als Leitbild dient dabei der Arbeitsplatz für qualifizierte menschliche Tätigkeit ([Kil+93]).

WAM steht sowohl für

- die Entwurfsmetapher **W**erkzeug **A**utomat **M**aterial als auch
- die Konstruktionsmetapher **W**erkzeug **A**spekt **M**aterial,

so daß in letzter Zeit auch von WAAM gesprochen wird.

Die Begriffe lassen sich nach [Kil+93] und [Gry95] wie folgt fachlich und technisch definieren:

Werkzeug

- fachlich: Ein Werkzeug ist ein Arbeitsmittel, mit dem Arbeitsgegenstände (Materialien) interaktiv bearbeitet werden. Es verändert den Zustand der Materialien und stellt diesen dar. Es ist somit eine aktive Komponente.
- technisch: Ein Werkzeug spaltet sich in eine Funktionskomponente, in der die Funktionalität modelliert ist, und in eine oder mehrere Interaktionskomponenten, die die Präsentation bereitstellen, auf.

Automat

- fachlich: Ein Automat ist ein Arbeitsmittel, das die Arbeitsgegenstände (Materialien) nach einem fest vorgegebenen Algorithmus, der über einen längeren Zeitraum ablaufen kann, bearbeitet. Dabei verändert er den Zustand der Materialien und ist somit wie ein Werkzeug eine aktive Komponente.
- technisch: Ein Automat spaltet sich wie ein Werkzeug in eine Funktionskomponente und eine oder mehrere Interaktionskomponenten, die dem Anwender Parametereinstellungen ermöglichen, auf.

Material

- fachlich: Ein Material ist ein Arbeitsgegenstand, dessen Zustand durch Werkzeuge und Automaten verändert und sondiert werden kann. Es ist somit eine passive Komponente.

- technisch: Materialklassen stellen Operationen zum Verändern und Sondieren des Zustandes von Materialinstanzen bereit.

Aspekt

- fachlich: Aspekte definieren die Schnittstelle zwischen Materialien auf der einen Seite und Werkzeugen und Automaten auf der anderen. Sie geben zum einen an, wie Materialien von Werkzeugen und Automaten veränderbar und sondierbar sind und zum anderen, wie Werkzeuge und Automaten Materialien benutzen.

- technisch: Aspektklassen entkoppeln Werkzeug- und Automatenklassen von Materialklassen. Werkzeugklassen benutzen Aspektklassen, die wiederum Oberklassen von Materialklassen bilden.

1.4 Aufgabenstellung

Der Arbeitsbereich Softwaretechnik hat abgesehen von den Arbeiten [Wul+94] und [Gry95] keine fachlich motivierten Konzepte für die Verwaltung persistenter Materialien. Materialien stehen demnach nur transient zur Laufzeit innerhalb von Arbeitsumgebungsprozessen zur Verfügung.

Das Ziel dieser Arbeit soll es deshalb sein, die Grundlage für eine konzeptionelle Architektur zur Verwaltung persistenter Materialien, ein *Persistentes objektorientiertes Materialmanagementsystem (PooMMeS)*, zu erarbeiten. Diese Architektur teilt sich nach unserer Vision in ein *Materialmanagementsystem (MMeS)* und in ein *Objektmanagementsystem (OMeS)* auf.

> **Begriff 1 :** *Materialmanagementsystem (MMeS)*
> Ein Materialmanagementsystem ist ein rein fachlich motiviertes System. Es dient zur Unterstützung der Organisation von Materialien innerhalb einer Umgebung und liefert eine Sicht über die Umgebungsgrenze hinaus auf einen Materialraum.

> **Begriff 2 :** *Objektmanagementsystem (OMeS)*
> Ein Objektmanagementsystem ist ein rein technisches System. Es stellt einen logischen Raum (Objektspeicher) bereit, der auf der physikalischen Ebene durch einen partitionierten, verteilten Datenspeicher realisiert ist. Das OMeS regelt den (konkurrenten) Zugriff auf Objekte und sorgt so für die Konsistenz und Integrität des Objektspeichers.

Diese PooMMeS-Architektur soll aufbauend auf den technischen Realisierungen persistenter Spracherweiterungen eine rein fachlich motivierte Verwaltung persistenter Materialien auf einfache und geschickte Weise ermöglichen.

Dabei sind folgende Punkte zu betrachten:

1. Klare Trennung der technischen und fachlichen Komponenten der Architektur.

2. Beliebig viele Materialien müssen verwaltbar sein.

3. Beliebig große Materialien müssen handhabbar sein.

4. Die Anzahl der von einem Material referenzierten Instanzen kann sehr hoch sein. Es sollte möglich sein, nur die zu einem Zeitpunkt interessanten Instanzen der Anwendung zur Verfügung zu stellen.

5. Die Integration einer Ausprägung der Architektur in vorhandene WAM-Applikationen sollte sehr einfach sein und möglichst wenige Änderungen nach sich ziehen.

Die Untersuchungen dieser Arbeit beschränken sich zunächst auf Konzepte für eine Realisierung eines Einzelarbeitsplatzes mit zentraler Datenhaltung in der Programmiersprache C++. Dabei sollte die Übertragbarkeit der Konzepte auf Mehrplatz-Arbeitsumgebungen mit dezentralem Datenbestand in anderen Programmiersprachen möglich sein.

1.5 Übersicht

Das Ziel dieser Arbeit ist die Herleitung von Forderungen, die wir an die Gesamtarchitektur PooMMeS stellen wollen. Die Abbildung 1.1 zeigt den Aufbau dieser Arbeit.

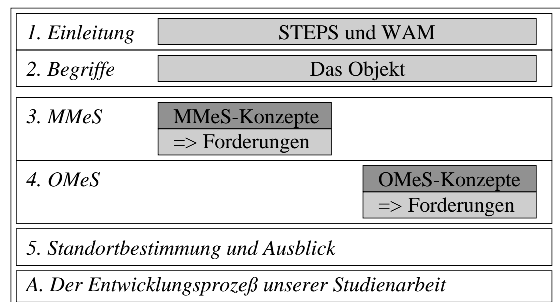


Abbildung 1.1: Der Aufbau dieser Studienarbeit

In Kapitel 2 werden wichtige Begriffe der Objektorientierung eingeführt, die als Grundlage der folgenden Kapitel dienen.

Auf der fachlichen Ebene beginnend liefern wir in Kapitel 3 einen chronologischen Überblick über die bestehenden Ansätze fachlich motivierter persistenter Materialverwaltungen und erheben Forderungen an ein zukünftiges *Materialmanagementsystem (MMeS)*. Die Realisierung unserer Ideen setzt ein neues Repertoire technischer Konzepte voraus, welches wir in Kapitel 4 durch die Vorstellung verschiedener persistenter Spracherweiterungen erarbeiten wollen. Dabei werden wir Forderungen an ein *Objektmanagementsystem (OMeS)* formulieren.

Rückgreifend auf Kapitel 3 und Kapitel 4 stellen wir in Kapitel 5 unseren derzeitigen Stand unserer Arbeit dar. Die noch fehlenden Aspekte und die weiteren Schritte werden genannt.

Im Anhang wird der Verlauf und die Problematik der Teamarbeit dieser Studienarbeit dargestellt.

1.6 Schreibweisen

1.6.1 Textuelle Konventionen

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

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

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

1.6.2 Grafische Konventionen

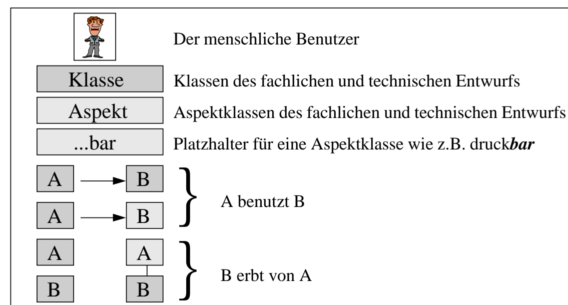


Abbildung 1.2: Konventionen für Klassendiagramme

Kapitel 2

Begriffe

2.1 Das Objekt

„Ein Objekt besitzt einen *Zustand*, ein *Verhalten* und eine *Identität*. Die *Struktur* und das Verhalten gleichartiger Objekte sind in ihrer gemeinsamen Klasse (Typ) festgelegt. Ferner sind die Begriffe *Instanz*[1] und Objekt austauschbar”([Boo91]).

Die Struktur einer Klasse setzt sich aus ihren *Merkmalen* (properties) zusammen. Merkmale sind *Eigenschaften* (attributs) und *Beziehungen* (relationships). Diese können konstant, d.h. in der Zeit unveränderlich, oder im Gegensatz dazu variabel sein.

Der Zustand eines Objektes ist immer auf einen Zeitpunkt bezogen und wird durch die aktuelle Ausprägung seiner Merkmale bestimmt. Das *Verhalten* (behaviour) eines Objektes wird durch seine Operationen festgelegt.

Zur Laufzeit von Anwendungen geben *Meta-Objekt-Daten* Auskunft über die Klassen und deren Instanzen. So kann z.B. Wissen über die Struktur von Objekten oder die Menge der Instanzen einer Klasse gewonnen werden.

2.2 Die Identität

„Das Konzept der Identität ermöglicht es, die Individualität eines Objekts sicherzustellen, unabhängig von dem Weg, über den auf ein Objekt zugegriffen werden kann. Eine Identität ist eine interne Objektqualität” ([Sch+94]).

Der Begriff Identität ist eindeutig von dem Begriff der Adressierbarkeit zu trennen. Mit der Adressierbarkeit wird erreicht, „daß auf ein Objekt evtl. von unter-

[1] Der Begriff *Instanz* ist aus der fehlerhaften deutschen Übersetzung des englischen Wortes *instance* hervorgegangen. Als Synonym ist deshalb auch der Begriff *Exemplar* gebräuchlich.

schiedlichem Kontext (scope) aus zugegriffen werden kann. Adressierbarkeit ist eine externe Objektqualität, die kontextabhängig ist" ([Sch+94]).

[Kir95] stellt folgende Grundsätze für Identitäten auf:

1. Jedes Objekt hat eine eigene, genau ihm zugehörige Identität.
2. Es gibt keine Identität losgelöst von einem Objekt.
3. Die Identität kann durch eine geeignete Form repräsentiert werden (Identitätsrepräsentation).

Die Identität eines Objektes wird häufig mit OID abgekürzt. Sie bleibt dem Benutzer verborgen und ist nicht von ihm manipulierbar.

2.3 Die Vererbung und Beziehungen zwischen Objekten

Objekte können in folgenden Beziehungen stehen:

- *Vererbung*:
  Bei der Vererbung übernimmt eine Klasse die Merkmale und das Verhalten einer Oberklasse. Das geerbte Verhalten ist modifizierbar. Jede Instanz der Klasse kann auch als Instanz der Oberklasse dienen.

- Beziehung (relationship[2]):

  1. *using*: Ein Objekt hat eine Referenz auf ein anderes Objekt.
  2. *containing*: Ein Objekt beinhaltet ein anderes Objekt.

  Beim Klassenentwurf muß eine sorgfältige Wahl der Objektbeziehungen erfolgen. [Boo91] vertritt folgende Meinung: „Containing rather than using an object is sometimes better because containing reduces the number of objects that must be visible at the level of the enclosing object. On the other hand, using is sometimes better than containing because containing leeds to undesirable higher coupling among objects."

2.4 Die Referenz

Eine Referenz stellt einen eindeutigen Bezug zu einem Objekt her. Diese Eindeutigkeit läßt sich erreichen, indem die Referenz durch die Identität des zu referenzierenden Objektes bestimmt wird ([Kir95]). Durch die Einführung von

[2] Begriffsbildung nach [Boo91]

Referenzen, die auch als Stellvertreter für das referenzierte Objekt (*Identitätsrepräsentation*) genutzt werden, kann ein Objekt eine oder mehrere Beziehungen mit einem anderen Objekt eingehen, indem es diese referenziert.

[Kir95] stellt folgende Bedingungen an eine Identitätsrepräsentation:

1. Eindeutigkeit
2. Unabhängigkeit vom aktuellen Zustand
3. Unabhängigkeit von der Umgebung
4. Unabhängigkeit von der Zeit
5. Unabhängigkeit vom Typ

[Kir95] stellt unterschiedliche Realisierungen von Identitätsrepräsentationen gegenüber:

- Adressen
- Strukturierte Bezeichner
- Benutzer-definierte Bezeichner
- Tupel-Bezeichner und Tupel-Surrogat
- Surrogat

2.5 Die indirekte Referenz (smart pointer)

Eine *indirekte Referenz* (smart pointer) stellt nur einen mittelbaren eindeutigen Bezug zu einem Objekt her. Eine indirekte Referenzierung geschieht in mehreren Schritten. Auf diese Weise können zusätzliche Vorgänge angestoßen werden.

Dazu referenziert die indirekte Referenz z.B. auf ein Stellvertreterobjekt (*Schattenobjekt*) oder enthält implizit Informationen, so daß eine zusätzliche Instanz für die Referenzierung auf das gewünschte Objekt sorgen kann. Damit kann eine indirekte Referenz nur in Abhängigkeit von dieser zusätzlichen Instanz als Identitätsrepräsentation genutzt werden. [Bil+93], [Dab+95] und [Vau+92] erörtern detailiert verschiedene Realisierungen von smart pointer-Mechanismen.

2.6 Die Lebensdauer

Die Lebensdauer eines Objektes (*object lifetime*) ist der Zeitraum, in dem es existiert und benutzbar ist (nach [Kir95]). Ein Objekt heißt *persistent*, wenn es seine erzeugende Instanz überlebt, andernfalls ist es *transient*.

Wir sprechen von *orthogonaler Persistenz*, wenn Objekte beliebigen Typs persistent gemacht werden können. Das ist dann der Fall, wenn Persistenz ein grundlegendes Merkmal ist und nicht von anderen Merkmalen eines Objektes abhängt.

In einem System ist die *transparente Persistenz* gegeben, wenn es einem Objekt nicht anzusehen ist, ob es persistent oder transient ist. Die transparente Persistenz schließt die orthogonale ein, aber nicht umgekehrt.

2.7 Die Repräsentation eines Objektes im transienten Speicher

Die Repräsentation von Objekten im transienten Speicher unterliegt keinem in der Literatur festgelegten Standard. Um aber trotzdem Aussagen darüber treffen zu können, wie ein Objekt zur Laufzeit im transienten Speicher repräsentiert wird, haben wir Untersuchungen auf Basis des GNU C++ Compilers durchgeführt. Dieser Abschnitt soll die festgestellten Ergebnisse kurz vorstellen.

Unser Hauptaugenmerk lag bei den Untersuchungen auf der Lokalität der *versteckten Zeiger* (*hidden pointers*) und der Rolle, die sie für die Interpretation der Speicherrepräsentation spielen.

> **Begriff 3 :** *Versteckte Zeiger (hidden pointers)*
> Ein Objekt, das virtuelle Methoden enthält und mehrfach geerbt hat, enthält im transienten Speicher sogenannte versteckte Zeiger (*hidden pointers*), die zum einen auf geerbte Instanzteile (*vbase pointer*) und zum anderen auf die Tabelle der dynamisch gebundenen Methoden (*dispatch table* oder *vtable*) zeigen. Diese versteckten Zeiger sind nur für einen Programmlauf gültig und können deshalb nicht persistent gemacht werden.

Eingebettet in die Speicherrepräsentation, die sich als ein monolytischer Block im Speicher befindet, gibt es neben den versteckten Zeigern die Repräsentationen der Merkmale des Objektes. Diese Merkmale können zum einen die Werte der Attribute des Objektes oder C++-Zeiger auf benutzte Objekte sein.

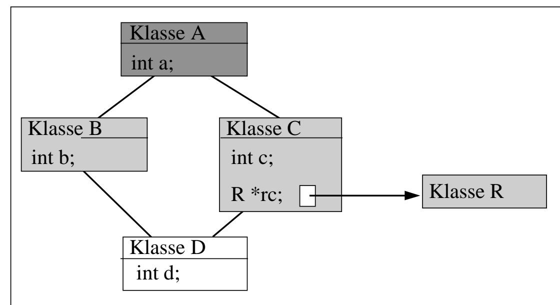


Abbildung 2.1: Klassenhierarchie zur Abbildung 2.2

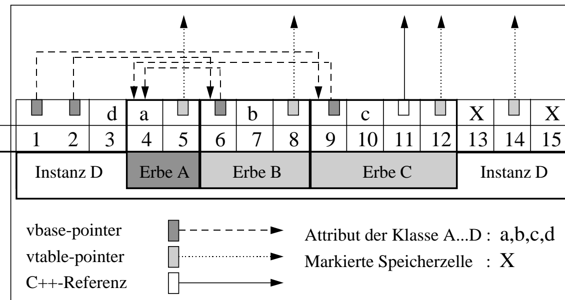


Abbildung 2.2: Speicherrepräsentation eines Objektes der Klasse D

Zum Abschluß dieses Abschnitts soll das Verständnis für den Aufbau der Speicherrepräsentation durch ein Beispiel[3] erhöht werden. Zu diesem Zweck liefert die folgende Liste Interpretationen für die einzelnen Speicherzellen der Speicherrepräsentation aus der Abbildung 2.2:

1. *Klasse D: Erster vbase-pointer (Mehrfachvererbung)*:
   Zeiger auf die Repräsentation der geerbten Instanz der Klasse C

2. *Klasse D: Zweiter vbase-pointer (Mehrfachvererbung)*:
   Zeiger auf die Repräsentation der geerbten Instanz der Klasse B

3. *Klasse D: Merkmal*:
   Der Wert des (nicht ererbten) Attributes d vom Typ integer

4. *Klasse A: Merkmal*:
   Der Wert des (nicht ererbten) Attributes a vom Typ integer

5. *Klasse A: vtable-pointer*:
   Zeiger auf die zugehörigen Methoden der Klasse A

6. *Klasse B: vbase-pointer (Einfachvererbung)*:
   Zeiger auf die Repräsentation der geerbten Instanz der Klasse A

[3] vgl. Abbildung 2.1 und 2.2

7. *Klasse B: Merkmal*:
   Der Wert des (nicht ererbten) Attributes b vom Typ integer

8. *Klasse B: vtable-pointer*:
   Zeiger auf die zugehörigen Methoden der Klasse B

9. *Klasse C: vbase-pointer (Einfachvererbung)*:
   Zeiger auf die Repräsentation der geerbten Instanz der Klasse A

10. *Klasse C: Erstes Merkmal*:
    Der Wert des (nicht ererbten) Attributes c vom Typ integer

11. *Klasse C: Zweites Merkmal*:
    Ein C++-Zeiger auf das von eine Instanz der Klasse R

12. *Klasse C: vtable-pointer*:
    Zeiger auf die zugehörigen Methoden der Klasse C

13. *Markierte Speicherzelle als Flag*:
    Bestimmt zusammen mit Speicherzelle Nr.15 das Ende der Speicherrepräsentation

14. *Klasse D: vtable-pointer*:
    Zeiger auf die zugehörigen Methoden der Klasse D

15. *Markierte Speicherzelle als Flag*:
    Bestimmt zusammen mit Speicherzelle Nr.13 das Ende der Speicherrepräsentation

Kapitel 3

Forderungen an ein Materialmanagementsystem MMeS

3.1 Einleitung

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

Aufgabe dieses Kapitels soll es sein, die bestehenden Ansätze von [Kil+93], [Wul+94] und [Gry95] kurz vorzustellen und gegeneinander abzugrenzen, um abschließend einen Katalog von Forderungen an ein rein fachlich motiviertes Materialmanagementsystem (MMeS) aufstellen zu können.

3.2 Konzeptionelle Anbindung einer relationalen Datenbank ([Kil+93])

3.2.1 Einleitung

Um eine konkrete Datenbank vor einem Anwendungsprogramm zu verbergen, wird in [Kil+93] ein Datenbankautomat beschrieben, der dazu dient, die ihm übergebenden Materialien in einer Datenbank abzulegen, abgelegte Materialien

im transienten Speicher zu rekonstruieren, aber auch Materialien im persistenten Speicher zu löschen.

3.2.2 Der Datenbankautomat

Der Datenbankautomat[1] benutzt dabei den Aspekt *speicherbar*, der das Protokoll für die Ablage und Rekonstruktion persistenter Materialien festlegt. Diese abstrakte Schnittstelle muß in jedem Material konkretisiert werden, indem das Wissen implementiert wird, wie sich das Material ablegen und rekonstruieren läßt.

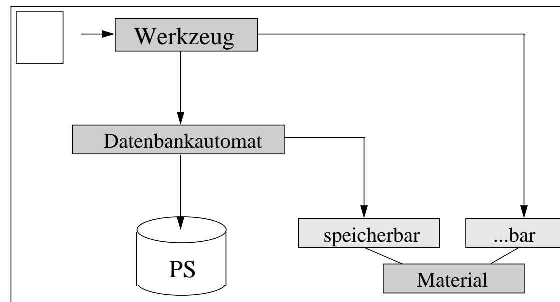
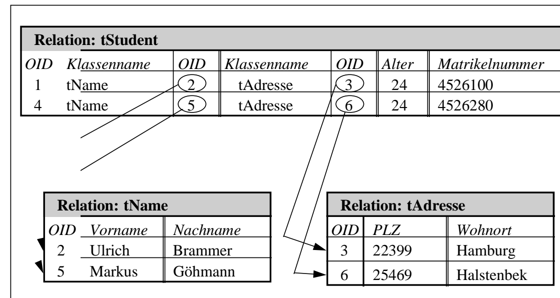


Abbildung 3.1: Einbindung eines Datenbankautomaten

Als Beispiel dient in [Kil+93] der Anschluß einer relationalen Datenbank. Für jede speicherbare Materialklasse existiert eine Relation, die Instanzen der Klasse aufnehmen kann.



Relation: tStudent

OID	*Klassenname*	*OID*	*Klassenname*	*OID*	*Alter*	*Matrikelnummer*
1	tName	2	tAdresse	3	24	4526100
4	tName	5	tAdresse	6	24	4526280

Relation: tName

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

Relation: tAdresse

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

Abbildung 3.2: Relation zur Speicherung einer Übungsgruppe

[1] vgl. Abbildung 3.1

Bei der Ablage eines Materials[2] in die Datenbank wird ein neues Tupel in die zugehörige Relation eingefügt, das als Primärschlüssel den eindeutigen Materialidentifikator[3] des zu speichernden Materials erhält. Zusätzlich wird das Tupel mit den Attributen des Materials gefüllt. Dabei werden Basistypen direkt und Klassentypen als Referenzen, bestehend aus dem Namen der Materialklasse und dem zur Instanz gehörenden Materialidentifikator, in das Tupel aufgenommen. Bei der Speicherung einer solchen Referenz wird gleichzeitig die Ablage des zugehörigen Objektes in die Datenbank angestoßen. Die Ablage komplexer Materialien erfolgt somit durch rekursiven Abstieg, der bis zu den Basistypen durchgeführt wird.

Die Rekonstruktion der Materialien aus der Datenbank erfolgt durch die Umkehrung des beschriebenen Ablaufs. Das Material wird attributweise rekursiv wieder ausgelesen. Dazu ist es erforderlich, daß der Datenbankautomat alle Materialklassen mit zugehörigen Klassennamen kennt, um Materialien erst erzeugen und anschließend mit Werten belegen zu können. Komplexe Materialien sind bei dem Konzept von [Kil+93] nicht partiell handhabbar.

3.3 Das Materialversorgungskonzept ([Wul+94])

3.3.1 Einleitung

In [Wul+94] wird das Problem der persistenten Haltung von Materialien erneut aufgegriffen und die Anbindung an die objektorientierte Datenbank ONTOS als Grundlage für die Beschreibung des fachlichen Konzeptes der *Materialversorgung* genutzt, in welches der Datenbankautomat aus [Kil+93] modifiziert als *Materialversorger* integriert wurde.

Zusätzlich werden Fragen der *Materialverwaltung* innerhalb einer Werkzeugumgebung und die *Materialkoordination* in einem Mehrbenutzersystem beleuchtet. Zu diesem Zweck wird das fachlich motivierte Bild von *Original und Kopie* für den Umgang mit gemeinsamen Materialien genutzt, bei dem nur ein Anwender im Besitz des Originals eines Materials sein kann und damit in der Lage ist, Änderungen an diesem vorzunehmen. Wenn das Original eines Materials ausgecheckt[4] ist, dann ist es anderen Anwendern bis zum Zeitpunkt des Eincheckens[5] nur möglich, Kopien anzufordern und auf diesen zu arbeiten.

---

[2] vgl. Beispiel in Abbildung 3.2

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

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

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

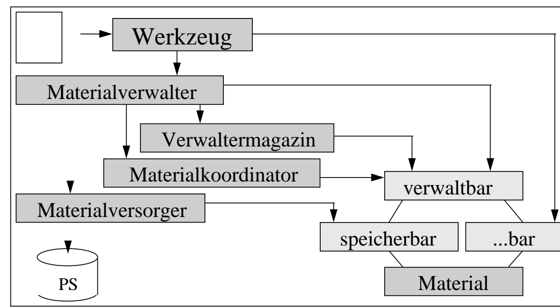


Abbildung 3.3: Die Komponenten des Materialversorgungskonzeptes

Das in [Wul+94] vorgestellte fachliche Konzept für die Materialversorgung[6] besteht aus den im folgenden beschriebenen Komponenten.

3.3.2 Der Materialverwalter

Der *Materialverwalter* ist ein Automat, der in jeder Werkzeugumgebung genau einmal existiert und zur Beschaffung von Materialien und zur Verwahrung und Bereitstellung von Materialien innerhalb der Werkzeugumgebung dient. Damit der Materialverwalter ein Material verwalten kann, muß es die durch den Aspekt *verwaltbar* spezifizierte Schnittstelle aufweisen.

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

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

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

[6] vgl. Abbildung 3.3

Da nach [Wul+94] auch auf Werkzeugumgebungsebene das fachliche Bild von Original und Kopie zum Tragen kommt und deshalb immer nur ein Werkzeug ein Materialoriginal bearbeiten kann, wird im Verwaltermagazin auch abgelegt, ob das Original eines Materials bereits an ein Werkzeug vergeben wurde. Wenn dies der Fall ist, wird auch im Verwaltermagazin verzeichnet, welches Werkzeug im Besitz des Originals ist.

3.3.3 Der Materialversorger

Der *Materialversorger* ist ein Automat, der wie der Datenbankautomat in [Kil+93] das Wissen um die konkrete Schnittstelle des persistenten Speichers kapselt. Der Materialversorger dient als ein „Materiallieferant", der nach außen eine Grundfunktionalität zur Verfügung stellt, mit deren Hilfe Materialien in den persistenten Speicher überführt, aus diesem rekonstruiert und im Speicher gelöscht werden können. Dazu müssen alle Materialien, mit denen der Materialversorger umgeht, die abstrakte Schnittstelle des rein technischen Aspektes *speicherbar* und eine konkrete Implementation dieser aufweisen.

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

Angestoßen wird der Materialversorger ausschließlich durch den Materialverwalter, so daß der Anwender nicht mehr direkt mit einem Speicher- bzw. Ladewerkzeug, wie in vielen Anwendungen üblich, umgehen muß[7], sondern nur noch mit dem Materialverwalter kommuniziert, der den im Hintergrund aktiven Materialversorger zur Aktivierung und Passivierung der gewünschten Materialien benutzt.

3.3.4 Der Materialkoordinator

Auch der zentrale *Materialkoordinator* ist ein Automat. Allerdings wird er nicht wie die bisher vorgestellten Komponenten lokal in einer Werkzeugumgebung, sondern nur einmal in einer Administrationsumgebung[8] erzeugt, so daß jeder lokalen Werkzeugumgebung lediglich ein MaterialkoordinatorProxy[9] zur Verfügung steht.

[7] vgl. hierzu Datenbankautomat nach [Kil+93]

[8] Eine in sich abgeschlossene Umgebung, die sich in mehrere Werkzeugumgebungen auf unterschiedlichen Knoten eines Netzwerks aufteilen kann.

[9] Ein Proxyobjekt dient als ein lokaler Platzhalter eines Objektes, das sich in einem anderen Adreßraum befindet.

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

3.4 Die Materialverwaltung ([Gry95])

3.4.1 Einleitung

Das in [Gry95] dargestellte Konzept der Materialverwaltung[10] entspricht im wesentlichen dem Materialversorgungskonzept aus [Wul+94]. Es finden sich jedoch drei wesentliche Unterschiede, die im folgenden dargestellt werden sollen.

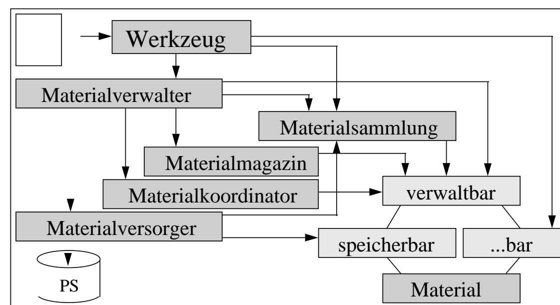


Abbildung 3.4: Die Komponenten der Materialverwaltung

3.4.2 Wesentliche Unterschiede zum Materialversorgungskonzept ([Wul+94])

1. *Keine Zugriffskontrolle innerhalb der lokalen Werkzeugumgebung nach dem Bild von Original und Kopie:*
   Die in [Wul+94] geforderte Zugriffskontrolle auf Werkzeugumgebungsebene wird in [Gry95] nicht realisiert, weil nach Meinung des Autoren der Benutzer einer Einzelplatz-Arbeitsumgebung zu einem Zeitpunkt nur ein Werkzeug verwendet und nicht mehrere gleichzeitig. Aus diesem Grund sei eine Zugriffskontrolle auf dieser Ebene nicht erforderlich.

2. *Ein Materialmagazin statt vieler Verwaltermagazine:*
   Auch die Möglichkeit mehrere Verwaltermagazine innerhalb einer Werkzeugumgebung einzusetzen, wird in [Gry95] nicht länger unterstützt. Zudem

[10] vgl. Abbildung 3.4

erfolgt eine Umbenennung des Verwaltermagazins in Materialmagazin, die wir für die folgende Diskussion übernehmen wollen.

3. *Materialsammlung als Ergebnis von Anfragen:*
   Das Ergebnis einer Anfrage eines Werkzeuges an den Materialverwalter ist in [Gry95] ein Behälter, der alle Materialien enthält, die der Anfrage genügen. Dieser Behälter wird *Materialsammlung* genannt und dient nach Angaben des Autoren auch als Ergebnis von Anfragen des Materialverwalters an einen Materialversorger.

3.5 Kritische Würdigung

Den vorgestellten Ansätzen liegt der Gedanke einer rein fachlich motivierten Materialverwaltung zu Grunde. Allerdings wurde dieses Ziel nicht erreicht. Dieses begründen wir wiefolgt:

- Die Nutzung der beschriebenen Mechanismen zur Ablage und Rekonstruktion von Materialien impliziert, daß komplexe Objekte nur komplett in den transienten Speicher geladen und auch nur komplett wieder aus diesem verdrängt werden können. In der in [Wul+94] beschriebenen Beispielimplementation entfällt im Vergleich zur Anbindung einer relationalen Datenbank in [Kil+93] lediglich die aufwendige Segmentierung der Materialien bzw. die Desegmentierung der Datenbankrepräsentation beim Datenaustausch zwischen der Datenbank und dem transienten Speicher, weil die Abbildung direkt von dem Anwendungsobjekt auf ein ONTOS-Objekt und umgekehrt erfolgen kann.

- In allen Ansätzen hat das Material Wissen darüber, wie es abgelegt (store) und in der Umkehr rekonstruiert (load) wird. Dadurch wird das Material zu einer aktiven Komponente der Anwendung - Widerspruch zur Definition des Materials in Abschnitt 1.3!

- Ein Materialversorger dient lediglich zur Umwandlung einer persistenten in eine transiente Repräsentation von Materialien und umgekehrt. Dieser Materialtransformation kann kein fachlich motivierter Vorgang zugeordnet werden, da sich Materialien einer Arbeitsdomäne in der Regel nicht verändern, wenn sie abgelegt werden. Damit ist ein Datenbankautomat wie auch ein Materialversorger eine rein technische Komponente und darf deshalb nicht als fester Bestandteil in eine fachlich motivierte Materialverwaltung integriert werden.

3.6 Forderungskatalog

Ziel der Entwicklung eines Materialmanagementsystems (MMeS) muß es sein, von der Persistenz der Materialien und fachlichen Werte auszugehen, ohne daß dafür die explizite Integration eines Materialversorgers erforderlich ist. Dazu ist der Materialversorger aus dem fachlichen Konzept herauszulösen und die Materialversorgung implizit im Hintergrund durchzuführen.

Dieses Ziel kann aber nur durch die Erfüllung folgender Forderungen an ein Materialmanagementsystem (MMeS) erreicht werden:

- Trennung der fachlichen von den technischen Komponenten.
- Alle Materialien und fachlichen Werte sind a priori persistent[11].
- Das Material darf nicht „wissen", auf welche Weise und wann es gespeichert wird.

Außerdem stellen wir folgende Forderungen an das Materialmanagementsystem (MMeS):

- Der Materialversorger ist eine technische Komponente und darf deshalb nicht Bestandteil eines MMeS sein.
- Das Materialmagazin soll ein Behälter sein, der alle Materialien einer Werkzeugumgebung enthält. Dieser Behälter sollte durch Kriterienvergabe logisch partitioniert werden können und somit zur fachlichen Organisation der Materialien innerhalb der Werkzeugumgebung beitragen. Kriterien könnten z.B. fachliche Ortsangaben, Zeitangaben oder andere der Arbeitsdomäne entnommene Angaben sein.
- Der Materialverwalter soll zu einem interaktiven Werkzeug werden. Dieses könnte durch Implementation eines „Navigationstools" erfolgen, mit dem es dem Benutzer einer Anwendung möglich ist, sich einen Überblick über die Materialien innerhalb und außerhalb seiner lokalen Werkzeugumgebung zu verschaffen und Materialien über den Materialverwalter interaktiv anzufordern. Daraus ergibt sich folgender Umgang des Anwenders mit Materialien:
  1. Materialien sollten direkt über den Materialverwalter bezogen werden können.
  2. Materialien sollten wie bisher aus einem Werkzeug mit Hilfe des Materialverwalters bezogen werden können.

[11] Fachliche Motivation: Arbeitsgegenstände der Arbeitsdomäne können nicht in transiente und persistente unterschieden werden.

Hier stellt sich die grundsätzliche Frage, ob zuerst ein Material und dann ein dazu passendes Werkzeug oder in der Umkehr erst das Werkzeug und dann ein dazu passendes Material selektiert wird. Im ersten Fall müßte der Werkzeugumgebung ein „Werkzeugverwalter" bereitgestellt werden, der eine Selektion von Werkzeugen ermöglicht.

Der Einsatz eines Materialmanagementsystems darf nicht auf eine Einzelplatz-Arbeitsumgebung beschränkt bleiben und muß deshalb die Option der Nutzung innerhalb eines verteilten Systems offenhalten:

- Dem Materialmagazin sollte ein *Materialkatalog* zur Seite gestellt werden, der die Materialien außerhalb der Werkzeugumgebung enthält, um auf diese Weise einen Überblick über alle benutzbaren Materialien zu ermöglichen.
- Nach unserer Auffassung ist der Materialkoordinator sowohl eine fachliche als auch eine technische Komponente:
  - *fachlich*:
    Der Materialkoordinator bedient sich des fachlich motivierten Konzeptes von Original und Kopie zwischen mehreren Arbeitsumgebungen.
  - *technisch*:
    Der Materialkoordinator setzt das Konzept von Original und Kopie auf der technischen Ebene durch Realisierung eines wechselseitigen Ausschlusses um. Damit kann die Konsistenz der Daten im persistenten Speicher zugesichert werden.

  Es ist deshalb notwendig, den Materialkoordinator in seine fachlichen und technischen Anteile aufzuspalten und diese getrennt zu betrachten.
- Materialien sollten über die Grenze der lokalen Werkzeugumgebung hinaus eindeutig durch einen Materialidentifikator identifizierbar und lokalisierbar sein.

Dieser Katalog von Forderungen dient als Orientierung auf dem Weg zu einem Materialmanagementsystem und ist gleichzeitig für die Bewertung einer späteren Realisierung notwendig.

Kapitel 4

Forderungen an ein Objektmanagementsystem OMeS

4.1 Motivation

Im letzten Kapitel haben wir herausgearbeitet, daß der Materialversorger, der das Bindeglied zwischen Materialverwalter und persistentem Speicher darstellt, keine fachliche Komponente ist. Zur Realisierung der Datenversorgung eines Materialmanagementsystems (MMeS) führen wir deshalb als technische Komponente das Objektmanagementsystem (OMeS)[1], das die Funktionalität des Materialversorgers ersetzen und erweitern soll, ein.

4.2 Einleitung

Bereits Anfang der 80er Jahre wurde die Unzulänglichkeit des relationalen Datenmodells (RDM) für sogenannte Nicht-Standard-Anwendungen, wie z.B. CAD-Systeme, erkannt. Die in solchen Anwendungen verwendeten komplexen Objekte müssen bei der Abbildung auf Relationen segmentiert und in der Umkehr aufwendig desegmentiert werden, was sowohl zu Ineffizienz während der Anwendungsentwicklung als auch zur -laufzeit führt.

Zwischen den Anwendungen und damit den Programmiersprachen auf der einen Seite und den relationalen Datenbanken und persistenten Speichern auf der anderen hatte sich eine „Lücke" (*impedance mismatch*) aufgetan, welche in den folgenden Jahren durch drei unterschiedliche Herangehensweisen zu schließen versucht wurde:

1. **der evolutionäre Ansatz:** Das relationale Datenmodell wurde in mehreren Schritten über das NF2-Datenmodell zu Datenmodellen weiterent-

[1] vgl. Begriffsbildung in der Einleitung

wickelt, die auf komplexen Objekten basieren. Ein Vertreter ist das Molekül-Atom-Datenmodell (MAD), welches Mitte der 80er Jahre entstand ([Mit86]).

2. **der revolutionäre Ansatz:** Anstatt das relationale Datenmodell zu erweitern, wurde das objektorientierte Programmierparadigma auf die Datenbankwelt übertragen. Es entstand das objektorientierte Datenmodell (OODM), das heute durch die wachsende Verbreitung der objektorientierten Sichtweise beginnt, in Form objektorientierter Datenbanken (OODB) die relationalen teilweise zu substituieren.

   Bislang existiert für das objektorientierte Objektmodell, das im Vergleich zum relationalen auf weniger formalen Grundlagen basiert, noch kein allgemein anerkannter Standard. Eine erste Beschreibung einer solchen Norm hat die Object Database Management Group (ODMG) 1993 in [Cat+94] veröffentlicht. Darin wird eine Object Definition Language (ODL), eine Object Manipulation Language (OML) und eine Object Query Language (OQL) definiert und deren Einbettung in die objektorientierten Programmiersprachen C++ und Smalltalk beschrieben.

3. **der programmiersprachliche Ansatz:** Im Gegensatz zu den beiden ersten Ansätzen, die den impedance mismatch von der Datenbankseite zu beseitigen versuchen, werden neue Programmiersprachen entwickelt oder bestehende um Persistenzkonzepte (bis hin zu klassischer Datenbankfunktionalität) ergänzt. Auf diese Weise soll ermöglicht werden, persistente Objekte direkt im Programm zu erzeugen, zu modifizieren und zu beseitigen. Eine solche persistente Spracherweiterung kann entweder durch Einbindung einer Klassenbibliothek oder durch Erweiterung des Sprachstandards um eine *erweiterte Syntax* (neue *Schlüsselworte*) oder geänderte *Semantik des Befehlssatzes* (z.B. persistente Speicherallokation durch new-Operator) unter Einsatz eines *Precompilers* erreicht werden.

Im folgenden werden mehrere Entwicklungen des programmiersprachlichen Ansatzes vorgestellt und analysiert. Bei dieser Vorstellung sind wir vor allem an den Lösungsansätzen der folgenden Problemstellungen interessiert:

1. *Erweiterung der Programmiersprache*

   - Durch welche ergänzenden Komponenten wird die Persistenz umgesetzt?
   - Wie sieht der Umgang mit den Komponenten aus?
   - Welche Verfahren zur Speicherrückgewinnung (*garbage collection*) werden im transienten und persistenten Speicher eingesetzt?

2. *Umgang mit Objekten*

    - Wird transparente Persistenz unterstützt?
    - Wird orthogonale Persistenz unterstützt?
    - Wie werden transiente und/oder persistente Objekte identifiziert?
    - Wie werden transiente und/oder persistente Objekte erzeugt?
    - Wie werden transiente und/oder persistente Objekte modifiziert?
    - Wie werden transiente und/oder persistente Objekte vernichtet?

3. *Repräsentationen von Objekten*

    - Welche Repräsentation hat ein Objekt zur Laufzeit im Hauptspeicher?
    - Wie werden Eigenschaften, Beziehungen und Operationen von Objekten dargestellt?
    - Wie werden Vererbung, Polymorphie und Kapselung realisiert?
    - Welche Rolle spielt dabei die Wahl des Compilers?

4. *Datenaustausch zwischen transientem und persistentem Speicher*

    - Welche Konzepte werden für den Objekttransfer eingesetzt?
    - Wie granular ist der Objekttransfer?
    - Welche Instanz sorgt für die Zuordnung zwischen transienter und persistenter Objektrepräsentation?
    - Wo befinden sich Meta-Objekt-Daten und wie sehen sie aus?

5. *Erhaltung der Konsistenz*

    - Welche Mechanismen (Transaktion, Fehlerbehebung) werden zur Erhaltung der Konsistenz eingesetzt?

6. *Anbindung von persistenten Speichern*

    - Wird eine Anbindung von relationalen Datenbanken unterstützt?
    - Wird eine Anbindung von objektorientierten Datenbanken unterstützt?
    - Welche Funktionalität der Datenbanken wird genutzt?
    - Wie wird die Anbindung realisiert?

7. *Zugriff auf Objekte*

    - Wie werden Objekte benannt?
    - Welche Verfahren (Assoziation, Navigation, Anfrage) zur Objektsuche im transienten und persistenten Speicher werden unterstützt?

8. *Skalierbarkeit des Ansatzes*
   - Ist die Nutzung des Ansatzes in einem verteilten System möglich?
   - Kann der Adreßraum beliebig skaliert werden?

Die Fragestellungen 6 und 8 wollen wir lediglich genannt haben, sollen im folgenden aber nicht erörtert werden. Aufgrund der Lösungsansätze für die Problemstellungen werden im letzten Abschnitt dieses Kapitels Anforderungen an das Konzept eines *OMeS* (Objektmanagementsystem) erarbeitet.

4.3 Darstellung persistenter C++-Erweiterungen

Grundlage der zu beschreibenden Erweiterungen ist die Programmiersprache C++, die im Kern nur transiente Objekte erzeugen, modifizieren und vernichten kann. Erst durch Einbindung sogenannter include-Dateien erhält die Sprache die Mächtigkeit auch auf ein Dateisystem als persistenten Speicher zugreifen zu können. Dafür stehen in der Sprache logische Dateien (*Streams*), die direkt im Programmcode angesprochen werden, zur Verfügung.

Diese Situation wollen wir als Ausgangspunkt für die folgenden Betrachtungen nehmen und aufeinander aufbauend die persistenten C++-Erweiterungen Arjuna, E, O++ und Texas vorstellen.

4.3.1 Arjuna

Einleitung

Die Entwickler von Arjuna[2] verfolgten das Ziel, eine Programmiersprache für verteilte Anwendungen zu realisieren und dabei Aspekte der Fehlertoleranz zu berücksichtigen. Zu diesem Zweck wurde die Programmiersprache C++ um persistente Objekte und Transaktionen erweitert, die durch die Einbindung einer Klassenbibliothek, die in einer Klassenhierarchie organisiert ist, realisiert werden. Zusätzlich erfordert die Übersetzung eines Arjuna-Programmes einen Precompiler, der aber lediglich für die Generierung der Kommunikationswege zwischen den Objekten (Stub-Generierung) notwendig ist und somit aus der für uns relevanten Betrachtung ausgeklammert werden kann.

Arjuna basiert auf einem sogenannten „natürlichen” Objektmodell, in welchem jedes Objekt als Bestandteil einer Objektgemeinschaft gesehen wird, in der jedes Mitglied einem festen Netzknoten zugeordnet ist, an dem es, durch Remote Procedure Calls (RPC) angestoßen, Zustandswechsel vornehmen und auf Anfragen

[2] Arjuna ist Ende der achtziger Jahre unter der Leitung von S. K. Shrivastava an der University in Newcastle entwickelt worden ([Shr88],[Dix88],[Dix+89]).

reagieren kann.

Arjuna realisiert Verteilung, Fehlertoleranz und Persistenz ausschließlich auf der Sprachebene und mit Hilfe des UNIX-Dateisystems als persistenten Speicher, und benötigt deshalb keine weiteren Komponenten, wie z.B. Datenbanken.

Die folgenden Betrachtungen fokusieren lediglich auf die Persistenzmechanismen von Arjuna; weitere Informationen über Verteilung und Fehlertoleranz werden in [Shr88], [Dix88] und [Dix+89] ausführlich vorgestellt.

Anbindung von persistenten Speichern

Der persistente Speicher von Arjuna (*Kubera*) basiert auf dem UNIX-Dateisystem und wird durch eine Instanz der Klasse *ObjectStore* gekapselt. Diese Instanz stellt zur Aktivierung eines persistenten Objektes die Operation *read_state()* und zur Passivierung *write_state()* zur Verfügung.

Umgang mit Objekten

Der Name eines persistenten Objektes (ein *ArjunaName*) wird durch den Ort des Objektes bestimmt. Er setzt sich aus dem Netzknoten, dem durch die Klasse festgelegten Dateiverzeichnis und dem Dateinamen zusammen. Im Arjuna-Code erfolgt die Bindung eines solchen Namens an ein persistentes Objekt mit der Definition der Instanz. So wird z.B. durch

```
tKonto MeinKonto (''Konto 4711'',''Rechner 30'');
```

der Name „Konto 4711" an die Instanz *MeinKonto* der Klasse *tKonto* gebunden und festgelegt, daß sich das Objekt auf dem Rechner Nr.30 befindet. Wird das Objekt dort nicht gefunden, wird am entsprechenden Ort im Verzeichnis „tKonto" die Datei „Konto 4711" zur Laufzeit erzeugt, die anschließend als persistente Repräsentation von *MeinKonto* dient.

Datenaustausch zwischen transientem und persistentem Speicher

Der Implementator eines Arjuna-Programmes muß dafür sorgen, daß jedes persistente Objekt Instanz einer Unterklasse der Klasse *StateManager* ist.

In dieser virtuellen Basisklasse wird die Schnittstelle für die zur Persistenz erforderlichen Mechanismen[3] festgelegt, die im wesentlichen aus drei Operationen besteht:

[3]vgl. Abbildung 4.1

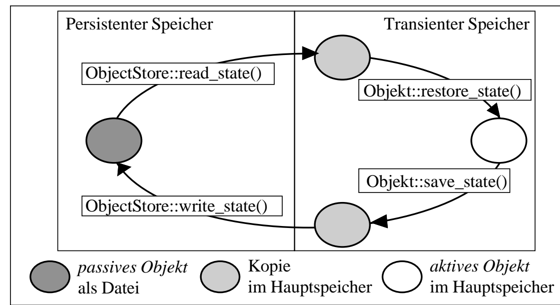


Abbildung 4.1: Der Objekttransfer in Arjuna

1. Durch Aufruf von *activate()* wird das Arjuna-Objekt attributweise aus dem persistenten in den transienten Speicher kopiert. Dies geschieht durch den Aufruf der geschützten Operation *restore_state*, die das dafür notwendige Meta-Objekt-Wissen enthält. Die Implementation dieses Wissens ist Aufgabe des Implementators.

2. Durch Aufruf von *deactivate()* wird das transiente Arjuna-Objekt attributweise im persistenten Speicher abgelegt. Auch hier muß der Programmierer für die Implementation des Meta-Objekt-Wissens in einer geschützten Operation (*save_state*) sorgen.

3. Durch Aufruf von *destroy()* wird ein Arjuna-Objekt sowohl im transienten als auch im persistenten Speicher freigegeben und steht anschließend nicht länger zur Verfügung. Im Rumpf der Operation erfolgt ein Aufruf der geschützten Operation *delete_state*.

Komplexe Objekte werden in Arjuna-Programmen ausschließlich attributweise abgelegt und rekonstruiert und können damit lediglich im Ganzen aktiviert und passiviert werden. Dem Implementator bleibt allerdings überlassen, welche Attribute abgelegt werden sollen. Die Festlegung der Reihenfolge erfolgt durch einen rekursiven Abstieg. Für fundamentale Typen (z.B. *int*, *char*, usw.) stehen in Arjuna spezielle Operationen zur Ablage im persistenten Speicher (*pack()*) und zur Rekonstruktion im transienten Speicher (*unpack()*) zur Verfügung, so daß Attribute dieser Typen den Abschluß der Rekursion bilden.

Erhaltung der Konsistenz

In den meisten Arjuna-Programmen erfolgt kein expliziter Aufruf von *activate()* und *deactivate()*, weil alle Objektmodifikationen innerhalb von (geschachtelten) Transaktionen erfolgen. Der notwendige Aufruf der Operationen findet bei der Erzeugung bzw. dem Abschluß der Transaktionen statt. Auf diese Weise wird ein zu veränderndes Objekt „automatisch" aktiviert (*activate*), d.h. aus dem persistenten Speicher in den transienten überführt, und nach erfolgreichem Abschluß

(COMMIT) „automatisch" passiviert (*deactivate*), d.h. in den persistenten Speicher zurückgeschrieben. Bei einem Abbruch der Transaktion (ABORT) werden die bereits gemachten Änderungen nicht übernommen und sind verloren.

Verwandte Arbeiten

Etwa zur gleichen Zeit wie Arjuna entstand Avalon/C++, welches auch für die Erstellung verteilter und fehlertoleranter Programme entwickelt wurde. Die Nutzung der Persistenzmechanismen wurde wie in Arjuna durch Bereitstellung einer hierarchischen Klassenbibliothek ermöglicht.

4.3.2 E

Einleitung

E[4] wird in der Literatur als die erste persistente Programmiersprache, die transparente Persistenz unterstützt, bezeichnet. Sie entstand im Rahmen des EXODUS Projektes an der University of Wisconsin.

Aufgabe dieses Projektes war es, eine Umgebung zu entwickeln, die es dem Implementator einer Anwendung ermöglicht, direkt in einer zu C++ aufwärts kompatiblen Programmiersprache auf Datenbankfunktionalität zuzugreifen.

Erweiterung der Programmiersprache

Aus dem EXODUS Projekt ist das aus drei Hauptkomponenten bestehende EXODUS Toolkit entstanden:

- Der *EXODUS Storage Manager* stellt einen gekapselten persistenten Speicher sowohl für Objekte als auch Instanzen fundamentaler Typen (wie z.B. int, float, char, usw.) durch Bereitstellung von Grundoperationen für das Ein- und Auschecken dieser zur Verfügung.

- Die *Programmiersprache E* stellt Typisierung, Funktionen und Prozeduren bereit, die als ODL, OML und OQL des EXODUS Storage Managers dienen. Der dafür erforderliche *E-Compiler* entstand durch die Erweiterung des AT&T C++-Compilers cfront zu efront, der die erweiterte Syntax von E in C-Code übersetzt. Dabei werden auch Aufrufe des EXODUS Storage Manager (sogenannte Calls), die dem Implementator verborgen bleiben, in den Code integriert.

[4] Die Programmiersprache E wird seit Ende der achtziger Jahre gemeinsam an der University of Wisconsin in Madison und dem IBM Almaden Research Center entwickelt ([Ric+89],[Ric+93]).

- Der *EXODUS Query Generator* erlaubt es dem Datenbankimplementator eine benutzerdefinierte Anfrageoptimierung vorzunehmen.

Datenaustausch zwischen transientem und persistentem Speicher

Der EXODUS Storage Manager bietet ein Grundmanagement für Objekte und Transaktionen und stellt die Schnittstelle zwischen der Anwendung und dem persistenten Speicher dar. Der persistente Speicher wird nach außen als eine Menge sogenannter *StorageObjects* zugänglich gemacht. Diese bestehen aus einer beliebig großen Liste von uninterpretierten Speicherzellen und können Objekte fast beliebiger Größe (bis mehrere Gigabytes) aufnehmen.

Jede persistente Repräsentation eines persistenten Objektes wird durch eine 12 Byte lange OID des zugehörigen StorageObjects benannt; der Name einzelner Attribute besteht zusätzlich noch aus einem 4 Byte langen Offset. Beide Angaben zusammen bilden einen *persistent handle*.

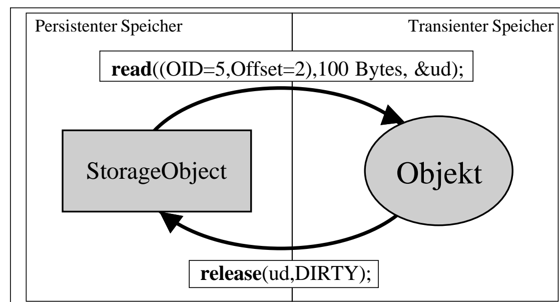


Abbildung 4.2: Der Objekttransfer in E

Für den Objekt- und Attributzugriff stellt der EXODUS Storage Manager die Basisoperationen *read()* und *release()* zur Verfügung[5]:

- *read(persistent handle, Länge, Zeiger auf einen user descriptor)* dient zum Auschecken (pin) des durch den *persistent handle* und die Länge spezifizierten Attributes oder Objektes. Der EXODUS Storage Manager übergibt der Anwendung eine Referenz auf eine transiente Repräsentation des Objektes (*user descriptor*) in Form eines C++-Zeigers.

- *release(user descriptor, Zustand)* dient zum Einchecken (unpin) des user descriptors in den persistenten Speicher. Der Zustand gibt an, ob Modifikationen am user descriptor vorgenommen worden sind (DIRTY) oder nicht (CLEAN).

[5] vgl. Abbildung 4.2

E arbeitet damit, wie das etwa zur gleichen Zeit entstandene Arjuna, nach dem load/store-Modell, bei dem jedes Objekt vor einer Modifikation aktiviert und danach wieder deaktiviert werden muß. Ein grundlegender Unterschied liegt in der Art und Weise, mit der die erforderlichen Operationen im Programmcode platziert werden. Während in E der Compiler efront die Kommandos möglichst „optimal" platziert, ist in Arjuna der Implementator direkt oder indirekt über die Transaktionen dafür zuständig.

Erhaltung der Konsistenz

Um die Konsistenz des Datenbestandes zu garantieren, wird zwischen der Anwendung und dem EXODUS Storage Manager ein Vertrag über die Nutzung der *read()*- und *release()*-Operationen geschlossen, der folgende Bestimmungen enthält:

1. Der EXODUS Storage Manager verschiebt kein ausgechecktes Objekt im transienten Speicher.

2. Die Anwendung greift nur auf Objekte zu, die zuvor durch die *read()*-Operation ausgecheckt worden sind (pin).

3. Die Anwendung checkt ein ausgechecktes Objekt in einer „zügigen" Weise durch die *release()*-Operation wieder ein (unpin).

Umgang mit Objekten

E stellt für die Definition von Klassen und Typen und deren persistente Instanzen zwei neue Schlüsselworte zur Verfügung:

- *db*:
  Durch Einsatz des Schlüsselwortes *db* wird dem E-Compiler signalisiert, daß die Instanz des im Quellcode folgenden *db*-Typs persistent sein kann. Für jeden fundamentalen C++-Typ und für die Schlüsselworte *class*, *struct* und *union* gibt es in E das zugehörige *db*-Pendant (*dbshort*, *dbint*, *dblong*, *dbfloat*, *dbdouble*, *dbchar*, *dbvoid* sowie *dbclass*, *dbstruct* und *dbunion*). Zusätzlich sind Zeiger auf *db*-Typen erlaubt.

  Referenzen auf *db*-Instanzen werden durch den Compiler in *persistent handles* transformiert. Im Gegensatz dazu werden Referenzen auf Instanzen von nicht *db*-Typen unverändert wie normale C++-Referenzen behandelt.

- *persistent*:
  Durch das Schlüsselwort *persistent* wird dem E-Compiler angegeben, daß die folgende Instanz persistent ist. Das ist aber nur für Instanzen von *db*-Typen möglich.

Im Gegensatz zur Definition transienter und persistenter Objekte im Programmcode ist die Manipulation dieser syntaktisch transparent und erfordert keine besondere Behandlung durch den Programmierer.

Alle für den Objekttransfer zwischen dem transienten und dem persistenten Speicher erforderlichen Aufrufe des EXODUS-Storage Managers werden während der Übersetzung durch den im folgenden Abschnitt beschriebenen E-Compiler efront integriert. Die Aufrufe sind somit implizit in dem Implementationscode enthalten und damit für den Programmierer transparent.

Zugriff auf Objekte

Wie in der Einleitung bereits beschrieben, ist der E-Compiler ein erweiterter AT&T C++-Compiler. Der E-Compiler[6] überführt die speziellen E-Konstrukte in C-Ausdrücke und Aufrufe des EXODUS Storage Managers.

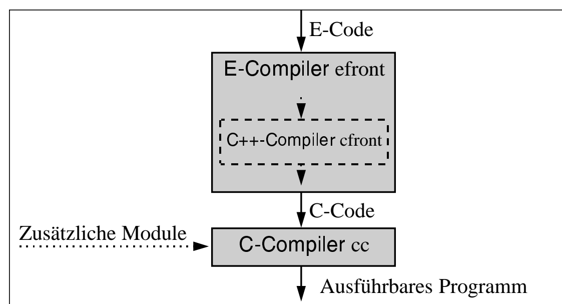


Abbildung 4.3: Der E-Compiler efront

Bei dieser Übersetzung werden im Gegensatz zu Referenzen auf Instanzen von nicht-*db*-Klassen, die unverändert C-Referenzen bleiben, Referenzen auf *db*-Objekte in sogenannte *db*-Referenzen transformiert, die als Instanzen des *struct DBREF* modelliert sind und einen *persistent handle* repräsentieren.

Die erforderliche Abbildung von E-Variablennamen auf die zugehörigen *persistent handles* erfolgt zur Übersetzungszeit. Erkennt der Compiler eine persistente Instanz eines *db*-Typen, dann wird ein neuer *persistent handle* bei dem EXODUS Storage Manager angefordert und statisch in das ausführbare Programm integriert. Damit dient das Programm als Zuordnungsinstanz zwischen transienten und persistenten Repräsentationen der Objekte. Diese Zuordnung geht erst dann verloren, wenn das compilierte Programm gelöscht wird.

---
[6] vgl. Abbildung 4.3

Entwurfsentscheidungen

- *keine Orthogonalität der Persistenz*:
  Die Entwickler von E haben sich bewußt gegen die orthogonale Persistenz entschieden, weil es erstens nicht sinnvoll sei, alle Objekte persistent machen zu können, zweitens der C++-Dereferenzierungsmechanismus unverändert für nicht-*db*-Instanzen mit gleicher Performanz bestehen bleibt und drittens die Größe der OIDs klein gehalten werden kann.

- *global hash table und tag-Zuordnungstabelle zum Umgang mit versteckten Zeigern*:
  Jedes Objekt einer *db*-Klasse erhält bei der Compilierung einen, durch die *db*-Klasse festgelegten, unverwechselbaren *tag*, der in einer persistenten *tag-Zuordnungstabelle* abgelegt ist. Der *tag* ist ein Verweis auf einen Eintrag in der *global hash table*, die bei jedem Programmstart initialisiert wird. Mit Hilfe der *global hash table* kann über die *vtable* auf die korrekte Methode eines Objektes zugegriffen werden (siehe Abbildung 4.4).

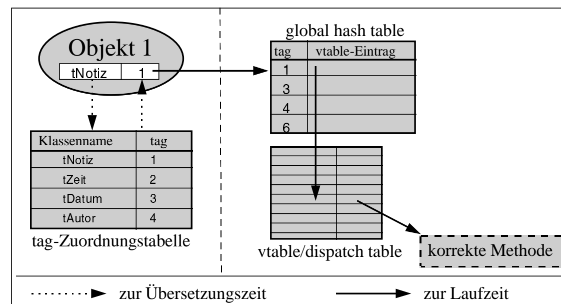


Abbildung 4.4: Lösung des hidden pointer-Problems in E

Weiterführende Arbeiten

Die oben gegebene Beschreibung von E basiert ausschließlich auf der ersten Version von E aus dem Jahr 1988. In den folgenden Jahren entstand eine Vielzahl unterschiedlicher efront-Compiler. Dieses Kapitel wollen wir deshalb mit einer kurzen Darstellung der Veränderungen des aktuellen E-Compilers beenden:

- *dynamisch erzeugte persistente Objekte:* Alle Persistenz-Mechanismen sind in der vorgestellten Version nur für statisch definierte Objekte verwendbar, weil der E-Variablenname zur Übersetzungszeit an den zugehörigen *persistent handle* gebunden wird. Dieses Vorgehen schließt eine dynamische Instanzierung von persistenten Objekten aus.

  Um diese Einschränkung zu beseitigen, wurden in E Version 2.0 der *new()*- und der *delete()*-Operator überladen. Auf diese Weise können persistente

Objekte dynamisch erzeugt und als Elemente in sogenannte Kollektionen eingefügt werden. Die Kollektionen werden statisch zur Übersetzungszeit instanziert.

- *Wechsel vom load/store-Modell zu einem virtuellen Speichermanagement:* Bereits im Jahr 1990 fand ein Wechsel von dem zuvor propagierten load/store-Modell zu einer auf einem virtuellen Speichermanagement basierenden Architektur statt. Aufgabe dieser war es, die Objekte nicht unmittelbar nach einer Modifikation im transienten Speicher wieder in den persistenten Speicher zurückzuschreiben, sondern diesen Vorgang erst dann auszulösen, wenn transienter Speicher benötigt wird.

  Bei der Realisierung dieser Architektur gibt es zwei unterschiedliche Realisierungen: Ein Ansatz nutzt den Puffer des EXODUS Storage Managers als Objektcache und stellt eine Hash-Taballe zur Lokalisierung der Objekte bereit. Einen anderen Ansatz bietet die Version 3.0, in der das Objekt in einen virtuellen Speicher kopiert wird, und dabei die auf Objektidentifikator basierenden Referenzen in virtuelle Speicheradressen überführt werden (*pointer swizzling*[7]).

- *Unabhängigkeit von AT&T C++:* Die Entwickler von E erkannten, daß sie sich mit ihrer Architektur in die Abhängigkeit des AT&T C++-Compilers begeben hatten. Aus diesem Grund fanden Anfang der 90er Jahre Portierungen auf andere Compiler und Betriebssysteme statt. So basiert z.B. die Version 3.0 auf GNU C++.

Die neueren Versionen von E sind im Kontext mit dem im folgenden vorgestellten Ansatz von O++ zu sehen, weil O++ etwa zur gleichen Zeit und mit Unterstützung der University of Wisconsin entstand.

4.3.3 O++

Einleitung

Die erste Version von O++[8] entstand im Jahr 1993. Die Sprache dient zur Programmierung des Ode Object Managers, der Bestandteil des Ode Datenbanksystems ist. Wie E sollte auch O++ durch Integration von Datenbankfunktionalität in die Programmiersprache C++ zu einer allgemein anwendbaren Sprache sowohl für datenbankintensive als auch -extensive Anwendungen werden.[9] O++ stellt ein integriertes Datenmodell zur Verfügung, das sowohl für Datenbankanwendungen

[7] vgl. Seite 44

[8] O++ ist in den AT&T Bell Laboratories in Murray Hill/New Jersey entwickelte persistente C++-Erweiterung ([Agr+93]).

[9] So wie O++ durch die Programmiersprache E beeinflußt wurde, so ist auch der Einfluß von O++ auf aktuelle Versionen von E unverkennbar.

als auch für allgemeine Belange (*general purpose*) geeignet ist. So wird z.B. ein Query-Konstrukt zur Verfügung gestellt, mit dessen Hilfe jeweils über einen Teil des Datenbestandes iteriert werden kann.

Erweiterung der Programmiersprache

Der Ode Object Manager bietet die Grundfunktionalität zur Erzeugung und Modifikation persistenter Objekte und bietet dabei die Sicht auf eine globale Datenbank, die aus einer Sammlung separater lokaler Datenbanken (*cluster groups*)[10] besteht.

Eine als UNIX-Datei realisierte *cluster group* besteht aus einer Menge von *clusters*, die sich wiederum aus einer Menge von *Tripeln* zusammensetzen. Diese Tripel bestehen aus einem Objektidentifikator, einer Referenz auf die persistente Repräsentation und ggf. einer Referenz auf die transiente Repräsentation eines Objektes. Damit fungiert eine *cluster group* sowohl als Zugriffsinstanz auf persistente Objekte als auch als Zuordnungsinstanz zwischen persistenter und transienter Repräsentation dieser.

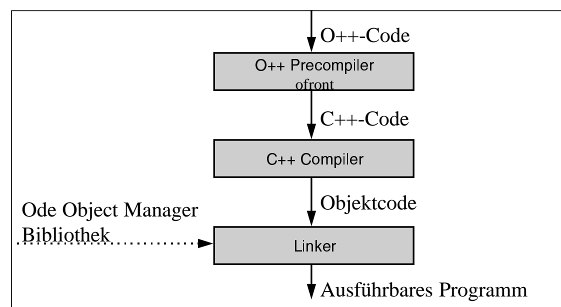


Abbildung 4.5: Der Precompiler ofront

Um O++-Programme zu übersetzen wird der Precompiler ofront bereitgestellt, der keine Erweiterung eines bestehenden Compilers (vgl. E) ist. Er übersetzt O++-Programme direkt in C++-Programme mit integrierten Aufrufen des Ode Object Managers. Auf diese Weise vorübersetzte Programme werden mit einem C++-Compiler übersetzt.[11] Abschließend wird die Ode Object Manager Bibliothek an den Code gebunden, so daß ein ausführbares Programm entsteht.

[10] vgl. Abbildung 4.6
[11] vgl. Abbildung 4.5

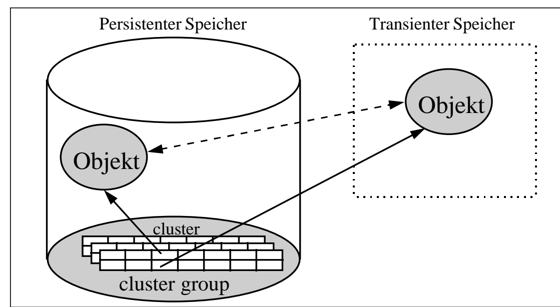


Abbildung 4.6: Der Objekttransfer in O++

Datenaustausch zwischen transientem und persistentem Speicher

Eine Anwendung kann mehrere *cluster groups* nutzen, die der Anwendung folgende Operationen zur Verfügung stellen:

- *activation*: Anlegen einer transienten Kopie eines angegebenen Objektes.
- *passivation*: Beseitigung der transienten Kopie des spezifizierten Objektes ohne die an der Kopie vorgenommenen Änderungen zu übernehmen.
- *synchronization*: Beseitigung der transienten Kopie des spezifizierten Objektes mit Übernahme aller an der Kopie vorgenommenen Änderungen.

Im Gegensatz zu den bisher dargestellten Ansätzen (mit Ausnahme der neueren Versionen von E) realisiert der Ode Object Manager ein virtuelles Speichermanagement, in dem er im Hintergrund nach Bedarf für den Aufruf der drei Funktionen *activation*, *passivation* und *synchronization* sorgt. Dadurch wird es ermöglicht, daß ein Objekt nicht sofort nach einer Modifikation zurückgeschrieben werden muß.

Um dieses Speichermanagement unabhängig von der Anwendung zu halten, muß jede persistente Klasse von der Klasse *PersBase* erben, die die Spezifikation der folgenden drei Methoden zur Verfügung stellt:

- *disksize:* gibt als Rückgabewert die Größe der persistenten Repräsentation.
- *readObj:* wird bei der Aktivierung (activation) eines Objektes aufgerufen.
- *writeObj:* wird für die Synchronisation (synchronization) eines Objektes benötigt.

Im Gegensatz zu Arjuna sind die Funktionen *disksize()*, *readObj()* und *writeObj()* nicht durch den Programmierer zu implementieren. Die erforderliche Implementation dieser Funktionen wird durch den O++-Precompiler ofront generiert.

Umgang mit Objekten

Die Programmiersprache O++ dient nicht nur dazu, persistente Objekte zu erzeugen, zu modifizieren und zu vernichten, sondern unterstützt den Implementator auch dabei, die Objekte in *cluster* zu gruppieren, assoziativ auf die Objekte zuzugreifen und über die *cluster* zu iterieren.

Um diese Funktionalität zur Verfügung zu stellen, bietet O++ zwei Schlüsselworte: *dual* und *persistent*. Beide sind vergleichbar mit den Schlüsselworten *db* und *persistent* in der Programmiersprache E. Im Gegensatz zu E, können in O++ lediglich Instanzen von *dual*-Klassen persistent gemacht werden.

Zusätzlich zu diesen Schlüsselworten bietet O++ weitere Makros, Operatoren und Funktionen:

- *copen(clustername)*: Vor jeder Objekterzeugung und jedem -zugriff muß der zugehörige *cluster* durch das Makro *copen()* geöffnet werden.
- *pnew()* dient zur Erzeugung von persistenten Objekten. *pnew* ruft zuerst den durch C++ zur Verfügung gestellten *new()*-Operator auf, der eine Repräsentation des Objektes im transienten Speicher anlegt. Zusätzlich wird ein Eintrag in dem zugehörigen *cluster* gemacht. Erst wenn der Ode Object Manager im Hintergrund die transiente Repräsentation synchronisiert (*synchronization*), wird das persistente Objekt physikalisch angelegt, der Inhalt kopiert und die transiente Repräsentation gelöscht.
- *pdelete()* dient zur Vernichtung von persistenten Objekten, indem die persistente und ggf. die transiente Repräsentation beseitigt und der zugehörige Eintrag im *cluster* gelöscht wird.

Durch Nutzung der Operatoren *pnew()* und *pdelete()* ist der Implementator nicht auf den Gebrauch von Kollektionen, wie in den neueren Versionen von E, angewiesen, um dynamisch persistente Objekte zu erzeugen. Allerdings kapseln *pnew()* und *pdelete()* den Zugriff auf *cluster*, die vergleichbar mit den Kollektionen in E sind.

Zugriff auf Objekte

Jeder *cluster* kann nur Instanzen genau einer Klasse aufnehmen. Damit enthält dieser implizit die Sammlung aller erzeugten Instanzen einer Klasse (*extent*) und kann deshalb für Anfragen an die Datenbank genutzt werden. Zu diesem Zweck bietet der Ode Object Manager sogenannte *cluster iterators* und ein in O++ nutzbares Query-Konstrukt (*for...suchthat*) an. Dieses ermöglicht die Iteration über alle Objekte eines *cluster*. Dabei wird für alle Objekte, die eine angegebene Bedingung erfüllen, eine Funktion ausgeführt.

Entwurfsentscheidungen

Der Umgang mit versteckten Zeigern wird in O++ durch den Aufruf der Funktionen *readObj()* bzw. *writeObj()* umgangen. Wird eine transiente Repräsentation eines persistenten Objektes angelegt, so wird zunächst der *new()*-Operator genutzt, um ein leeres Objekt mit gültigen versteckten Zeigern anzulegen. Erst in einem zweiten Schritt wird das Objekt rekursiv attributweise mit Werten gefüllt.

Bei der Synchronisation der transienten und persistenten Repräsentation des Objektes wird der umgekehrte Weg beschritten. Zuerst werden durch Aufruf von *writeObj()* die Werte aller Attribute rekursiv im persistenten Speicher gesichert und anschließend der *delete()*-Operator aufgerufen, um die transiente Repräsentation zu löschen.

Weiterführende Arbeiten

Wie auch bei E sind unterschiedliche Versionen von O++ entstanden. In neuen Versionen wurde der Ode Object Manager so stark vereinfacht, daß die Funktionen *disksize()*, *readObj()* und *writeObj()* für die Durchführung des virtuellen Speichermanagements nicht mehr erforderlich sind. Dieses führte allerdings dazu, daß der Object Manager kein sicheres Typecasting mehr zur Verfügung stellt.

4.3.4 Texas

Einleitung

Texas wurde Anfang der neunziger Jahre von Vivek Shingal, Sheetal V. Kakkad und Paul R. Wilson an der Universität von Texas in Austin entwickelt und für die Programmiersprache C++ implementiert. Es wurde das Ziel verfolgt, ein *objektorientiertes persistentes Speichersystem* zu entwickeln, das folgenden Entwicklungskriterien genügt ([Kak+92]):

- Das Konzept der Persistenz soll dem Programmierer verborgen sein.
- Anstatt neue Sprachkonstrukte einzuführen, sollen Klassenbibliotheken und Link-Module entworfen werden.
- Das persistente System soll performant sein.

Der Programmierer soll sich nicht um das Laden und Speichern von Objekten kümmern (vgl. Arjuna) und keine neue Programmiersprache erlernen (vgl. E, O++).

Erweiterung der Programmiersprache

Die Hauptkomponenten von Texas sind:

- *der Heap-Manager*
  Der Heap-Manager kapselt den Persistenzmechanismus. Er ist für das Ein- und Auslagern von Objekten zuständig und bedient sich der Funktionalität des *Mapping & Swizzling Moduls* und des *Storage & Recovery Managers*.

- *das Mapping & Swizzling Modul*
  Das Mapping & Swizzling Modul stellt Funktionen zur Konvertierung von Zeigern und Abbildung von Objektrepräsentationen zwischen dem transienten und dem persistenten Speicher bereit.

- *der Storage & Recovery Manager*
  Der Storage & Recovery Manager stellt eingeschränkte Datenbankfunktionalität zur Verfügung. Er organisiert den persistenten Speicher und stellt die *Konsistenz* zwischen dem transienten und dem persistenten Speicher sicher.

Die Komponenten stehen im wesentlichen orthogonal zueinander und umfassen nur einige tausend Source-Code-Zeilen. Zusätzlich wird ein Precompiler bereitgestellt.

Solange nur mit C++-Anweisungen Applikationen entworfen werden, macht Texas dem Programmierer keine Einschränkungen. Lediglich Zeigerfelder in Union-Datenstrukturen, wie sie in C generiert werden können, kann Texas nicht verarbeiten.

Alte Programme können nur dann von persistenten Objekten Gebrauch machen, wenn ihr Source-Code um die Implementation von *Root-Objekten* erweitert wird.

Anbindung von persistenten Speichern

Als persistenter Speicher wird das UNIX-Dateisystem, eigentlich nur eine einzelne *UNIX-Datei*, genutzt. Dieses stellt für das entwickelte Konzept, Objekte auch über einen Programmlauf verfügbar zu halten, keine Restriktion dar. Die Unix-Datei ist als *B-tree* organisiert. Die Blätter des Baumes repräsentieren die persistent abgelegten Speicherseiten (auch als *Datenblöcke* bezeichnet). Die Knoten des Baumes stellen den Index auf diese dar. Abbildung 4.7 stellt die Organisation einer einfachen Datei dar.

Umgang mit Objekten

Der Lösungsansatz von Texas soll zunächst durch ein Beispiel motiviert werden.

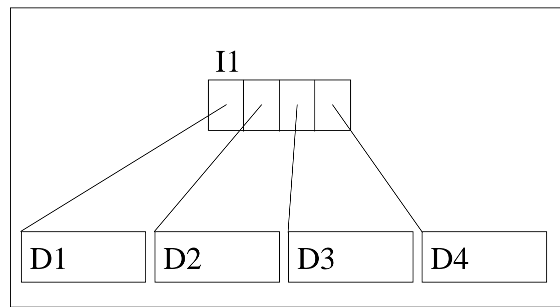


Abbildung 4.7: Eine b-tree indizierte Datei

```
(1) ...
(2) Notiz* pNotiz;
(3) ...
(4) pNotiz = new Notiz;
(5) pNotiz->GibInhalt();
(6) delete pNotiz;
```

Texas stellt dem Programmierer keine neuen Schlüsselworte für die Programmiersprache C++ bereit. Sowohl statische (Zeile 2) als auch dynamische Objekte (Zeile 4 bis 6) werden mit Hilfe der gewöhnlichen C++-Konstrukte erzeugt, manipuliert und gelöscht. Der programminterne Ablauf des compilierten Beispiels kann wie folgt beschrieben werden:

> Ein dynamisches Objekt vom Typ *Notiz* wird auf dem Heap erzeugt. Die Speicheradresse, ab der die Notiz abgelegt ist, wird in dem statischen Zeiger *pNotiz* als Referenz auf das Objekt vom Typ *Notiz* abgelegt. Nachfolgend wird die Operation *GibInhalt()* aufgerufen, in dem das Objekt vom Typ *Notiz* über den Zeiger *pNotiz* dereferenziert wird. Abschließend wird die Referenz *pNotiz* gelöscht und der durch das Objekt belegte Speicherbereich wieder freigegeben.

Auf den ersten Blick stellt sich der Programmablauf einer Texas-Anwendung wie der einer herkömmlichen C++-Anwendung dar. Die Neuerungen von Texas bleiben dem Programmierer verborgen. Sie sind mit dem *Heap* gekapselt.

> **Begriff 4 :** *Der Heap*
> Ein *Heap* ist ein Speicherbereich, der zur Laufzeit einer Anwendung als transienter Speicher für dynamisch erzeugte Objekte zur Verfügung steht.

Alle dynamischen Objekte werden auf dem Heap verwaltet. Wird eine Anwendung beendet, sei es durch einen Fehler, der die Anwendung abrupt terminiert,

oder durch ein normales Programmende, gehen die Objekte verloren und stehen bei einer erneuten Ausführung dieser Anwendung nicht mehr zur Verfügung.

Texas stellt einen neuen *Heap-Manager* bereit, der den von C++ ersetzt. Er ermöglicht die persistente Verwahrung der Objekte des Heap, in dem er sich Konzepte des *virtuellen Speichers* zueigen macht. Unter einem virtuellen Speicher verstehen wir:

> **Begriff 5 :** *Der virtuelle Speicher*
> Ein *virtueller Speicher* beschreibt einen *Speicherraum*, der von einem *Vektor* (Zeiger, z.B. der Länge 64Bit) aufgespannt wird und auf den direkt zugegriffen werden kann. Der Speicherraum besteht aus den Komponenten *Primärspeicher* (Hauptspeicher) und *Sekundärspeicher* (Festspeicher: Dateien oder Datenbanken). Der Speicherraum wird in *Speicherseiten* gleicher Größe aufgeteilt, die mit der Methode des *Paging* zwischen dem Primär- und dem Sekundärspeicher ausgetauscht werden.

Der Sekundärspeicher (UNIX-Datei) könnte nicht nur als persistenter Speicher dienen, sondern auch dem *virtuellen Speichermanager* des Betriebssystems als Instanz zum Auslagern von Speicherseiten aus dem Hauptspeicher zur Verfügung stehen. Die Erörterung der Vor- und Nachteile, die bei einer Verflechtung des virtuellen Speichermanagers und eines Heap-Managers zu einer Instanz entstehen, ist nicht Bestandteil dieser Arbeit.

Der Heap-Manager kapselt die Funktionalität von Texas:

- Alle Objekte, die zur Laufzeit nicht vom Heap gelöscht sind, werden durch das Paging auf den persistenten Speicher abgelegt und stehen einer erneuten Ausführung der Anwendung zur Verfügung.

- Obwohl im Hauptspeicher die Heapgröße begrenzt ist, wird diese nur durch die Länge der C++-Zeiger, die einen Speicherraum aufspannen, beschränkt.

- Zwischen transienten und persistenten Objekten wird nicht unterschieden (transparente Persistenz). Zugriffe auf persistente Objekte bleiben der Anwendung verborgen. Wird ein Objekt, das sich nicht im Hauptspeicher befindet, referenziert, wird die Speicherseite, auf der sich das Objekt befindet, durch das Paging in den Heap geholt.

- Durch *Checkpoints* werden Speicherseiten, auf denen sich veränderte Objekte befinden, auf dem persistenten Speicher gesichert. Dieser Speicher hat immer einen konsistenten Zustand.

Die Persistenz von Objekten ist nicht von dem Typ eines Objektes abhängig (orthogonale Persistenz). Wissen über die Struktur und den Typ von Objekten benötigt lediglich das Mapping & Swizzling-Modul. Alle anderen Komponenten von Texas behandeln Objekte bzw. Speicherseiten mit Objekten als uninterpretierte *Datenblöcke*.

Repräsentationen von Objekten

Die Identität von Objekten in Texas ist durch eine eindeutige (individuelle) Adresse im persistenten Speicher gegeben. Sie hat die Gestalt und die Länge (z.B. 64Bit) eines C++-Zeigers.[12]

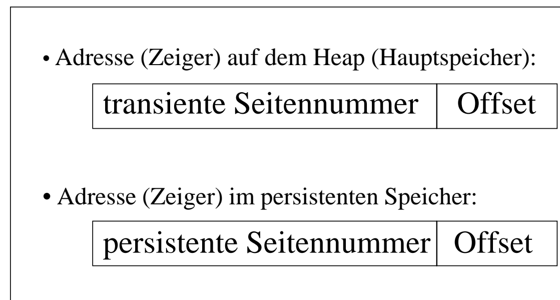


Abbildung 4.8: Das Zeigerformat von Texas

Beim Paging wird die persistente Seitennummer durch die transiente Seitennummer (Adresse einer Speicherseite des Heaps) substituiert, d.h. der Offset bleibt erhalten und nur die höherwertigen Bits werden ausgetauscht.

Datenaustausch zwischen transientem und persistentem Speicher

Die Idee des *pointer swizzling*[13] *at page fault time* entstand Ende der siebziger Jahre bei der Entwicklung von Persistent Algol.

> **Begriff 6 :** *pointer swizzling*
> *Swizzling* bezeichnet das Ersetzen von *Objektidentifikatoren* durch Adressen. Unter *pointer swizzling* wird die Konvertierung von einem Zeigerformat in ein anderes verstanden.

In unserem Kontext bedeutet pointer swizzling die *Substitution* einer persistenten Seitennummer durch die Adresse einer Speicherseite und umgekehrt.

[12] vgl. Abbildung 4.8
[13] [englisch]: Umrühren alkoholischer Mixgetränke

Bei herkömmlichen pointer swizzling Systemen muß bei jeder Dereferenzierung eines Zeigers geprüft werden, ob dieser bereits in eine Hardwareadresse geswizzled wurde. Dadurch entstehen kontinuierliche Laufzeitkosten. Ferner muß der Compiler mit unterschiedlichen Zeigerformaten umgehen können.

Texas konvertiert Zeiger bereits dann, wenn Speicherseiten in den Hauptspeicher geladen werden. Zeiger, die Objekte auf dem Heap referenzieren, werden mit den aktuellen Hauptspeicheradressen belegt. Für Zeiger, die noch nicht geladene Objekte referenzieren, werden Speicherseiten auf dem Heap reserviert und mit einem Zugriffschutz[14] versehen. Obwohl die Speicherseite noch nicht geladen wird, erhält der Zeiger die zukünftig gültige Hauptspeicheradresse.

Wenn das Nachladen einer Speicherseite zuviele Reservierungen von weiteren Seiten nach sich zieht, so daß der Heap nicht alle Reservierungen vornehmen kann, muß der Heap komplett neu aufgebaut werden. Um diesen hohen Aufwand zu vermeiden, empfehlen die Entwickler, Texas um einen periodischen *Remapping Algorithmus* zu erweitern, der nicht genutzte Seiten auslagert.

Eine Speicherseite wird erst dann in den transienten Speicher transferiert (lazy-loading[15]), wenn ein Zeiger auf eine zugriffgeschützte Speicherseite dereferenziert wird. Durch den Zugriffschutz wird eine *Ausnahme (exception)* ausgelöst. Die *Ausnahmebehandlung (protection fault handler)* lädt die benötigte Seite, sucht alle Zeiger und swizzled sie.

Zeiger in Objekten können mit Hilfe von Meta-Wissen aufgefunden werden. Im Unterschied zu Programmiersprachen wie Smalltalk, Eiffel und Modula-3, bei denen Wissen über die Gestalt und das Aussehen von Objekten (*Meta-Objekt-Daten*) automatisch generiert wird und zur Laufzeit von Applikationen verfügbar ist, schreibt der C++-Standard das Anlegen von Meta-Objekt-Daten nicht vor.[16] Damit Texas unabhängig von *Typinformationen* eines Compilers ist, stellen die Entwickler einen Precompiler und eine Modifikation der malloc-*Anweisung*[17] bereit. Der Precompiler scannt den Source-Code und erzeugt für jede definierte

---

[14]Texas nutzt den Zugriffschutz des virtuellen Speichermanagements

[15]Strategien beim Nachladen von Speicherseiten:

1. frühes Laden (eager-loading) Eine Speicherseite wird bereits dann in den Hauptspeicher geladen, wenn Referenzen auf sie zeigen.
2. spätes Laden (lazy-loading) Eine Speicherseite wird im Hauptspeicher reserviert und erst dann geladen, wenn Zeiger auf diese Seite dereferenziert werden.

[16]Einige C++-Compiler, so z.B. GNU C++, erzeugen Meta-Objekt-Daten, das keinem Standard folgt.

[17]*malloc()* ist eine C-Anweisung, die dynamischen Speicher alloziert.

Klasse ein *Typobjekt*, das die Typbeschreibung (Typ-Descriptor-Prinzip) enthält. Wird ein Objekt dynamisch erzeugt, fügt *malloc()* einen Zeiger, der auf das Typobjekt verweist, in den Kopf des Objektes ein.

Mit Hilfe der Meta-Objekt-Daten und des Wissens, daß Objekte direkt hintereinander auf einer Speicherseite gehalten werden, lassen sich alle Objekte identifizieren und damit alle Zeiger auffinden.

Zum Auffinden bereits geladener persistenter Seiten im transienten Speicher wird die *Mapping Tabelle* genutzt, die eine eindeutige Zuordnung zwischen diesen Seiten herstellt. Ihre Einträge halten die korrespondierenden Seiten der beiden Speicher fest.[18]

persistente Seitennummer	transiente Seitennummer
...	...
...	...

Abbildung 4.9: Mapping Tabelle

Durch die Mapping Tabelle wird sichergestellt, daß keine Seite doppelt in den transienten Speicher geladen wird und daß eine persistente Seite nur von ihrer zugehörigen Seite im transienten Speicher überschrieben wird.

Die Mapping Tabelle ist relativ klein, weil nur ein Eintrag pro Seite anstatt einem Eintrag pro Objekt (vgl. *cluster* bei O++) gemacht werden muß.

Ein Beispiel soll den Vorgang des pointer swizzling at page fault time und die oben erörterten Sachverhalte demonstrieren:

> Ein Objekt A, das sich im transienten Speicher befindet, referenziert zwei weitere Objekte B und D, für die bereits zwei Speicherseiten reserviert und diese mit einem Zugriffschutz versehen worden sind (vgl. Abbildung 4.10). Wird der Zeiger auf das Objekt B dereferenziert, wird eine Ausnahme ausgelöst, die Speicherseite mit dem Objekt B in den transienten Speicher nachgeladen. Alle Zeiger dieser Seite werden geswizzled und eine Seite reserviert und geschützt, weil Objekt B ein Objekt C auf einer noch nicht geladenen Seite referenziert (vgl. Abbildung 4.11).

[18] vgl. Abbildung 4.9

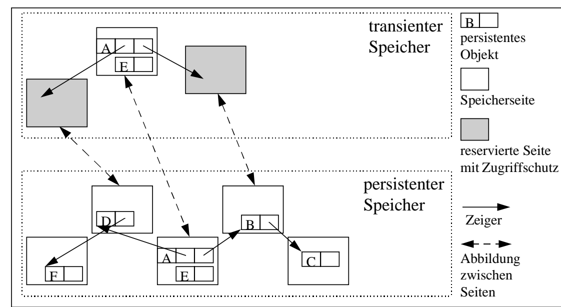


Abbildung 4.10: Reservierung von Speicherseiten

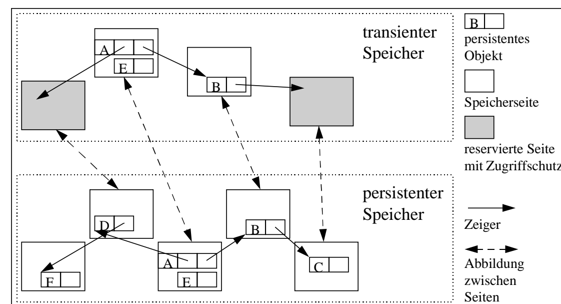


Abbildung 4.11: Dereferenzierung eines Zeigers

Abschließend sei noch erwähnt, daß die *Zeiger-Semantik* und damit die Identifizierbarkeit der Objekte aufgrund der Zeigerrepräsentation und des pointer swizzling nicht verloren geht.

Erhaltung der Konsistenz

Speicherseiten im transienten Speicher, die veränderte Objekte enthalten, werden in regelmäßigen Abständen (zum Zeitpunkt eines *Checkpoints*) in die oben beschriebene Unix-Datei zurückgeschrieben. Dabei werden nicht die alten persistenten Seiten überschrieben, sondern freie Bereiche der Unix-Datei zur Ablage genutzt. Damit die aktuellen Seiten im persistenten Speicher vom *Wurzelknoten* und damit für spätere Zugriffe verfügbar sind, werden rekursiv die Elternknoten der Datenblöcke bis hin zur Wurzel neu erstellt. D.h., der Index des Baumes wird aktualisiert. Der alte Index wird nicht überschrieben. Abbildung 4.12 veranschaulicht einen Vorgang, bei dem die Speicherseiten D2 und D4 verändert wurden.

Der Umgang mit Speicherseiten ist besonders dann performant, wenn nicht nur ein Objekt je Seite verändert wurde. Das ergibt sich aus der Tatsache, daß nicht

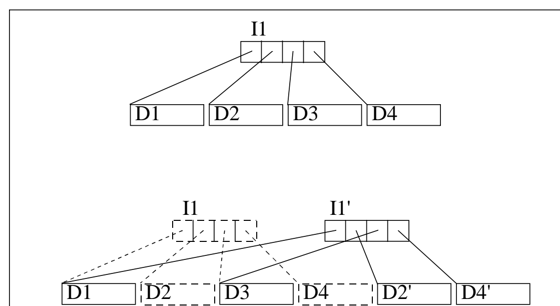


Abbildung 4.12: Eine b-tree indizierte Datei

jedes Objekt überprüft werden muß, ob es sich geändert hat, sondern nur jede Seite. Veränderungen an Objekten werden durch eine Marke auf ihrer Speicherseite angezeigt.

Damit Texas auch bei geringfügigen Änderungen je Speicherseite performant ist, schlagen die Entwickler die Implementation des *sub-page logging* vor. Dieser Mechanismus schreibt die Änderungen, die auf einer Seite stattfinden, (anstatt ganzer Seiten) in ein Logbuch. Bei einem Checkpoint werden dann die Originalseiten mit den notierten Änderungen verglichen und der persistente Speicher durch neue Seiten aktualisiert. Die Entwickler sind der Auffassung, daß das sub-page-logging zunehmend attraktiver wird ([Kak+92]). Die Nachteile dieses Verfahrens

- das Halten einer Originalseite und einer veränderten Seite (dirty page) im Hauptspeicher und
- der Aufwand des Vergleichens

würden in der Zukunft durch das schnellere Wachstum der CPU-Geschwindigkeit im Vergleich zu den Plattenzugriffzeiten mindestens aufgewogen.

Weitere Merkmale von Texas

- *Speicherrückgewinnung*
  Datenblöcke im persistenten Speicher, die nur von einem alten Index aus erreicht werden können, sind *Speichermüll*. Diese Seiten können schnell ausfindig gemacht und gelöscht werden.
- *Historienverfolgung*
  Anstatt Speicherseiten, die nicht vom aktuellen Index adressiert werden, zu löschen, können sie auch zur Verfolgung von Veränderungen an Objekten (*Historienverfolgung*) genutzt werden.
- *Behandlung großer Objekte*
  Objekte, die größer als eine Speicherseite sind, werden auf mehrere Seiten, die markiert werden und in einer Tabelle registriert werden, aufgeteilt.

- *Umgang mit Hidden Pointers*
  Beim *pointer swizzling at page fault time* werden mit Hilfe der *Typobjekte* (vgl. Type-Descriptor-Prinzip) die versteckten Zeiger gefunden und auf aktuell gültige Zeiger gesetzt.

Offene Probleme in Texas und Forschungsziele

1. *Garbage Collection*: Auffinden von nicht referenzierten Objekten sowohl im persistenten als auch im transienten Speicher.
2. Suchen von Objekten z.B. mit Hilfe von *SQL-ähnliche Anfragen*
3. Verteilung

Darüberhinaus konnte den Veröffentlichungen nicht genau entnommen werden, wie bei einem erneuten Start einer Applikation mit Hilfe von Root-Objekten auf persistente Objekte zugegriffen werden kann (vgl. [Kak+92]).

Verwandte Arbeiten

ObjectStore ist ein kommerzielles *objektorientiertes Datenbankmanagement System*, das von der Firma ObjectDesign, Inc. zeitgleich zu Texas entwickelt wurde. Es bietet eine Schnittstelle zur Anbindung beliebiger Datenbanken und arbeitet nur mit dem AT&T C++-Compiler cfront zusammen. Obwohl nur wenige technische Details in Erfahrung zu bringen sind ([Kak+92]), bedienen sich Texas und ObjectStore ähnlicher Mechanismen beim Umgang mit Objekten:

- Konzepte des virtuellen Speichermanagements
- pointer swizzling at page fault time
- Type-Descriptor-Prinzip

4.4 Übersicht der persistenten C++-Erweiterungen

Um eine Vergleichbarkeit der persistenten C++-Erweiterungen herzustellen, greifen wir die Problemstellungen aus der Einleitung auf und ordnen den Problemstellungen Merkmale, die zunächst erläutert werden, zu. Die einzelnen Ausprägungen der Merkmale von Arjuna, E, O++ und Texas werden abschließend in einer Tabelle gegenübergestellt.

1. *Erweiterung der Programmiersprache*
   - **Entwicklungsziel** gibt an, welches Entwicklungsziel dem Ansatz zugrunde liegt.
   - **Erweiterung** nennt die für die Persistenz erforderlichen zusätzlichen Komponenten.
   - **Garbage collection** gibt an, ob eine Speicherrückgewinnung in den Ansätzen realisiert ist.
2. *Umgang mit Objekten*
   - **Transparenz** gibt an, ob zwischen persistenten und transienten Objekten nicht unterschieden wird.
   - **Orthogonalität** gibt an, ob Objekte beliebigen Typs persistent gemacht werden können.
   - **Adressierung** beschreibt die Realisierung von Referenzen im Source-Code auf persistente Objekte im persistenten (PS) und transienten Speicher (TS).
   - **Identitätsrepräsentation** beschreibt das Aussehen eines persistenten Objektes im persistenten (PS) und transienten Speicher (TS).
   - **Fundamentale Typen** gibt an, ob nur Objekte oder auch Standardtypen, wie *int*, *char* usw., persistent gemacht werden können.
3. *Repräsentation von Objekten*
   - **Komplexe Objekte** beschreibt die Handhabung komplexer Objekte.
   - **Große Objekte** beschreibt die Handhabung großer Objekte.
4. *Datenaustausch zwischen transientem und persistentem Speicher*
   - **Transfer** beschreibt den Mechanismus zum Austausch persistenter Objekte zwischen dem persistenten und transienten Speicher.
   - **Granularität** beschreibt die Granularität des Datenaustausches zwischen dem transienten und dem persistenten Speicher.

- **Dereferenzoverhead** gibt die Extrakosten bei der Ausführung einer Dereferenzoperation an.
- **Zuordnungsinstanz** nennt die Instanz, die die Zuordnung zwischen transienter und persistenter Repräsentation persistenter Objekte vornimmt.
- **Meta-Objekt-Wissen** beschreibt den Ort und die Erzeugung der Meta-Objekt-Daten.

5. *Erhaltung der Konsistenz*

   - **Robustheit** gibt die Fehleranfälligkeit bezüglich der Konsistenz der persistenten Daten an.

6. *Anbindung von persistenten Speichern*

   - **Persistenter Speicher** nennt den Speicher, der zur Verwahrung der persistenten Objekte genutzt wird.

7. *Zugriff auf Objekte*

   - **Objektsuche** gibt an, ob ein Objektzugriff per Assoziation, Navigation oder per Query möglich ist.

8. *Skalierbarkeit des Ansatzes*

   - **Skalierbarkeit (Adreßraum)** gibt die Veränderbarkeit der Größe des Adreßraumes an.
   - **Skalierbarkeit (Verteilung)** gibt an, welche Einschränkungen hinsichtlich des Einsatzes in einem verteilten System existieren.

9. *Grundsätzliche Eigenschaften*

   - **Kompatibilität** gibt an, ob der Ansatz kompatibel zu bestehenden Klassenbibliotheken und Implementationen ist.
   - **Vererbung** gibt an, ob eine einfache oder mehrfache Vererbung möglich ist.
   - **Portabilität** beschreibt die Abhängigkeit von einem bestimmten Betriebssystem (B) und/oder Compiler (C).
   - **Performanz** nennt die Geschwindigkeit der Ansätze bezüglich der Persistenz im Vergleich zueinander.
   - **Verfügbarkeit** gibt das Nutzungsrecht der Implementation des Ansatzes an.

	Arjuna	*E*	*O++*	*Texas*
Entwicklungsziel	Programmiersprache für Verteilung und Fehlertoleranz	Programmiersprache für Datenbankanwendungen	Programmiersprache für Datenbankanwendungen	persistenter Objektspeicher
Erweiterung	Klassenbibliothek	neuer Compiler Schlüsselworte	Precompiler Schlüsselworte	Precompiler Heap-Modul
Garbage collection	nein	nein	nein	nur im persistenten Speicher
Transparenz	nein	nein	nein	ja
Orthogonalität	nein	nein	nein	ja
Adressierung	PS: UNIX-Dateiname TS: C++-Referenz	PS,TS: analog einer C++-Referenz	PS,TS: analog einer C++-Referenz	PS,TS: C++-Referenz
Identitätsrepräsentation	siehe Adressierung	PS: StorageOID (+StorageOffset) TS: Speicheradresse	PS: ObjektID TS: Speicheradresse	C++-Pointer als PID (z.B.64Bit) im PS, als Speicheradresse im TS
Fundamentale Typen	nein	ja	nein	ja
Komplexe Objekte	aufgrund natürlicher Modellierung keine komplexen Objekte	Objekt wird partiell behandelt, (partielles Nachladen)	Objekt wird partiell behandelt, (partielles Nachladen)	Objekt wird partiell behandelt, (partielles Nachladen)
Große Objekte	keine besondere Behandlung	keine besondere Behandlung	keine besondere Behandlung	Registrierung in Tabellen, wenn Objekt größer als Speicherseite
Transfer	load/store	load/store	indirekt durch Konzept des virtuellen Speichers	indirekt durch Konzept des virtuellen Speichers
Granularität	Attribut	Objekt	Objekt	Speicherseite

Tabelle 4.1: Übersicht der Auswertung

	Arjuna	*E*	*O++*	*Texas*
Dereferenz-overhead	aufgrund des natürlichen Objektmodells nicht bewertbar	Swizzling zwischen persistenter und transienter Adresse bei jeder Dereferenzierung (worst case)	prinzipiell kein	prinzipiell kein
Zuordnungs-instanz	Programmierer	im compilierten Code	cluster group als UNIX-Datei	Mapping Tabelle
Meta-Objekt-Wissen	muß für jede persistente Klasse implementiert werden	im EXODUS Storage Manager	wird für jede persistente Klasse durch Precompiler generiert	Precompiler erzeugt Typobjekte, auf die die Objekte verweisen
Robustheit	geschachtelte Transaktionen	nur kurze flache Transaktionen	nur flache Transaktionen	lange flache und kurze flache Transaktionen
Persistenter Speicher	UNIX-Dateisystem	durch EXODUS Storage Manager gekapselter Speicher	durch Ode Object Manager gekapselter Speicher	UNIX-Datei
Objektsuche	Assoziation keine Navigation keine Query	Assoziation Navigation keine Query	Assoziation Navigation Query über Cluster	Assoziation Navigation keine Query
Skalierbarkeit (Adreßraum)	beliebig	durch EXODUS Storage Manager beschränkt	durch Ode Object Manager beschränkt	beliebig, nach oben begrenzt durch Zeigerlänge der Hardware
Skalierbarkeit (Verteilung)	beliebig, vgl. Adressierung	keine	keine	Netzwerk muß einen Adreßraum aufspannen, sonst keine
Kompatibilität	nein	ja, aber keine Persistenz	ja, aber keine Persistenz	ja
Vererbung	einfach	mehrfach	mehrfach	mehrfach
Portabilität	B: UNIX C: unabhängig	B: unabhängig C: AT&T C++	B: UNIX C: unabhängig	B: UNIX C: unabhängig
Performanz	nicht einzuordnen	gut	besser	am besten
Verfügbarkeit	frei	frei	frei?	frei

Tabelle 4.2: Übersicht der Auswertung (Fortsetzung)

4.5 Kritische Würdigung und Forderungskatalog

4.5.1 Die Problematik der Bewertung

Im Gegensatz zur kritischen Würdigung der Materialmanagementsysteme ist die Bewertung persistenter Spracherweiterungen schwierig. Die Entwickler bewegen sich mit ihren Zielvorgaben innerhalb eines mehrdimensionalen Raumes, der durch eine Reihe sich beeinflußender und teilweise ausschließender Ziele aufgespannt wird. Im folgenden soll eine kleine Auswahl der in der bearbeiteten Literatur genannten Ziele vorgestellt werden, deren Spannungsfeld in Abbildung 4.13 grafisch dargestellt ist.

1. **Erweiterung der Programmiersprache**
   (a) Einfache Implementation von Anwendungssystemen.
   (b) Leichte Erlernbarkeit und damit kurze Einarbeitungszeit in die neue persistente Programmiersprache.
   (c) Keine Änderung der Sprachsyntax, aber Überlagerung der Sprachsemantik.
   (d) Keine Änderung der Sprachsemantik, aber Erweiterung der bestehenden Sprachsyntax.
   (e) Kompatibilität zum ANSI-C++-Standard.
   (f) Bereitstellung von OO-Konzepten wie Mehrfachvererbung, Polymorphie usw.
   (g) Keine Nutzung von ausschließlich in C++ realisierten Konzepten (z.B. Überladen von Operatoren).

2. **Umgang mit Objekten**
   (a) Kontrolle der Persistenzmechanismen durch den Programmierer.
   (b) Konzept der Persistenz dem Programmierer verbergen (transparente Persistenz).
   (c) Bereitstellung orthogonaler Persistenz.
   (d) (Partielle) Handhabung beliebig großer und komplexer Objekte.

3. **Repräsentation von Objekten**
   (a) Referenzmechanismus (außer hidden pointers) persistenter Objekte analog zu C++.

4. **Datenaustausch zwischen transientem und persistentem Speicher**
   (a) Performanz des Datentransfers.
   (b) Sicherheit des Datentransfers.
   (c) Objekte zum Zwecke eines schnelleren Zugriffs solange wie möglich im transienten Speicher halten.
   (d) Automatische Generierung der Meta-Objekt-Daten.
   (e) Bindung von Meta-Objekt-Daten an die zugehörigen Objektklassen.
   (f) Meta-Objekt-Daten gebündelt an einer Stelle organisiert.

5. **Erhaltung der Konsistenz**
   (a) Robustheit bzgl. der Datenkonsistenz und -integrität.
   (b) Transaktionslänge sowohl vom Programmierer als auch durch den Anwender frei gestaltbar.
   (c) Geringe Anzahl von Zugriffssperren.
   (d) Keine Sperrung ungenutzter Daten.
   (e) Individuelle Gestaltung der Checkpoint-Granularität durch den Programmierer in Abhängigkeit der zu entwickelnden Anwendung.

6. **Anbindung von persistenten Speichern**
   (a) Unabhängigkeit von bestimmten persistenten Speichern.

7. **Zugriff auf Objekte**
   (a) Anfragen gegen den persistenten Speicher über den Inhalt von Objekten.
   (b) Entwicklung einer Anfragesprache unabhängig von den persistenten Speichern.
   (c) Zugriff auf Objekte sowohl per Navigation, Assoziation und Anfrage.

8. **Skalierbarkeit des Ansatzes**
   (a) Skalierbarkeit für zukünftige Entwicklungen.

9. **Grundsätzliche Eigenschaften**
   (a) Portabilität.
   (b) Keine Entwicklung neuer Komponenten (z.B. Precompiler) für Programmierumgebungen.

(c) Keine Veränderungen an bestehenden Komponenten einer Programmierumgebung.

(d) Keine Unterscheidung des persistenten und transienten Speichers.

Jedes Entwicklungsteam nimmt einen anderen Punkt innerhalb des durch die Ziele aufgespannten Raumes ein, so daß eine Bewertung der Ansätze ohne einen eigenen Referenzkatalog von Zielen nicht möglich ist.

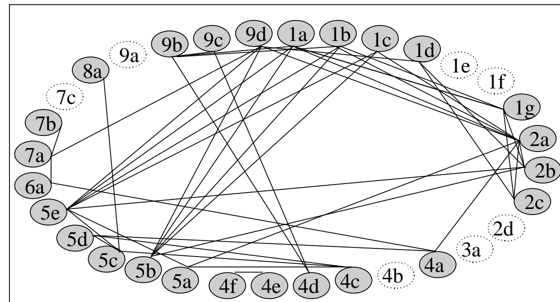


Abbildung 4.13: Darstellung von Konfliktmöglichkeiten der Zielsetzungen für ein optimales OMeS (multidimensionales Spannungsfeld)

Der Entwurf eines Objektmanagementsystems wird stark durch die konkurrierenden Ziele beeinflußt. Daß der Versuch, von jedem Ansatz nur die subjektiv positiven Konzepte zu nehmen und diese in einem alternativen Vorschlag zu vereinen, fehlschlagen muß, wollen wir an einem kleinen Beispiel darstellen:

Ohne Zweifel ist es wünschenswert, daß die Persistenz transparent ist und damit die Persistenzproblematik vor dem Programmierer verborgen wird, um eine einfache, leicht erlernbare und damit besser akzeptierte Programmiersprache zu erhalten. Auf der anderen Seite wird uns der Leser rechtgeben, daß es sinnvoll ist, nicht alle Daten a priori als persistent anzunehmen und die Lebensdauer der Daten individuell durch den Programmierer bestimmen zu lassen. Damit müßten diesem aber Sprachkonstrukte zur Verfügung gestellt werden, die der Transparenz widersprechen würden. Die gleichzeitige Erreichung beider Ziele ist somit offensichtlich nicht möglich.

Schon dieses kleine Beispiel zeigt, daß sich zwei Ziele, die einzeln betrachtet als wünschenswert eingestuft werden, sich in der Realisierung ausschließen können. Aus diesem Grund wollen wir uns durch die Ideen der vorgestellten Ansätze leiten lassen und im nächsten Unterabschnitt einen widerspruchsfreien Forderungskatalog erstellen. Anschließend können wir mit Hilfe dieses Kataloges die Mängel der einzelnen Ansätze zeigen.

4.5.2 Der Forderungskatalog

In diesem Abschnitt lösen wir uns von den Konzepten persistenter Programmiersprachen und nennen die Anforderungen, die unserer Meinung nach an ein Objektmanagementsystem (OMeS) zu stellen sind. Unser Hauptaugenmerk liegt dabei auf der Berücksichtigung komplexer Objekte.

Die externen Partner des OMeS

1. *Der persistente Speicher*:
   Jedes Datenbanksystem und jedes Dateisystem sollte prinzipiell als persistenter Speicher geeignet sein. Der Vorteil von Datenbanksystemen liegt u.a. in der Bereitstellung von Transaktionen zur Konsistenzsicherung.

2. *Der transiente Speicher*:
   Bevor wir unsere Forderungen bzgl. des transienten Speichers erheben, wollen wir sie durch die folgenden Überlegungen motivieren:

   Es ist festzustellen, daß

   - *die ersten Speicher der Computergeschichte persistent waren*: Charles Babbage entwickelte 1835 mit der „analytical engine (AE)" die erste programmgesteuerte Maschine der Welt[19]. Die Programme für diese wurden auf einer Kette von Lochkarten notiert, die sowohl den Algorithmus als auch die für dessen Ausführung nötigen Daten kodiert enthielten.
   - *der Einsatz transienter Speicher nicht erforderlich wäre*, wenn die persistenten Speicher eine ausreichend hohe Zugriffsgeschwindigkeit zur Verfügung stellen würden. Das noch vor wenigen Jahren bestehende Problem der Adressierbarkeit persistenter Speicher ist mit der Einführung der 32Bit- und 64Bit-Architektur hinfällig, weil damit auch die Adressierung großer Speicher[20] möglich wurde.

[19] vgl. [Hei92]

[20] 16Bit-Architektur: 64 kByte, 32Bit-Architektur: 4 GByte, 64Bit-Architektur: 16 EByte (1 EByte = 1048576 TByte)

- *ein Wechsel vom load/store-Modell*[21] *zu einem virtuellen Speichermanagement*[22] *stattgefunden hat.* So ist ein virtuelles Speichermanagement Grundlage der akzeptierten kommerziellen objektorientierten Datenbank ObjectStore.

Aus diesen Überlegungen folgern wir, daß es für ein OMeS wünschenswert ist, den transienten Speicher neben der Funktion als Programmspeicher ausschließlich als Zwischenspeicher (*Cache*) für persistente Daten zu nutzen, um das Problem der hohen Zugriffszeit des persistenten Speichers zu mildern.

Begriff 7 : *Cache (nach [Gil93])*
Ein Cache ist einem Speicher, dessen Zugriffszeit wesentlich höher und dessen Speicherkapazität wesentlich größer als die des Caches ist, zugeordnet. Der Cache enthält Kopien von Teilen dieses Speichers und muß für den Programmierer transparent sein, d.h., daß der Programmierer weder von der Existenz des Caches wissen noch sich um dessen Kontrolle kümmern muß.

Begriff 8 : *Virtueller Speicher (nach [Gil93])*
Ein virtueller Speicher (vgl. hierzu Abschnitt 4.3.4) erweitert einen Speicher, dessen Zugriffszeit wesentlich niedriger und dessen Speicherkapazität wesentlich kleiner als die des virtuellen Speichers ist. Der erweiterte Speicher enthält Kopien von Teilen des virtuellen Speichers, der für den Programmierer transparent sein muß, d.h., daß der Programmierer weder von der Existenz des virtuellen Speichers wissen noch sich um dessen Kontrolle kümmern muß.

Die Einführung eines Caches unterscheidet sich von der Nutzung eines virtuellen Speichers nach [Gil93] lediglich in der Sichtweise:

(a) *Einsatz eines virtuellen Speichers*:
Es wird eine Komponente zwischen transienten und persistenten Speicher eingefügt, um die Speicherkapazität des transienten Speichers zu vergrößern. Dabei wird eine höhere Zugriffszeit in Kauf genommen.

(b) *Einsatz eines Caches*:
Es wird eine Komponente zwischen transienten und persistenten Speicher eingefügt, um die Zugriffszeit des persistenten Speichers zu verkleinern. Dabei wird eine geringere Speicherkapazität in Kauf genommen.

[21] vgl. Arjuna und die ersten Versionen von E
[22] vgl. neuere Versionen von E, O++ und Texas

Durch die Einführung eines Caches wären somit alle Daten a priori persistent und würden nur temporär bis zu ihrer Verdrängung aus dem Cache im transienten Speicher gehalten, um damit einen schnelleren Zugriff zu ermöglichen. Die weitere Diskussion wird zeigen, daß wir diese Forderung noch relativieren, aber nicht aufgeben müssen.

Der Datentransfer

1. *Die Granularität*:
   Die vorgestellten Spracherweiterungen bieten unterschiedliche Granularitäten des Datentransfers: Arjuna transferiert Objekte attributweise, E und O++ objektweise und Texas lädt komplette Speicherseiten in den transienten Speicher.

   Im folgenden soll der attributweise Datentransfer gegen den seitenweisen abgegrenzt werden, um anschließend eine Forderung bezüglich der Granularität erheben zu können:

   - *Transferkosten*:
     Ein attributweiser Transfer aktiviert nur die gebrauchten Daten, muß dafür aber im Durchschnitt öfter angestoßen werden. Ein seitenweises Vorgehen benötigt weniger Zugriffe. Es wird dafür aber erheblich mehr gelesen als benötigt wird. Zudem können Daten in den transienten Speicher kopiert werden, für deren Benutzung der Anwender nicht authorisiert ist.
   - *Zugriffssperren*:
     Der Nachteil eines attributweisen Lesens liegt in dem hohen Aufwand, der durch das nötige Logging aller Attribute gegen zwischenzeitliche Zugriffe entsteht. Ein seitenweiser Transfer verringert zwar die Anzahl der Zugriffssperren, aber sperrt dafür mehr Daten als effektiv zu sperren sind.
   - *Transaktionslänge*:
     Aus den gemachten Ausführungen folgt, daß sich ein attributweiser Zugriff vor allem für kurze und ein seitenweiser Zugriff für lange Transaktionen eignet.

   Diese Gegenüberstellung zeigt deutlich, daß sich durch die Forderungen

   - kleine Transferkosten
   - wenige Zugriffssperren
   - beliebige Transaktionslänge

ein Zielkonflikt ergibt, der ausschließlich durch einen Kompromiß gelöst werden kann.

Ein Objektmanagementsystem sollte deshalb einen Transfer auf Objektebene durchführen[23], weil dann die Transferkosten proportional zur benötigten Datenmenge ist und Nachteile des attributweisen und des seitenweisen Transfers abgeschwächt werden können, was durch die gleichzeitige Abschwächung der Vorteile beider Ansätze erkauft wird. Diese Herangehensweise impliziert, daß der persistente Speicher als ein reiner Objektspeicher benutzt wird und Instanzen fundamentaler Typen nicht persistent gemacht werden können. Dieses stellt aber unter Zugrundelegung der objektorientierten Sichtweise keine Einschränkung der Allgemeinheit dar.

2. *Die Konsistenzsicherung*:
   Wir haben die Forderung erhoben, daß der transiente Speicher als Objektcache des persistenten Speichers genutzt werden soll. Dieses Vorgehen erfordert aber auch die Bereitstellung von Mechanismen zur Sicherung der Datenkonsistenz[24] oder zumindest der Datenkohärenz[25]. In der Literatur über Speichermanagement und Cachemechanismen (z.B: [Gil93]) finden sich insbesondere zwei verschiedene Verfahren, die den Abgleich zwischen einem Cache und dem zugehörigen persistenten Speicher organisieren:

   - *Durchschreiben (store through)* Jede Änderung im Cache zieht die unmittelbare Änderung des persistenten Speichers nach sich. Dadurch kann die Datenkonsistenz der Speicherzugriffe zugesichert werden.
   - *Rückschreiben (copy back)* Eine im Cache gemachte Änderung wird erst bei der Verdrängung des geänderten Eintrags im persistenten Speicher nachvollzogen. Durch diese Methode ist lediglich die Datenkohärenz der Speicherzugriffe gesichert.

   Bei der Einführung von Verteilung (*Mehrbenutzerbetrieb*, *Multiprocessing*) müssen *Sperrmechanismen* realisiert werden, um die Datenkonsistenz oder die Datenkohärenz zusichern zu können.

Das Objekt

1. *Die Komplexität*:
   Das komplexe Objekt soll in der Anwendung als Einheit betrachtet werden können, obwohl es sich evtl. nur partiell im transienten Speicher und damit

[23] Dieses schließt die partielle Handhabung komplexer Objekte keinesfalls aus.

[24] „Datenkonsistenz bedeutet, daß [...] zu keinem Zeitpunkt verschiedene Kopien desselben Datenobjekts existieren." ([Gil93])

[25] „Datenkohärenz bedeutet, daß beim Lesen eines Datenobjekts [...] immer der zuletzt geschriebene Wert gelesen wird." ([Gil93])

im Cache des persistenten Speichers befindet. Wenn ein bestimmter Teil des Objektes gebraucht wird, muß dieser in den transienten Speicher geholt und somit für die Anwendung zugreifbar werden. Dieses kann wie wir oben gesehen haben entweder durch frühes Laden (eager-loading) oder spätes Laden (lazy-loading) erreicht werden.

2. *Die Größe*:
   Objekte von beliebiger Größe sollten handhabbar sein, damit das OMeS z.B. für Multimedia-Anwendungen genutzt werden kann. Dazu ist es erforderlich, daß das Objektmangementsystem in der Lage ist, mit großen Objekten wie z.B. speicherintensiven Bildern umzugehen.

3. *Nutzung objektorientierter Konzepte (wie z.B. Mehrfachvererbung, Polymorphie, usw.)*:
   Im Gegensatz zu der Spracherweiterung Arjuna, in der keine Mehrfachvererbung vorgesehen ist, sollte das Objektmanagementsystem die in C++ zur Verfügung gestellten objektorientierten Konzepte voll unterstützen.

4. *Die Referenzierung*:
   In einem OMeS müssen Objekte in einfacher Weise andere Objekte referenzieren können. Um dieses Ziel zu erreichen, sollte das Objektmangagementsystem zur Referenzierung persistenter Objekte einen Mechanismus ähnlich der transienten Referenzen normaler C++-Zeiger zur Verfügung stellen.

   Wünschenswert, aber nicht zwingend erforderlich, sollte eine Adressierung sein, die die Welt als einen Adreßraum begreift und somit auch die Adressierung ferner Objekte ermöglicht. Da diese Forderung vor dem Hintergrund einer möglichen Verteilung zu sehen ist, wollen wir diese Forderung nur nennen, aber nicht weiter beleuchten.

5. *Die Identität*:
   Wir wollen die folgenden Forderungen an die Identitäten, die Identifikatoren und die Namensgebung von Objekten erheben:

   - Die Objektidentität ist diejenige Qualität eines Objektes, die es von allen anderen vom OMeS verwalteten Objekten unterscheidbar macht.
   - Die Objektidentität ist unabhängig von dem aktuellen Zustand des Objektes, vom Ort des Objektes, von allen Zustandsübergängen (Änderung der Ausprägungen der Merkmale) und von seinen evtl. Rollen.
   - Der Objektidentifikator wird vom OMeS erzeugt und kann vom Benutzer nicht geändert werden, weil er für den Benutzer unsichtbar ist.

- Ein Objektidentifikator kann zur gesamten Lebensdauer einer Instanz eines OMeS nur einmal erzeugt werden und wird höchstens einem Objekt zugeordnet. Auf diese Weise kann die eindeutige Identifizierbarkeit garantiert werden.
- Einem Objekt sollten mehrere Namen zugeordnet werden können.
- Die Namen eines Objektes sollten allein vom Benutzer vergeben werden und für das OMeS keine Semantik enthalten.
- Die Namensgebung sollte keinen Restriktionen unterliegen, so daß ein Name auch mehrfach vergeben werden darf.

6. *Der Zugriff*:
   Eine assoziative Suche sollte über die Ausprägungen von Eigenschaften eines Objektes und über benutzervergebenen Namen möglich sein.

   Die Möglichkeit per Navigation auf ein Objekt zugreifen zu können, wäre wünschenswert, aber nicht zwingend erforderlich. Wichtiger ist die Bereitstellung einer Anfragesprache, mit der eine Untermenge der Objekte im persistenten Speicher ausgewählt werden kann, auf deren Elemente ein assoziativer Zugriff ermöglicht wird.

   Bei der assoziativen Suche über einen benutzervergeben Namen und beim Zugriff mit Hilfe der Anfragesprache sollten nicht alle verfügbaren zur Anfrage passenden Objekte gesammelt und anschließend zusammen übergeben werden, sondern „web[26]-ähnlich" in der Reihenfolge und zum Zeitpunkt ihrer Entdeckung. Durch dieses Vorgehen können der anfragenden Instanz bereits während der Suche Objekte zur Verfügung gestellt werden, so daß die Suche vorzeitig für beendet erklärt werden kann, wenn die gewünschten Objekte vorliegen.

Die Persistenz

1. *Die Transparenz*:
   Die Transparenz der Persistenz sollte so groß wie möglich sein, damit der Programmierer weniger Arbeit mit dem Objekttransfer hat. Auf der anderen Seite widerspricht dieses Ziel, wie wir oben gesehen haben, dem Wunsch, dem Programmierer die Kontrolle über die Persistenz zu übertragen, weil es nicht sinnvoll sein kann, alle Objekte a priori als persistent anzunehmen.

   Es existiert also der Zielkonflikt, daß eine möglichst individuelle Gestaltung der Persistenz die Transparenz beschneidet, aber die transparente Persistenz dem Programmierer alle Möglichkeiten der Kontrolle entzieht.

[26]web steht für *World Wide Web (WWW)* und bezeichnet einen Dienst des Internets.

Der von uns favorisierte Kompromiß liegt darin, dem Programmierer die Entscheidung zu überlassen, welche Objekte durch das OMeS verwaltet werden sollen und welche nicht. Zu diesem Zweck schlagen wir vor, daß Klassen, deren Instanzen persistent sein sollen, von der Oberklasse `persistent` erben müssen. Diese Oberklasse spezifiziert und implementiert das Protokoll zur Nutzung durch das OMeS.

2. *Die Orthogonalität*:
   Der gemachte Kompromiß bzgl. der transparenten Persistenz und die bereits getroffene Entscheidung, den persistenten Speicher als einen persistenten Objektspeicher zu sehen, impliziert die in Abbildung 4.14 dargestellte Einschränkung der Orthogonalität[27].

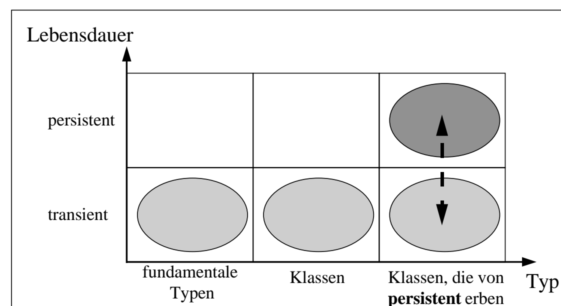


Abbildung 4.14: Forderungen an die Orthogonalität in OMeS

Die Persistenz ist orthogonal bzgl. aller Unterklassen von `persistent`, weil die Struktur, die Merkmale und deren Ausprägung keinerlei Einfluß auf die Persistenz der Instanzen dieser Klassen haben.

Die technische Realisierung der Erweiterung

1. *Art der Erweiterung*:
   Bevor wir eine Forderung aufstellen können, soll im folgenden das Spannungsfeld, in dem wir uns mit der Fragestellung bewegen, beschrieben werden:

   - *Verwendung einer nicht generischen C++-Klassenbibliothek*:
     Der Implementator muß geerbte Konzepte und Schnittstellen ausimplementieren, um die Persistenz der Objekte zu ermöglichen. Der Vorteil liegt in der größten Unabhängigkeit von der technischen Realisierung eines bestimmten Compilers.

[27]Die Orthogonalität ist nicht eingeschränkt, wenn in der objektorientierten Welt die Wertsemantik durch die Objektsemantik simuliert wird.

- *Verwendung einer generischen C++-Klassenbibliothek*:
Die Klassenbibliothek stellt nicht nur die Schnittstellen, sondern auch die Implementation der Konzepte bereit. Die Einbindung erscheint kryptisch und erfordert ein hohes Maß an Einarbeitung durch den Programmierer, nimmt dem Programmierer aber auch einen erheblichen Teil seiner Arbeit ab.

- *Einbindung eines Precompilers*:
Wie bereits weiter oben beschrieben, erfordert die Einbindung eines Precompilers die Entwicklung dieser Komponente und einen zusätzlichen Übersetzungslauf bei jeder Übersetzung eines Programmes. Dabei wird eine erweiterte Sprachsyntax ersetzt und/oder die veränderte Sprachsemantik vorübersetzt.

- *Nutzung eines Laufzeit-Interpreters*:
Ein Laufzeit-Interpreter versucht die nötigen Meta-Objekt-Daten zur Laufzeit durch Parsen der Klassenspezifikationen zu erschließen. Ein solches Vorgehen ist extrem zeitaufwendig und erfordert bestimmte Konventionen der Ablage der nötigen Daten.

- *Nutzung der Interna eines speziellen Compilers*:
Die Nutzung der Interna eines speziellen C++-Compilers (z.B. Nutzung der Meta-Object-Daten von GNU C++) erfordert eine sehr compilernahe Implementation des Objektmanagementsystems. Dadurch entsteht eine starke Abhängigkeit zwischen dem Objektmanagementsystem und dem Compiler. Die Vorteile eines solchen Vorgehens sind das Fehlen neuer Komponenten, das Entfallen eines Precompilerlaufes und der geringe zusätzliche Implementationsaufwand für den Programmierer.

Um eine möglichst hohe Entlastung des Programmierers zu erreichen und ihm ein transparentes Vorgehen zu ermöglichen, sollte das OMeS aufbauend auf den Interna eines speziellen C++-Compilers eine generische Klassenbibliothek bereitstellen. Dadurch kann eine größtmögliche Kapselung des OMeS ermöglicht werden. Eine Änderung von Persistenzmechanismen würde nur Änderungen am OMeS nachsichziehen und keine Veränderung aller persistenten Objektklassen erzwingen.

Falls die Meta-Objekt-Daten, die dem C++-Compiler entnommen werden können, nicht ausreichen, dann sollte diesem ein möglichst kleiner Precompiler vorgeschaltet werden.

2. *Zentralität der Meta-Objekt-Daten*:
Ein weiteres Spannungsfeld ergibt sich bei der Frage nach dem Ort der Verwaltung von Meta-Objekt-Daten. Eine Meinung ist, das Wissen nahe an den

zugehörigen persistenten Objektklassen zu verwalten, um die Abhängigkeit von diesen zu betonen. Ein anderer Ansatz verlangt das gesamte Wissen gebündelt an einer einzigen Stelle zentral zu organisieren, damit Änderungen am Persistenzmechanismus und dem zugehörigen Protokoll möglichst nur die Veränderung einer zentralen Komponente erfordert und nicht eine Modifikation aller Objektklassen.

Dieser Argumentation können wir uns anschließen und fordern deshalb, die Meta-Objekt-Daten zentral an einer Stelle innerhalb des OMeS zu organisieren. Ein solches Vorgehen erleichtert auch den Anschluß anderer persistenter Speicher, weil lediglich zentral an einer Stelle des OMeS Änderungen vorzunehmen sind.

3. *Portabilität*:
   Die Vorstellung der Spracherweiterungen zeigt, daß die Portabilität eine problematische Forderung darstellt. So sind alle Spracherweiterungen entweder von einem bestimmten Betriebssystem (UNIX) abhängig oder erfordern einen bestimmten C++-Compiler (AT&T C++).

   Es sollte deshalb nicht erforderlich sein, daß das Objektmangementsystem, obwohl es wünschenswert wäre, voll portabel ist. Es sollte aber auf einer etablierten und ausgereiften Plattform arbeiten und keine „exotischen" Komponenten erfordern. Eine geeignete Plattform wäre z.B. das Betriebssystem UNIX und der GNU C++-Compiler.

4. *Kompatibilität*:
   Ein Objektmangementsystem sollte möglichst wenige Erweiterungen oder Veränderungen an bestehenden Anwendungen nachsichziehen. Wünschenswert, aber wenig realistisch, wäre die unmittelbare Nutzung der durch das OMeS bereitgestellten Persistenz in diesen Anwendungen.

5. *Robustheit*:
   Ein Ergebnis der Gegenüberstellung von Spracherweiterungen ist die Erkenntnis, daß im Laufe der Jahre das Interesse an Datenbankfunktionalität gefallen ist. So finden sich für Texas nur ansatzweise Mechanismen zur Konsistenzsicherung und Regelung von konkurrenten Zugriffen, während Arjuna eine breite Palette dieser Mechanismen zur Verfügung gestellt hat.

   Ein Objektmanagementsystem sollte Konsistenzmechanismen nicht nur enthalten und diese automatisch durchführen, sondern den Programmierern und den Anwendern ein Transaktionskonzept bereitstellen.

6. *Performanz*:
   Obwohl die Akzeptanz eines Objektmanagementsystems maßgeblich von

der Performanz des Systems abhängt, sollte die Performanz nicht das Maß aller Dinge sein und das wesentliche Bewertungskriterium darstellen.

7. *Skalierbarkeit*:
   Zu wünschen wäre, daß das Objektmanagementsystem keine Einschränkung hinsichtlich der Skalierbarkeit macht. Der Adreßraum sollte zum Zwecke der Verteilung frei skalierbar sein.

   Da eine mögliche Verteilung nicht Hintergrund dieser Arbeit ist, wollen wir diese Forderung hintenanstellen, aber trotzdem nicht aus den Augen verlieren.

4.5.3 Die Bewertung der Ansätze

Für die kritische Würdigung der Ansätze kann nach unseren Überlegungen nicht die Erreichung eines einzelnen Zieles[28] ausschlaggebend sein, sondern nur die Erreichung des kompletten Forderungskataloges kann beurteilt werden. Aus diesem Grund wollen wir im folgenden kurz darstellen, daß kein vorgestellter Ansatz diesen Katalog erfüllt. Dieses soll durch Angabe der wichtigsten Abweichungen erfolgen:

Arjuna

- Kein virtuelles Speichermanagement bzw. Cachemanagement
- Keine partielle Behandlung komplexer Objekte
- Keine Mehrfachvererbung
- Dezentralität der Meta-Objekt-Daten
- Kein Objektzugriff über eine Anfragesprache
- Programmiersprache ist nicht kompatibel zu C++.

E

- Kein virtuelles Speichermanagement bzw. Cachemanagement
- Keine Konsistenzmechanismen
- Referenzierung ausschließlich über den Objektidentifikator
- Kein Objektzugriff über eine Anfragesprache
- Es ist ein Precompiler erforderlich.

[28]vgl. Abbildung 4.13

O++

- Keine Konsistenzmechanismen
- Referenzierung ausschließlich über den Objektidentifikator
- Es wird lediglich eine rudimentäre Anfragesprache bereitgestellt.
- Dezentralität der Meta-Objekt-Daten
- Es ist ein Precompiler erforderlich.

Texas

- Nur unzureichende Konsistenzmechanismen
- Eine seitenweise Datentransfergranularität
- Referenzierung ausschließlich über den Objektidentifikator
- Kein Objektzugriff über eine Anfragesprache
- Es ist ein Precompiler erforderlich.

Kapitel 5

Standortbestimmung und Ausblick

5.1 Standortbestimmung

Das Ziel dieser Arbeit war es, widerspruchsfreie Forderungen an eine Architektur zur Verwaltung persistenter Materialien zu formulieren. Dabei haben wir darauf wertgelegt, eine klare Trennung zwischen der fachlichen Seite (MMeS) und der technischen Seite (OMeS) zu erreichen.

Um Forderungen an eine PooMMeS-Architektur formulieren zu können, ist es unverzichtbar gewesen, sich mit der Evolution fachlich motivierter Materialversorgungskonzepte ([Kil+93],[Wul+94] und [Gry95]) und technisch basierter persistenter Programmiersprachen (Arjuna, E, O++, Texas) zu beschäftigen.

Die Darstellung der vorgestellten Materialversorgungskonzepte ist dabei im Gegensatz zu den persistenten Programmiersprachen unkomplizierter, weil alle Konzepte aus einer einzigen Entwicklungslinie stammen. Die persistenten Spracherweiterungen hingegen wurden an unterschiedlichen Orten und mit unterschiedlichen Zielen entwickelt, so daß sich die Chronologie als einzige Bezugsgröße als unzureichend erweist. Deshalb ist in einem ersten Schritt ein Katalog von Fragestellungen entstanden, der durch Bereitstellung von Kriterien eine vergleichbare Darstellung der unterschiedlichen Ansätze ermöglicht.

Auf diese Weise konnten die grundsätzlichen Entwicklungslinien aufgezeigt und das Fundament, auf dem die neu zu entwickelnde PooMMeS-Architektur stehen sollte, beschrieben werden. Zudem wurde durch die Darstellung der einzelnen Ansätze die Formulierung von Forderungskatalogen für ein fachlich motiviertes MMeS und ein technisch basiertes OMeS ermöglicht. Dabei wurde besonderer Wert auf die Behandlung von komplexen Objekten und Materialien ge-

legt. Die Kataloge sind ohne Zweifel noch nicht vollständig und dienen deshalb hauptsächlich als Grundlage für weitere Diskussionen.

Obwohl aus dieser Arbeit keine Architekturbeschreibung hervorgegangen ist, kann trotzdem festgestellt werden, daß die aufgestellten Forderungen die Vision einer PooMMeS-Architektur enthalten, die auf einer klaren Trennung der fachlichen und der technischen Details basiert. Genau dieses Ziel galt es mit dieser Arbeit zu erreichen.

5.2 Richtungsbestimmung für die weiterführende Arbeit

Die von uns aufgestellten Forderungen stellen lediglich eine Ideensammlung von wichtigen Entwurfentscheidungen dar, deren Realisierbarkeit Gegenstand einer weiterführenden Arbeit sein sollte. Bei der Herleitung der Forderungen haben wir mehr auf die Kausalität als auf die Einordnung in die durch Fragestellungen bereitgestellten Kriterien geachtet, so daß diese Zuordnung noch erfolgen sollte.

In der weiterführenden Arbeit muß auch eine kritische Auseinandersetzung mit den Forderungen aus dieser Arbeit und evtl. eine Erweiterung dieser erfolgen, so daß dann die Spezifikation und der Entwurf der PooMMeS-Architektur stattfinden kann, die von einer Implementation begleitet sein sollte. Dabei wird eine Verbindung zwischen dem MMeS und dem OMeS erforderlich, welche durch eine Komponente mit „Gateway"-Funktion (z.B. Oberklasse **persistent**[1]) realisiert sein könnte.

Weiterhin wäre es interessant, in der weiterführenden Arbeit die Verteilung und damit die *Mobilität* von (komplexen) Materialien und Objekten in verteilten Systemen und die mögliche Erweiterung der PooMMeS-Architektur um ein *WMeS (Werkzeugmanagementsystem)* und ein *AMeS (Automatenmanagementsystem)* zu untersuchen.

Abschließend sei erwähnt, daß noch Klärungsbedarf bei der Einordnung des Objektmanagementsystemes in die wesentlichen Konzepte zur Anbindung von persistenten Speichern an Programmiersprachen besteht. Wir haben in Kapitel 4 aufgezeigt, daß es

1. *den evolutionären* und *den revolutionären Ansatz* auf der Datenbankseite und

2. *den programmiersprachlichen Ansatz* auf der Seite der Softwaretechnik

[1] vgl. Abschnitt 4.5.2

gibt.

Nach unserem heutigen Wissensstand sind wir der Meinung, daß wir einen Standpunkt zwischen diesen beiden Sichtweisen einnehmen, weil wir bei der Realisierung eines OMeS auf der einen Seite eine Programmiersprache um Konzepte der Persistenz erweitern[2] und auf der anderen klassische Funktionalität von Datenbankmanagementsystemen[3] benutzen und/oder implementieren müssen. Um eine genaue Einordnung vornehmen zu können, müssen die nachfolgenden Fragestellungen untersucht werden:

1. Inwieweit soll ein OMeS auf bestehende Datenbankfunktionalität zurückgreifen ohne sich in die Abhängigkeit einer Datenbank zu begeben?

2. Inwieweit soll ein OMeS die Funktionalität klassischer Datenbanksysteme neu implementieren, um unabhängig von den unterschiedlichen Datenbankmanagementsystemen zu sein?

3. Kann ein OMeS von den unterschiedlichen Datenbanktechnologien abstrahieren?

Da wir uns im Grenzbereich zwischen Programmiersprachen und Datenbankmanagementsystemen bewegen, ist erst nach einer gründlichen Abwägung dieser Problemstellung zu entscheiden, ob wir in der Lage sind, eine Zwischenschicht („Middleware") zwischen der Programmiersprache C++ und den Datenbankmanagementsystemen zu realisieren.

[2]Dabei werden die speziellen Anforderungen durch komplexe Objekte berücksichtigt.
[3]Sperr-, Transaktions- und Datenrückgewinnungsmechanismen

Anhang A

Entwicklungsprozeß unserer Studienarbeit

A.1 Unser Hintergrund

Unsere Studienschwerpunkte und damit Hauptinteressen liegen

1. in der objektorientierten Softwareentwicklung (WAM, Booch, Rambaugh, Petrinetze)
2. in der Unterstützung des Implementators durch eine Programmierhilfe (dinx)
3. in der Gestaltung verteilter Systeme

Die Grundlagen wurden uns durch verschiedene Vorlesungen der Arbeitsbereiche Datenbanken und Informationssysteme (DBIS), Softwaretechnik (SWT) und Theoretische Grundlagen der Informatik (TGI) vermittelt.

Durch den positiven Verlauf des Softwareentwicklungsprojektes am Arbeitsbereich TGI im Sommersemester 1995 beeinflußt, waren wir seit Mitte 1995 auf der Suche nach einem passenden Studienarbeitsthema. Trotz vieler Ideen verzichteten wir auf das Schreiben einer Studienarbeit am Arbeitsbereich TGI, weil uns die Personalplanung und damit die Sicherstellung einer dauerhaften Betreuung zu unsicher erschien. Aus diesem Grund orientierten wir uns im weiteren Verlauf auf den Arbeitsbereich SWT.

Den endgültigen Ausschlag für die Entscheidung, diese Arbeit am Arbeitsbereich SWT zu schreiben, gab das SWT Softwareentwicklungsprojekt im Wintersemester 95/96. Hier wurde Ende Januar auf die fehlende Möglichkeit der dauerhaften Speicherung von Materialien hingewiesen und erste Ansätze zur Lösung dieses Problems kurz angerissen.

Obwohl wir über keine Vorkenntnisse im Bereich der Persistenz verfügten und als fachliche Grundlage lediglich die Studienarbeit von [Wul+94] existierte, entschieden wir uns für die Bearbeitung des Themas dieser Arbeit.

A.2 Die Zusammenarbeit

Wir lernten uns bereits während des Grundstudiums kennen und schätzten den gleichen Arbeitsstil an uns. Mehrere Prüfungsvorbereitungen machten das deutlich. Da sich ferner unsere Interessengebiete (s.o.) deckten, lag es nicht fern, während des Hauptstudiums zusammenzuarbeiten.

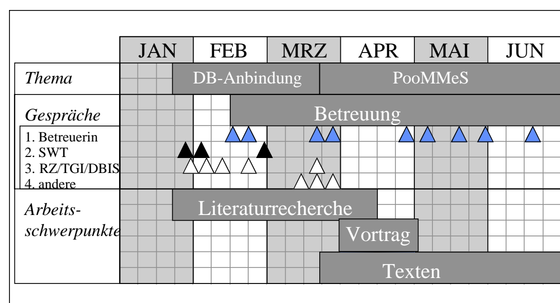


Abbildung A.1: Der Projektverlauf

Unser Arbeitsstil ist geprägt durch Schnelligkeit, ständiges Hinterfragen von (vermittelten) Sachverhalten, Auffassungsgabe, aber auch durch Spaß an der Arbeit. Wir waren uns schnell einig, daß wir die Studien- und die Diplomarbeit als eine Einheit betrachten und auf einem noch unerforschten Gebiet ein innovatives Produkt erstellen wollen. Durch gegenseitiges Aufbauen und „Pushen" konnten wir unser erstes Teilziel in Form dieser Studienarbeit zügig erreichen. Dabei fiel uns auf, daß sich von Zeit zu Zeit höchstens einer von uns in einem Stimmungstief befand.

Unsere Betreuerin, die wir zu Beginn unserer Arbeit nicht kannten, konnten wir schnell für unsere Ideen gewinnen, zumal das Thema der Arbeit im Bereich eines ihrer Forschungsschwerpunkte liegt. Obwohl wir anfänglich nur wenige Gespräche mit ihr hatten, konnten wir Frau Dr. Wetzel unsere Überlegungen durch Seminarvorträge, die zur Grundlage dieser Arbeit wurden, vermitteln.

Literaturverzeichnis

[Agr+93] R. Agrawal, S. Dar, N. Gehani: *The O++ Database Programming Language: Implemention and Experience*, in: Proceedings Ninth International Conference on Data Engineering, IEEE Computer Society, Wien, 1993.

[Bil+93] A. Biliris, S. Dar, N. H. Gehani: *Making C++ Objects Persistent: the Hidden Pointers*, in: Software-Practice and Experience, Vol.23, No.12, Seite 1285-1303 , 12/1993.

[Boo91] G. Booch: *Object Oriented Design with Applications*, Benjamin/Cummings Publishing Company Inc., 1991.

[Cat+94] R.G.G: Cattell et al.: *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, Inc., San Francisco, 1994.

[Com91] Communications of the ACM: *Next-Generation Database Systems*, Special Issue, Communications of the ACM, Vol.34, No.10, 10/1991.

[Dab+95] D. Dabbene, S. Damiani: *Adding persistence to objects using smart pointers*, Alenia Aeronautica, Artificial Intelligence Laboratory and Systems Engineering Dept., in: Journal of object-oriented programming, Vol.8, Nummer 3, Torino 6/1995.

[Dix88] G. N. Dixon: *Object Management for Persistence and Recoverbility*, Technical Report Series, No. 276, University of Newcastle upon Tyne, 12/1988.

[Dix+89] G. N. Dixon, G.D. Parrington, S.K. Shrivastava, S.M. Wheater: *The Treatment of Persistent Objects in Arjuna*, Technical Report Series, No. 283, University of Newcastle upon Tyne, 6/1989.

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

[Gil93] W. K. Giloi: *Rechnerarchitektur*, 2. überarbeitete Auflage, Springer Verlag, Berlin u.a., 1993.

[Gry95] G. Gryczan: *Situierte Koordination computergestützter qualifizierter Tätigkeit über Prozeßmuster*, Dissertation, Fachbereich Informatik, Universität Hamburg, 1995.

[Hei92] Klaus von der Heide: *Grundzüge der Informatik B*, Skriptum zur gleichnamigen Vorlesung, Fachbereich Informatik, Universität Hamburg, 1992.

[Kak+92] Vivek Singhal, Sheetal V. Kakkad, Paul R. Wilson: *Texas: An Efficient, Portable Persistent Store*, in: Persistent Object Systems, Antonio Albano et al., Springer, 9/1992.

[Kem+92] A. Kemper, G. Moerkotte: *Grundlagen objektorientierter Datenbanksysteme*, Bericht Nr. 92/9, Aachener Informatik-Berichte, Fachgruppe Informatik, Rheinisch-Westphälische Technische Hochschule Aachen, 1992.

[Kil+93] K. Kilberth, G. Gryczan, H. Züllighoven: *Objektorientierte Anwendungsentwicklung*, Konzepte, Strategien, Erfahrung, Vieweg, Braunschweig, Wiesbaden, 1993.

[Kir95] H. Kirschke: *Persistenz in der objekt-orientierten Programmiersprache CLOS am Beispiel des PLOB! Systems*, Bericht Nr. 179, Fachbereich Informatik, Universität Hamburg, 1995.

[Loo93] M. E. S. Loomis: *Object programming + database management*, Versant Object Technology, in: Journal of object-oriented programming, Vol.5, Nummer 9, Colorado Springs 1993.

[Mit86] B. Mitschang: *MAD. Ein Datenmodell zur Verwaltung von komplexen Objekten*, (Sonderforschungsbereich 124 Vlsi-Entwurfsmethoden und Parallelität: reports; 85/20), 1986.

[Ric+89] J. E. Richardson, M. J. Carey: *Persistence in the E Language: Issues and Implementation*, in: Software-Practice and Experience, Vol. 19(12), Seite 1115-1150, 12/1989.

[Ric+93] J. E. Richardson, M. J. Carey: *The Design of the E Programming Language*, in: ACM Transactions on Programming Languages and Systems, Vol.15, No.3, Seite 494-534, 7/1993.

[Sch+94] J. Schmidt, R. Müller: *Objektspeichersysteme*, Skriptum zur gleichnamigen Vorlesung, Fachbereich Informatik, Universität Hamburg, 1994.

[Shr88] S. K. Shrivastava, G. N. Dixon et al.: *A technical overview of Arjuna: a system for reliable distributed computing*, Technical Report Series, No. 262, University of Newcastle upon Tyne, 7/1988.

[Sin+92] Vivek Singhal, Sheetal V. Kakkad, Paul R. Wilson: *Texas: Good, Fast, Cheap Persistence in C++*, in: ACM SIGPLAN OOPS, Vol.4,No.2, Seite 145-147, 1993. upon Tyne, 7/1988.

[Suz+94] Shinji Suzuki, Masaru Kitsuregawa, Mikio Takagi: *An Efficient Pointer Swizzling Method for Navigation Intensive Applications*, in: Persistent Object Systems, Malcolm Atkinson et al., Springer, 1995.

[Vau+92] Francis Vaughan, Alan Dearle: *Supporting Large Persistent Stores using Conventional Harware*, in: Persistent Object Systems, Antonio Albano et al., Springer, 9/1992.

[Wil+92] Sheetal V. Kakkad, Paul R. Wilson: *Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware*, in: IEEE Press., Workshop on Object Orientation in Operating Systems, Seite 364-377, 1992. upon Tyne, 7/1988.

[Wul+94] M. Wulf und D. Weske: *Konzepte zur Materialversorgung verteilter Werkzeugumgebungen am Beispiel einer objektorientierten Datenbank*, Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1994.