OMS-orientierte Werkzeugarchitekturen

Udo Kelter

15.11.1999

Zusammenfassung dieses Lehrmoduls

Durch Einsatz eines Objektmanagementsystems (OMS) soll im Prinzip der Aufwand für die Realisierung von Werkzeugen reduziert werden und/oder die Qualität der Werkzeuge verbessert werden. In diesem Lehrmodul untersuchen wir, welche Dienste des OMS in diesem Sinne ausnutzbar sind und wie die Werkzeuge beschaffen sein müssen, damit sie diese Dienste überhaupt effektiv ausnutzen können. Hierzu diskutieren wir OMS-orientierte Werkzeugarchitekturen. Weiter skizzieren wir Anforderungen, die an einige Dienste von OMS zu stellen sind.

Vorausgesetzte Lehrmodule:

obligatorisch:
- Vorgehensmodelle
- Software-Entwicklungsumgebungen
- Integrationsrahmen für Software-Entwicklungsumgebungen

Stoffumfang in Vorlesungsdoppelstunden: 1.3

Inhaltsverzeichnis

1 Einleitung

In diesem Lehrmodul gehen wir von der Annahme aus, daß die Daten einer SEU in einem Objektmanagementsystem (wie z.B. H-PCTE) verwaltet werden sollen, und untersuchen die Frage, welche Konsequenzen dies für die Architektur von Werkzeugen, insb. Editoren, in einer SEU hat, oder anders gesagt, wie sich ein OMS in die Gesamtarchitektur einer SEU einordnet.

Ein zentrales Ziel eines OMS sollte es sein,

1. die Realisierung von SEU bzw. Werkzeugen zu vereinfachen und insbesondere den Aufwand zur Programmierung und Wartung von Werkzeugen zu reduzieren[1], indem komplexe Datenverwaltungsfunktionen nur einmal im OMS realisiert werden und nicht in jedem Werkzeug erneut[2]. Eine Aufwandsreduktion ist möglich, indem Teilprobleme bei der Werkzeugrealisierung durch Dienstleistungen des OMS gelöst werden. Der Nutzeffekt eines OMS sollte deutlich sichtbar sein, möglichst in gleicher Weise wie bei konventionellen DBMS und konventionellen Anwendungen, wo viele Standardprobleme durch eine einfache und kurze Anfrage gelöst werden können, während ohne ein DBMS ein längliches und fehlerträchtiges Programm geschrieben werden müßte.
2. bessere Werkzeuge zu ermöglichen, d.h. es zu erlauben, Funktionen und Eigenschaften von Werkzeugen zu realisieren, deren Realisierung ohne ein OMS zu aufwendig wäre.
3. einen hohen Grad der Integration der Werkzeuge einer SEU zu ermöglichen.

Mit anderen Worten sollte ein Entwickler einer SEU einen meßbaren Nutzen erkennen, wenn er ein OMS verwendet (andernfalls verwendet

[1] im Vergleich zur Benutzung von Dateisystemen.

[2] Dies gilt ganz allgemein für *application frameworks*; bei der Realisierung von SEU wird man speziell für graphische Werkzeuge neben einen OMS auch ein UIMS und ggf. weitere Basissysteme einsetzens. auch [IRA].

er nämlich keins). Letztlich sind die vorgenannten Punkte softwaretechnische Ziele für eine spezielle Klasse von Software, nämlich gerade SEU.

Aus den vorstehenden Überlegungen folgt, daß die von einem OMS angebotenen Funktionen möglichst direkt die Probleme bei der Verwaltung von Entwicklungsdaten einzelner Werkzeuge lösen sollten – dies kann man als eine Anforderung an ein OMS ansehen[3]. Umgekehrt ergeben sich aber auch Anforderungen an die Architektur von SEU! Um es an einem krassen Beispiel zu zeigen: wenn man die langen Felder, die OMS üblicherweise anbieten[4], dazu mißbraucht, ein Dateisystem zu simulieren, und eine SEU in altgewohnter Weise auf diesem Ersatz-Dateisystem realisiert, ist natürlich kein Vorteil durch das OMS gegenüber einem Dateisystem zu erwarten. Man muß also solche SEU-Architekturen anstreben, die es erlauben, die Leistungen eines OMS tatsächlich auszunutzen; solche SEU-Architekturen nennen wir "**OMS-orientiert**".

2 Potentiell ausnutzbare OMS-Dienste

Beispiele für Problemkomplexe, die bei der Konstruktion von Werkzeugen auftreten und die potentiell durch Dienste und technische Merkmale von OMS lösbar erscheinen, sind:

- die Datenintegration verschiedener Werkzeuge mit Hilfe externer Sichten
- die Überwachung der Konsistenz von Dokumenten mit Hilfe von Schemamechanismen oder Triggern

[3] Tatsächlich fällt es sehr schwer, diese Anforderung zu konkretisieren, ohne sich auf eine bestimmte Werkzeugarchitektur und die spezielle Art, wie diese Werkzeuge auf den Daten operieren, festzulegen.

Unabhängig davon erfüllen real existierende OMS nur selten alle Anforderungen, die man aus der Analyse des Bedarfs unterschiedlicher Werkzeuge ableiten kann. Bei den PCTE-Standards fehlen z.B. mengenorientierte Abfragemöglichkeiten.

[4] In H-PCTE sind z.B. alle String-Attribute an Objekten lange Felder: ihre Länge ist praktisch nicht begrenzt, und sie können zeichenweise gelesen und geschrieben werden.

- Zugriffskontrollen, die möglichst Gruppenstrukturen und rollenorientierte Rechte unterstützen sollten
- Synchronisation von parallelen Zugriffen mehrerer Werkzeugprozesse auf das gleiche Dokument, wobei verschiedene Grade denkbar sind, in denen Werkzeugprozesse voneinander isoliert werden.
- Notifizierung der Werkzeugprozesse über Änderungen an den Daten, die sie zuvor gelesen haben.

Speziell bei der Konstruktion von (graphischen) Editoren und Anzeigewerkzeugen erscheinen noch folgende Dienste des OMS potentiell ausnutzbar:

- die selektive Anzeige von Dokumenten mit Hilfe externer Sichten
- Generierung bestimmter Menüs / Kommandos der Editoren aus dem Datenbankschema
- die Realisierung von Undo-Kommandos in Editoren mit Hilfe von eines partiellen Rollbacks von Transaktionen
- die Propagation von Änderungen zwischen Fenstern mit Hilfe eines Benachrichtigungsmechanismus
- Suche nach relevanten Dokumenten oder Dokumentteilen mit Hilfe von Abfragesprachen

Der Umfang der Leistungen, die bei der Konstruktion von Werkzeugen ausnutzbar erscheinen, ist auf den ersten Blick groß; tatsächlich sind jedoch vielerlei Randbedingungen einzuhalten, um eine tatsächliche Aufwandsreduktion bei der Konstruktion von Werkzeugen zu erzielen.

Eine erste Randbedingung betrifft die Verwaltung von Dokumenten im Hauptspeicher von Werkzeugen. Wir wollen im folgenden zunächst zeigen, daß die Leistungen eines OMS mit *konventionellen Werkzeugarchitekturen nicht effektiv ausgenutzt werden können.*

3 Konventionelle Werkzeugarchitekturen

3.1 Merkmale

Konventionelle Werkzeuge speichern Dokumente in Dateien, z.B. je ein OOA-Diagramm in einer Datei. Wir sprechen hier von einer *grobgranularen Datenmodellierung*. Der Dateiinhalt hat eine bestimmte Syntax, mit deren Hilfe die Feinstruktur des Dokuments rekonstruiert werden kann. Allerdings können die Funktionen eines Werkzeugs (z.B. das Erzeugen oder Löschen einer Klasse beim Editieren eines OOA-Diagramms) nicht direkt auf dem Dateiinhalt durchgeführt werden; stattdessen muß zunächst der Dateiinhalt in eine temporäre Kopie im Hauptspeicher des Werkzeugprozesses konvertiert werden, bei Quellprogrammen beispielsweise in einen Syntaxbaum (s. Bild 1). Die temporäre Kopie eines Dokuments wird beim "Sichern" oder "Schließen" des Dokuments wieder zurückkonvertiert.

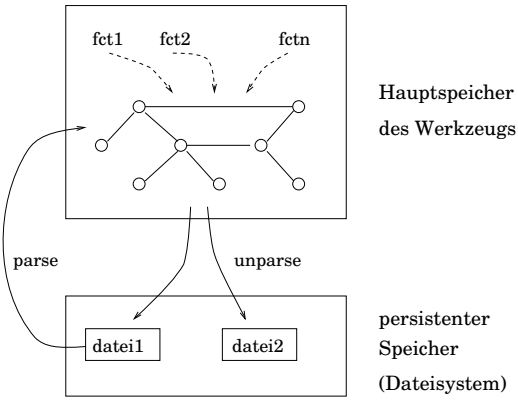

Abbildung 1: Arbeiten auf Dateien

3.2 Probleme konventioneller Werkzeugarchitekturen

Die grobgranulare Datenmodellierung und das mit einhergehende Arbeiten auf temporären Kopien weist eine ganze Reihe von Nachteilen

auf, die hier aus Platzgründen nur teilweise aufgezählt werden sollen.

1. Zunächst ist die Datenunabhängigkeit der Werkzeuge sehr gering. Sofern z.B. ein neues Werkzeug in eine Umgebung integriert werden soll und dieses Werkzeug spezielle zusätzliche Daten innerhalb von bereits vorhandenen Dokumenttypen benutzt, muß die Syntax der Dateien entsprechend erweitert werden. Daher müssen die Parser und Unparser in allen betroffenen Werkzeugen verändert werden, was überhaupt nicht möglich ist, wenn die Werkzeuge nicht in Quellform verfügbar sind. Weiterhin entstehen erhebliche Probleme beim Transport der zusätzlichen Daten durch vorhandene Werkzeuge hindurch.
2. Erfahrungsgemäß ist auch die Datenintegration mehrerer Werkzeuge nicht optimal oder nur unter ganz erheblichem Aufwand zu erreichen, weil hierzu viele Konvertierer, die äquivalente Daten zwischen unterschiedlichen Formaten umformen, konstruiert werden müssen, was aber nicht immer völlig verlustfrei möglich ist.
3. Ein weiterer, in der Praxis ganz entscheidender Nachteil ist, daß diese Architektur keine inkrementelle Überprüfung dokumentübergreifender Konsistenzkriterien erlaubt. Nehmen wir als Beispiel ein Programmsystem an, dessen Modulstruktur in einem Modulstrukturdiagramm beschrieben wird, das in einer Datei gespeichert ist. Für jedes Modul möge das Quellprogramm in einer weiteren Datei gespeichert sein. Wenn nun mit einem Editor der Typ eines Parameters in dem Modulstrukturdiagramm geändert wird, muß in den zugehörigen Quellprogrammen diese Änderung nachvollzogen bzw. bei allen Aufrufen der betroffenen Prozedur geprüft werden, ob der aktuelle Parameterwert kompatibel mit dem neuen Typ des formalen Parameters ist. Für derartige Prüfungen nehmen wir ein anderes Werkzeug an ("Prüfer").

   Bei grobkörniger Datenmodellierung muß nun der Quelltext des Moduls wieder linearisiert und in die Datei zurückgeschrieben werden; anschließend liest und analysiert das Prüfprogramm den Inhalt dieser Datei und ggf. den Inhalt Dutzender anderer Dateien, da typischerweise *alle* Konsistenzkriterien überprüft werden müssen,

nicht nur diejenigen, die mit dem einen veränderten Parametertyp zusammenhängen.

4. Weil so konstruierte Umgebungen *nicht inkrementell arbeiten können*, arbeiten sie oft extrem ineffizient. In der Praxis werden dokumentübergreifende Prüfungen deshalb möglichst aufgeschoben oder ganz unterlassen und Fehler u.U. überhaupt nicht oder erst in einem späten Stadium entdeckt.

Um jetzt Dateien als Speichermedium nicht zu sehr zu verteufeln, sei gleich zugegeben, daß vielfach gar keine Alternative verfügbar ist und daß auch OMS Nachteile haben, die wir später besprechen werden und die die Einsatzmöglichkeiten von OMS einschränken.

3.3 Management temporärer Objekte

Bei konventionellen Werkzeugen müssen die temporären Kopien der Dokumente im Hauptspeicher verwaltet werden; hierzu beinhaltet die Architektur konventioneller Werkzeuge naheliegenderweise für jeden Dokumenttyp, mit dem das Werkzeug arbeitet, ein Modul, das die temporären Kopien von Dokumenten im Hauptspeicher verwaltet, s. Modul MTO (Management temporärer Objekte) in Bild 2. Es handelt sich hier um ein klassisches Datentyp-Modul, das alle elementaren Operationen anbietet, mit denen man Exemplare dieses Dokumenttyps anlegen, verändern, Einzeldaten auslesen und ganz aus einem Dateinhalt aufbauen oder dorthin zurückschreiben kann.

Ersetzt man nun in dieser Architektur das Dateisystem durch ein OMS, so ergibt sich der neueste Stand der Dokumente nur noch aus den temporären Kopien! An dieser Stelle ist es wichtig, sich daran zu erinnern, daß bei der Softwareentwicklung fast ständig irgendwelche Entwicklungsdokumente verändert werden. Das OMS enthält deshalb sehr bald veraltete Daten und *sämtliche Leistungen des OMS, wie z.B. Abfragesprachen, werden praktisch wertlos*[5].

[5] Die gleichen Nachteile treten ein, wenn man zwar ein OMS verwendet, aber aus diesem nur per *check-out* und *check-in* Daten zum bzw. von Werkzeug überträgt.

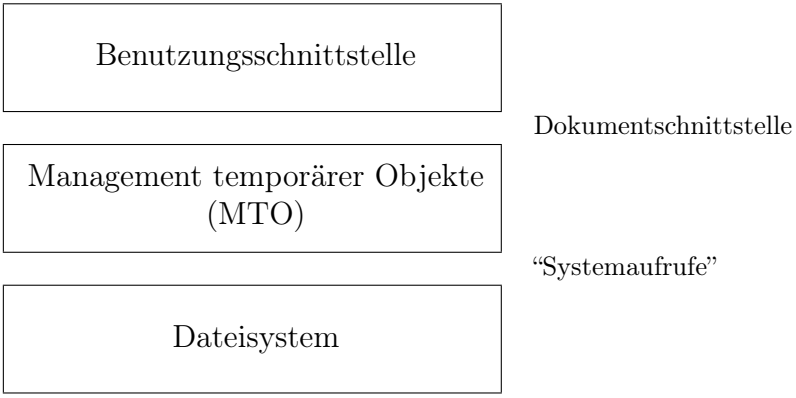

Abbildung 2: Konventionelle Werkzeugarchitektur

4 OMS-orientierte Werkzeugarchitekturen

4.1 Direkte Propagation von Änderungen

Will man die Leistungen eines OMS ausnutzen, muß man dafür sorgen, daß die Objektbank die richtigen Daten enthält. Aus dieser und den vorstehenden Beobachtungen ergeben sich zwei wesentliche Merkmale OMS-orientierter SEU-Architekturen:

1. Die Werkzeuge einer SEU müssen alle relevanten Änderungen an den Dokumenten, namentlich solche,die ein Benutzer durch eine Eingabe veranlaßt hat, *inkrementell* und *sofort* in das unterliegende OMS weitergeben.

   Dies schließt nicht aus, daß in den Werkzeugen trotzdem noch Kopien der Daten vorhanden sind, dies kann aus Performance-Gründen unverzichtbar sein.
2. Die Entwicklungsdaten müssen *feingranular* modelliert werden, d.h. die Feinstruktur der Dokumente muß vollständig im OMS nachgebildet werden. Hierfür gibt es mehrere Gründe:

   - Viele Dienste eines OMS (Abfragen, Sperren, Rollback u.a.) sind

nur denkbar, wenn das OMS die Feinstruktur der Daten kennt.

- Theoretisch kann man auch bei einer grobgranularen Datenmodellierung bei jeder relevanten Änderung an einem Dokument dieses komplett zurückschreiben; dies würde aber zu erheblichen Performance-Problemen führen (vgl. unten die Diskussion der Performance-Probleme bei redundanzfreien Architekturen).

Ein weiterer Nachteil der in Bild 2 gezeigten Architektur ist, daß der Aufwand zur Implementierung der Werkzeuge durch den Einsatz eines OMS *praktisch nicht reduziert* wird: die Parser und Unparser, die zwischen Dateiinhalten und Hauptspeicherdatenstrukturen konvertieren, werden ersetzt durch andere Software, die zwischen Datenbankinhalt und Hauptspeicherdatenstrukturen konvertiert. Diese Software ist aus Sicht von Werkzeugentwicklern möglicherweise sogar schwerer (!) zu realisieren als ein Parser und Unparser für eine einfache Syntax[6], weil hierzu Datenbankschemata entwickelt und ein kompliziertes API des OMS erlernt werden muß.

Wegen der komplexen Struktur der meisten Dokumenttypen werden die MTO-Module relativ umfangreich, typischerweise in der Größenordnung von 3000 bis 7000 Zeilen Quelltext. Da in einer Umgebung meist mehrere Dokumenttypen auftreten (typischerweise 8 - 12), stellen die MTO-Module ein erhebliches Implementierungs- und Wartungsproblem dar.

I.f. skizzieren wir mehrere denkbare Lösungsansätze.

4.2 Redundanzfreie Architekturen

Eine radikale Lösung besteht darin, die MTO-Module im Prinzip schlichtweg zu vermeiden und *direkt auf den Daten in der Objektbank zu arbeiten* (s. Bild 3). Diese Architektur ist in doppelter Hinsicht redundanzfrei: die doppelte Datenhaltung in der Objektbank und im Hauptspeicher und die doppelte Realisierung von Dienstleistungen im OMS und durch die Applikation werden vermieden.

[6] Womit nicht gesagt sein soll, es sei trivial, einen Compiler zu schreiben...

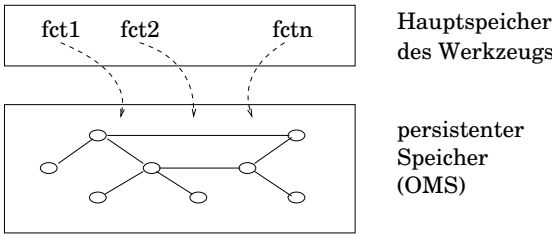

Abbildung 3: Direktes Arbeiten auf der Objektbank

Da hier noch nicht einmal Lesepuffer vorhanden sind, ist diese Lösung aus Performance-Gründen allerdings nur selten anwendbar, i.w. wenn nur wenige Objekte angezeigt werden[7].

4.3 Generatoren

Ein weiterer naheliegende Ansatz besteht darin, die MTO-Module zu generieren. Als Ausgangangsbasis kommen die Datenbankschemata infrage. Alternativ können die Datenbankschemata und die MTO-Module aus einer gemeinsamen Quelle generiert werden.

Nachteilig an der Generierung ist, daß die erforderlichen Generatoren (genaugenommen sind dies Übersetzer, die Spezifikationen in Quellcode transformieren) einen signifikanten Implementierungsaufwand verur-

[7]Bei graphischen Editoren und Anzeigewerkzeugen erweist sich hier das Neuanzeigen von Bildschirmen, bei dem jeweils alle Objekte eines Dokumente gelesen werden müssen, als "Killeroperation": Wenn ein Benutzer das Fenstersystem so eingestellt hat, daß das Fenster, über dem sich der Mauszeiger befindet, nach oben geholt wird, wird diese Operation sehr häufig ausgelöst.

Probleme sind auch bei Verschiebeoperationen in Graphiken zu erwarten. Betrachten wir als Beispiel ein netzartiges Diagramm (z.B. ein ER-Diagramm) und den Fall, daß man die Koordinaten eines Knotens oder die Stützpunkte einer Verbindungslinie in einem Attribut speichert und einen Knoten oder Stützpunkt mit der Maus verschiebt: dies führt zu einer hohen Rate von Ereignissen, bei denen die neuen Koordinaten in der Objektbank gespeichert werden müssen. Hinzu kommen weitere Leseoperationen, wenn aufgedeckte Fensterteile neu gezeichnet werden müssen. In solche Fällen ist allerdings denkbar, nur die Stützpunkte in Puffern zu speichern, während die restlichen Dokumentdaten nicht gepuffert werden.

sachen und daß sie wenig flexibel sind, denn für jede Änderung im generierten Code sind auch die Generatoren zu ändern.

Problematisch sind insb. Ausnahmefälle und semantische Besonderheiten einzelner Dokumenttypen, die entweder den Generator aufblähen oder, falls man hierfür nachträglich den generierten Code ändert, zu praktischen Problemen führen.

4.4 Interpreterarchitekturen

Anstatt Code der MTO-Module mit Hilfe von Generatoren aus Spezifikationen zu generieren, kann man prinzipiell auch die Spezifikationen dynamisch interpretieren. Der Interpreter erscheint dann als ein generisches MTO-Modul, das durch die Datenbankschemata und ggf. zusätzliche Steuerparameter gesteuert wird. Die Steuerparameter könnten z.B.

- in einer Resource-Datei
- als Programm-Konstanten
- in Objekten der Objektbank (!)

verwaltet werden.

Auf Interpreterarchitekturen trifft der Nachteil der mangelnden Flexibilität genauso zu wie auf Generatoren. Realisiert man den Interpreter allerdings in einer objektorientierten Sprache, können Ausnahmefälle noch vergleichsweise einfach durch geeignetes Überschreiben von Operationen integriert werden.

4.5 Konstruktion graphischer Bedienschnittstellen

Die vorstehenden Überlegungen abstrahierten vom konkreten Werkzeugtyp. Bei der wichtigen Klasse der (heute meist graphischen) Editoren und Anzeigewerkzeuge ergeben sich einige Besonderheiten.

Setzt man zur Konstruktion der graphischen Bedienschnittstellen ein UIMS ein, so muß man für dieses doch wiederum temporäre Kopien erzeugen; UIMS arbeiten nämlich normalerweise auf Basis des

Model-View-Controller-Paradigmas, und das Model darin ist eine Darstellung des angezeigten Dokuments im Hauptspeicher. Eine völlig redundanzfreie Architektur ist hier nicht ohne massive Eingriffe in das UIMS möglich und praktisch unrealistisch. Am ehesten kann man die oben vorgestellen Interpreterarchitekturen mit UIMS kombinieren, da ähnliche Konzepte auch in den UIMS eingesetzt werden.

Weiter zielten unsere bisherigen Überlegungen nur auf die Aufwandsreduktion bei der Konstruktion der MTO-Module; die Konstruktion der Bedienschnittstellen vieler Editoren und Anzeigewerkzeuge stellt aber ein ganz ähnliches Aufwandsproblem dar. Für jeden Editor muß nämlich die Bedienschnittstelle neu realisiert werden, da normalerweise dokumenttypspezifische Besonderheiten auftreten. Beispiele für solche Besonderheiten in graphischen Editoren für netzartige Diagramme sind:

- Typischerweise gibt es Menüs, durch die man Elemente der Graphen (z.B. Klassen in OOA-Diagrammen, Entitätstypen in ER-Diagrammen oder Zustände in Zustandsübergangsdiagrammen) erzeugen kann. Die erlaubten Knotentypen und ihre Darstellungen sind dokumenttypspezifisch.
- Analog gibt es Menüeinträge, durch die Verbindungen zwischen den Elementen erzeugt werden können, teilweise mit speziellen Darstellungsformen und Beschriftungen (z.B. eine Verbindung zwischen einem Entitätstyp und einem Beziehungstyp in einem ER-Diagramm, ggf. mit Kardinalitäten).
- Beim Erzeugen von Verbindungen müssen vielfach spezielle Konsistenztests durchgeführt werden (z.B. können in einem ER-Diagramm nicht zwei Beziehungstypen mit einer Kante verbunden werden, Namen innerhalb eines Diagramms müssen oft eindeutig sein usw.). Sowohl die Tests als auch die Fehlermeldungen oder sonstigen Interaktionen bei Auftreten eines Fehlers sind dokumenttypspezifisch.
- Es kann dokumenttypspezifische Kommandos geben. Beispielsweise ist es für das Bearbeiten von Klassendiagrammen sehr praktisch, wenn man mit einer einzigen Interaktion ein Attribut von einer Klasse zu einer anderen verschieben kann (z.B. indem man das Attribut

anklickt, "festhält", über die andere Klasse verschiebt und dort "fallen läßt") oder aus einem Attribut eine neue Komponentenklasse, die zunächst nur dieses eine Attribut enthält, machen kann.

Es liegt natürlich nahe, einen generischen Kern für alle Editoren zu bilden und die dokumenttypspezifischen Anteile in Steuerparameter einzukapseln, die der generische Kern interpretiert.

Wegen der Komplexität der dokumenttypspezifischen Anteile ist aber der Realisierungsaufwand bei diesen Vorgehensweisen erheblich. Besonders unschön ist, daß inhaltliche Redundanzen zwischen diesen Steuerstrukturen und dem Objektbankschema auftreten: die Menüeinträge entsprechen exakt bestimmten Objekt- oder Beziehungstypen in der Objektbank. Es liegt nahe, diese Redundanz zu vermeiden, indem man das Datenbankschema selbst als Steuerungsparameter verwendet.

5 Anforderungen an ein OMS in OMS-orientierten Architekturen

5.1 Performance

Da die architektonischen Vorteile des direkten Arbeitens auf der Objektbank und der feinkörnigen Datenmodellierung offensichtlich sind, stellt sich die Frage, warum derartige Architekturen bisher kaum realisiert werden. Ein erste entscheidendes Problem ist die Leistung des OMS. Die Leistungsanforderungen hängen stark von der Art des Werkzeugs ab. Syntaxeditoren oder Editoren für netzartige Diagramme benötigen eine Leistung von etwa 500 Operationen pro Sekunde, Editoren für sehr komplexe Graphiken oder Simulatoren benötigen eine Leistung von 10.000 oder sogar wesentlich mehr Operationen pro Sekunde[8] (typische Operationen sind das Lesen oder Schreiben von Attributwerten oder das Erzeugen oder Löschen von "elementaren" Objekten). Wenn

[8] Derartige Leistungen sind mit heutiger Datenbanktechnologie nicht erreichbar; deshalb betrachten wir diese Anwendungen nicht weiter im Kontext dieser Architektur.

man Leistungen in diesen Größenordnungen erzielen möchte, treten vor allem zwei Implementierungsprobleme auf:

1. Ein Plattenzugriff dauert selbst bei den heutigen sehr schnellen Platten größenordnungsmäßig etwa 10 ms. Wenn die Zeit für Plattenzugriffe nur einen Bruchteil der gesamten Operationsausführungszeiten ausmachen soll, dann darf nur ein Plattenzugriff bei mehreren hundert Operationen auftreten. Praktisch dürfen also überhaupt keine Plattenzugriffe auftreten, *das OMS muß als Hauptspeicher-DBMS realisiert werden.*
2. Auf heute erhältlichen Standardworkstations dauert der Austausch einer Nachricht zwischen zwei Prozessen etwa 0.2 - 1.0 ms. Werden OMS-Kern und Applikation in eigenen Prozessen ausgeführt - wie bei konventionellen DBMS üblich -, so fällt allein wegen der Interprozeßkommunikation die Leistung i.d.R. unter 1000 Operationen pro Sekunde. Hieraus folgt, daß derzeit, wenn man sehr hohe Leistungen erreichen will, *OMS-Kern (incl. DB-Puffer) und Applikationen im gleichen Prozeß* und insb. im gleichen Arbeitsspeicher ausgeführt werden müssen. Die resultierende Architektur nennen wir **Ein-Prozeß-Architektur**.

   Ihr gravierendster Nachteil ist ihre Unsicherheit: bei Sprachen wie C oder C++, die einen beliebigen Umgang mit Zeigern erlauben, muß man damit rechnen, daß durch Programmierfehler (oder sogar absichtlich) von der Applikation auf die Puffer oder den OMS-Kern zugriffen wird und dort Daten oder Programme verändert werden. Wirksame Schutzmaßnahmen, die nicht wieder zu Performance-Verlusten führen, sind schwer zu finden [Se93].

5.2 Werkzeuge als Bindemodule

Die Ein-Prozeß-Architektur hat deutliche Auswirkungen auf den Begriff eines Werkzeugs: Während dieser Begriff klassischerweise so interpretiert wird, daß ein Werkzeug als *ausführbares Programm* vorliegt, muß es bei der Ein-Prozeß-Architektur als *Bindemodul* vorliegen. Die vollständige Umgebung muß dann durch Binden dieser Bindemodule und des OMS-Kerns generiert werden. Dies stellt aber kein ernsthaftes

Hindernis dar; moderne Betriebssysteme unterstützen außerdem das dynamische Binden, bei dem einzelne Module erst bei ihrer ersten Benutzung zum laufenden Programm hinzugebunden werden. In einer ersten Näherung kann man sich den Hauptspeicher des SEU-Prozesses wie in Bild 4 dargestellt vorstellen. Die Module W_1 ... W_n sollen einzelne "Werkzeuge" darstellen.

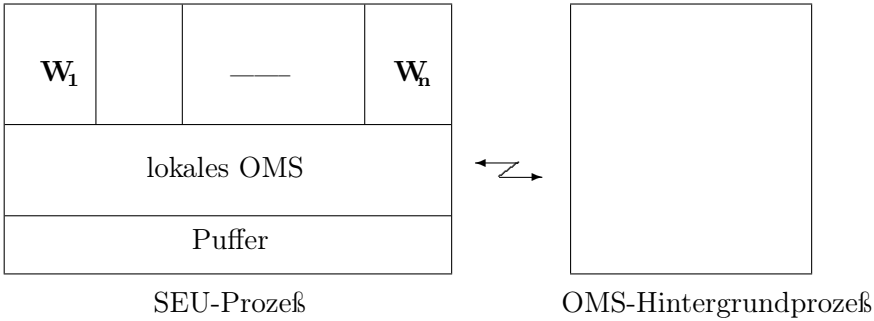

Abbildung 4: Hauptspeicher eines SEU-Prozesses

Werkzeuge vs. SEU-Komponenten. Der Begriff Werkzeug bzw. Komponente einer SEU entfernt sich hier etwas von dem, was man normalerweise als Benutzer unter einem "Werkzeug"versteht; er entspricht vielmehr dem Konzept eines unabhängig realisierbaren, wiederverwendbaren Moduls. Der Unterschied zwischen einem Werkzeug aus Benutzersicht und einer SEU-Komponente sei am Beispiel eines Werkzeugs zur Verwaltung von Modulhierarchien erläutert. Ein solches Werkzeug können wir uns aus folgenden SEU-Komponenten zusammengesetzt denken:

- Eine erste SEU-Komponente zeigt eine graphische Darstellung der gesamten Modulhierarchie in einem Fenster an. Hier sind die einzelnen Module nur sehr vergröbert als Rechtecke dargestellt und es können z.B. neue Module erzeugt, Modulnamen verändert und Import-/Export-Verbindungen eingetragen werden.

- Eine zweite SEU-Komponente zeigt die Details einer einzelnen Modulbeschreibung formularartig an und bietet übliche Editierkommandos an. Diese Komponente kann mehrfach parallel ausgeführt werden, um mehrere Modulbeschreibungen parallel ansehen und editieren zu können.
- Weitere SEU-Komponenten bieten Funktionen wie Konsistenzprüfungen, Drucken, Konversion in andere Darstellung usw. an. Diese Funktionen laufen originär nicht-interaktiv ab und enden schließlich mit einem Fehlercode. Aufgerufen werden sie aus den Editoren heraus (z.B. über Menüeinträge) oder über einen Browser.

Diese SEU-Komponenten können ebensogut als Teile anderer Werkzeuge erscheinen. Die Editoren für einzelne Modulbeschreibungen könnten z.B. auch innerhalb eines "Werkzeugs" zum Programmieren im Kleinen erscheinen, wenn z.B. die Parameterliste einer exportierten Funktion geändert werden muß. Konsistenzprüfungen könnten auch von einem Vorgehensmodelltreiber aufgerufen werden.

Leichtgewichtige Prozesse. Das vorstehende Beispiel des Modulbeschreibungseditors zeigt nebenbei, daß man strikt trennen muß zwischen einer SEU-Komponente im Sinne eines Stücks Software (also dem Bindemodul) und Ausführungen dieser Software: die gleiche Komponente kann nämlich mehrfach parallel ausgeführt werden. Bei konventionellen Werkzeugen geht dies einher mit einem *Kopieren* der Programme (in die virtuellen Adreßräume der ausführenden Prozesse). Bei unserer Ein-Prozeß-Architektur ist dies natürlich nicht möglich (ein Bindemodul kann nur einmal angebunden werden). Stattdessen benötigt man *leichtgewichtige Prozesse* (*threads*).

Für das OMS bedeutet dies, daß es möglich sein muß, in einem einzigen ("schwergewichtigen") Betriebssystemprozeß mehrere OMS-Prozesse zustarten, die unterschiedliche Schemata, Zugriffsrechte usw. haben können.

Verkompliziert wird die Situation bei graphischen interaktiven Werkzeugen dadurch, daß das UIMS oder Fenstersystem selbst in Form mehrerer Prozesse abläuft und u.U. davon ausgeht, selbst nach Belieben

Prozesse oder threads, in denen Benutzerinteraktionen bearbeitet werden, erzeugen und vernichten zu können. Die Details hängen stark von technischen Merkmalen des UIMS ab.

5.3 Problempunkte bei der Ausnutzung von OMS-Leistungen

Wir setzen im folgenden eine OMS-orientierte oder Interpreter-Architektur voraus und untersuchen, ob und wie unter dieser Annahme OMS-Leistungen effektiv ausnutzbar sind. Aus der in Abschnitt 2 aufgestellten Liste gehen wir vor allem auf Punkte ein, derentwegen die Werkzeugarchitektur weiter angepaßt werden muß oder aus denen sich weitere Anforderungen an ein OMS ergeben.

Aus Platzgründen können hier viele Themen nur angerissen werden; eine detaillierte Behandlung wird auf eigene Lehrmodule vertagt.

Konsistenzüberwachung durch Schemamechanismen: Softwaredokumente müssen in korrektem Zustand meist mehr oder weniger komplexe Konsistenzbedingungen einhalten. Viele DBMS erlauben es, entsprechende Integritätsbedingungen in Schemata zu definieren. Diese Bedingungen werden bei jeder Datenmanipulation oder beim Ende einer Transaktion überprüft und führen zur Zurückweisung der Operation bzw. zum Rollback der Transaktion. Diese Mechanismen sind nur sehr begrenzt oder gar nicht brauchbar:

- Ein Benutzer erwartet, wenn eine Inkonsistenz entdeckt wird, nicht nur eine Meldung, daß irgendwo irgendein Fehler vorhanden ist, sondern genaue Hinweise, welcher Fehler wo und in welchem Zusammenhang aufgetreten ist, damit er ihn sofort beheben kann.

  Schemamechanismen liefern normalerweise nur einen simplen Fehlercode, aus dem nur grob die Art der Konsistenzverletzung hervorgeht, nicht aber, welche Objekte involviert sind und welches Kriterium verletzt worden ist. Bei nichttrivialen Konsistenzkriterien bräuchte man eine Schnittstelle, über die im Fehlerfall auch eine komplexe Situationsbeschreibung geliefert werden kann. Derartige Schnittstellen fehlen normalerweise. Dies hat zur Folge, daß das

Werkzeug im Fehlerfall praktisch die gesamte Konsistenzprüfung wiederholen muß, um die betroffenen Objekte zu finden. M.a.W. muß der komplette Konsistenztest, den das OMS ausführt, im Werkzeug re-implementiert werden. Es wird also keinerlei Programmieraufwand durch das OMS eingespart. Im Gegenteil entsteht das Problem, daß der Konsistenztest im OMS und die nachprogrammierte Version möglicherweise nicht völlig einheitlich sind.

- In SEU müssen auch unfertige, inkonsistente Dokumente verwaltet werden. Inkonsistente Zwischenzustände treten auch bei klassischen Applikationen auf, aber nur sehr kurzfristig, so daß man dort das Problem umgehen kann, indem man die Konsistenztests erst am Ende einer Transaktion durchführt. Die Zeiträume, während der ein Dokument unfertig und inkonsistent ist, sind wesentlich länger[9] und liegen nicht komplett innerhalb eines Sitzung; daher kann auch nicht das klassische Transaktionskonzept zur Definition der Zeitpunkte, an denen Konsistenztests durchgeführt werden, heranhiehen.

Wegen dieser Probleme kann man durch Konsistenzbedingungen, die im Schema definiert werden und die sofort bei jeder Datenmanipulationsoperation oder am Ende einer Transaktion überprüft werden, nur sehr einfache Konsistenzbedingungen sinnvoll prüfen lassen, die (a) so wichtig sind, daß man auch eine kurzfristige Verletzung nicht zulassen möchte und die (b) sehr einfach in einer Fehlermeldung erläuterbar sind.

Weitergehende Kriterien sollten nur auf explizite Anforderung des Benutzers überprüft werden; hierzu eignen sich primär Abfragen, in denen nach den fehlerhaften Stellen eines Dokuments gesucht wird. Eine leistungsfähige Abfragesprache ist somit für die Konsistenzüberwachung viel wertvoller als Schemamechanismen.

Realisierung von Undo-Kommandos in Editoren mit Hilfe von eines partiellen Rollbacks von Transaktionen: Die ist im Prinzip ohne große Probleme realisierbar, sofern das OMS ein partielles

[9] Es soll sogar Dokumente geben, die nie fertig werden.

Rollback auf Sicherungspunkte unterstützt. Allerdings ist aus Benutzersicht ein Undo ohne die Möglichkeit eines Redo fast inakzeptabel. Ein partielles Redo innerhalb von Transaktionen wird von fast keinem OMS angeboten und stellt ein erhebliches Implementierungsproblem dar.

Datenintegration verschiedener Werkzeuge mit Hilfe externer Sichten (sog. *multiple view environments*): Dies bedeutet, daß unterschiedliche Dokumente inhaltlich überlappen können, daß der Gesamtzustand aller Dokumente in der Objektbank redundanzfrei modelliert wird und daß der Teil, der für ein bestimmtes Werkzeug relevant ist, durch eine Sicht herausgefiltert wird.

Probleme mit diesem trivial klingenden Ansatz entstehen, wenn mehrere Werkzeuge direkt parallel auf überlappenden Dokumenten arbeiten: ändert ein bestimmter Werkzeugprozeß ein Dokument, erhält er Schreibsperren, durch die andere Werkzeugprozesse das Dokument nicht mehr geschützt lesen können. Diese Probleme sind nur lösbar, wenn zum einen das OMS eine Kombination aus ungeschütztem Lesen und geschütztem Schreiben (mit Recovery) anbietet und wenn zum anderen das Schema der Objektbank sehr sorgfältig entworfen wird, um unnötige Sperrkonflikte zu vermeiden.

Propagation von Änderungen zwischen Fenstern mit Hilfe eines Benachrichtigungsmechanismus: die wesentliche Leistung solcher Benachrichtigungsmechanismen besteht darin, daß sich ein Prozeß, der Dokumente anzeigt, von Änderungen an diesem Dokument (durch andere parallele Prozesse) informieren lassen kann. Bei einem *verteilten* Benachrichtigungsmechanismus können die anderen Prozesse auch auf anderen Rechnern ablaufen.

Sog. aktive Datenbanken sind für die vorliegende Aufgabe prinzipiell nicht geeignet, weil als Reaktion auf Änderungen (oder andere überwachbare Ereignisse) jeweils nur eine Datenmanipulation oder eine gespeicherte Prozedur ausgeführt wird, durch die typischerweise Änderungen *innerhalb der Objektbank* propagiert werden, es ist nicht intendiert, einen oder mehrere externe Prozesse gezielt anzusprechen.

Zur Lösung der vorliegenden Aufgabe werden teilweise Mechanismen eingesetzt, bei denen Prozesse, die Dokumente ändern, selbst Änderungsnachrichten generieren müssen und bei denen die Empfänger der Nachrichten explizit Nachrichtenwarteschlangen verwalten müssen. Beides führt zu erheblichem Programmieraufwand und weiteren Problemen, die in [PlK97] ausführlicher diskutiert werden. Wesentlich weniger Probleme und Programmieraufwand verursachen Mechanismen, bei denen

- Nachrichten vom OMS generiert werden
- Nachrichten die vollständige Information über den neuen Zustand der geänderten Daten enthalten
- durch einen "upcall" direkt Operationen zur Korrektur des Bildschirminhalts vom OMS aus aufgerufen werden können.

Es muß dabei möglich sein, auch Nachrichten bzgl. Änderungen an Objekten infolge Rollback generieren zu lassen.

Literatur

[PlK97] Platz, D.; Kelter, U.: Konsistenzerhaltung von Fensterinhalten in Software-Entwicklungsumgebungen; Informatik – Forschung und Entwicklung 12:4, p.196-205; 1997/12 (ISSN 0178-3564)

[Se93] Seelbach, Wolfgang: Adreßraum-Partitionierung; Datenbank-Rundbrief, Mitteilungsblatt der GI-Fachgruppe Datenbanken (GI-FG 2.5.1) 11, p.5-8; 1993/03

[DBI] Kelter, U.: Lehrmodul "Einführung in Datenbanksysteme"; 1999/10

[IRA] Kelter, U.: Lehrmodul "Integrationsrahmen für Software-Entwicklungsumgebungen"; 1999/11

[SEU] Kelter, U.: Lehrmodul "Software-Entwicklungsumgebungen"; 1999/11

Index