Entwicklung verteilter Anwendungen mit IBM VisualAge For Smalltalk am Beispiel eines Pausenplaners

Studienarbeit
am Fachbereich Informatik
der Universität Hamburg

Arbeitsbereich Softwaretechnik
bei Dr. Guido Gryczan

vorgelegt von

Holger Breitling
Matrikel-Nr. 4625490

Mai 1999

Begriffe und Abbildungen: Viele der in dieser Arbeit verwendeten Fachbegriffe entstammen der objektorientierten Modellierung bzw. Konstruktion und werden z.B. im Kapitel 2 von [Zül98] zum großen Teil beschrieben (die sich hinter den eckig geklammerten Kürzeln verbergenden Literaturverweise können im Literaturverzeichnis nachgeschlagen werden, das sich am Ende der Arbeit befindet). Die in der Arbeit vorkommenden Klassen- und Sequenzdiagramme sind UML-konform (evtl. um erläuternde Elemente ergänzt) - siehe hierzu z.B. [Oes97]. Einige Abbildungen sind rein illustrativ und nicht formal.

Inhaltsverzeichnis

1 Einleitung

In dieser Arbeit möchte ich die Konstruktion verteilter Anwendungen mit VisualAge for Smalltalk (im folgenden meist mit „VA“ abgekürzt) im Rahmen der durch die Werkzeug-Material-Metapher formulierten Entwurfsprinzipien diskutieren. Besonderes Augenmerk lege ich dabei auf solche Anwendungen, bei denen konkurrente Zugriffe auf von mehreren Benutzern bearbeitbare Materialien auftreten.

Entlang einer Beispielanwendung, dem „Pausenplaner“, entwickle ich Schritt um Schritt die Fragestellungen, die vor allen Dingen die Bestimmung eines passenden Benutzungsmodells für die Anwendung, und die fachlichen und technischen Aspekte der Realisierung von Verteilung mit VA zum Gegenstand haben.

Dies ist das Vorgehen, Abschnitt für Abschnitt:

1. Im diesem ersten Abschnitt stelle ich den WAM-Ansatz kurz vor, der den konzeptionellen Rahmen für die weiteren Erörterungen und Entwürfe darstellt.
2. Der zweite Abschnitt dient mir dazu, verschiedene Kooperationsmodelle zu untersuchen, Modelle also, die beschreiben, wie die Benutzer eines verteilten Systems ihre Arbeit mit Hilfe dieses Systems organisieren.
3. Der dritte Abschnitt stellt die Beispielanwendung „Pausenplaner“ und die an sie gestellten Anforderungen vor. Die im vorigen Abschnitt bereitgestellten Begriffe werden verwendet, um ein passendes Kooperationsmodell für den Pausenplaner festzulegen. Dabei werden auch Werkzeuge und Materialien der Anwendung identifiziert und beschrieben.
4. Anschließend informiere ich über das *Distributed Feature* von VA – denn dieses stellt hinsichtlich der Verteilung den technischen Rahmen dar, in dem ich mich mit dieser Arbeit bewege – und kontrastiere es mit der Common Objects Request Broker Architecture (CORBA).
5. Dieser Abschnitt klärt die Begriffe „Original“ und „Kopie“ für den Pausenplaner und entwickelt ein Modell für den Zugriff des Benutzers auf die gemeinsam (kooperativ) genutzten Materialien.
6. Der vorletzte Abschnitt stellt zunächst Ereignismechanismus und den Umgang mit asynchronen Methodenaufrufen in der Anwendung dar und beschreibt dann die dreischichtige Architektur des von mir gebauten „Pausenplaner-Archivs“, welches ich nach den in den vorigen Kapiteln dargestellten Erkenntnissen gestaltet habe und das ein Kernstück der Applikation ist.
7. Dieser Abschnitt faßt die Ergebnisse der Arbeit zusammen
8. Der letzte Abschnitt ist ein Anhang und behandelt die im Verlauf der Implementierung hervorgetretene Problematik der Persistenz komplexer Materialien, also der Persistenz von Materialien, die sich aus mehreren unabhängigen Materialien zusammensetzen und der Sicherung ihrer Konsistenz.
9.

1.1 Was ist WAM?

Diese Arbeit ist vor dem Hintergrund des WAM-Ansatzes entstanden., der am Arbeitsbereich Softwaretechnik des Fachbereiches Informatik der Universität Hamburg entwickelt wurde und weiterentwickelt wird.

Mit dem Kürzel „WAM“, der für „Werkzeug-Automat-Material“ steht, sind drei zentrale Entwurfsmetaphern (mehr zu diesem Begriff später) namensgebend für einen Ansatz geworden, der die Anwendungsorientierung in den Mittelpunkt der Entwicklung von Software stellt.

Auf der Grundlage objektorientierter Entwurfs- und Konstruktionstechniken vereinigt der Ansatz so verschiedene Bestandteile wie ein Leitmotiv mit Entwurfsmetaphern, anwendungsorientierte Dokumente und evolutionäre Vorgehensweise mit Prototyping.

Da diese Arbeit nicht aus einem konkreten Anwendungsfall entstanden ist, sind natürlich auch nur Teile des gesamten WAM-Ansatzes zur Anwendung gekommen. Es handelt sich hierbei um die zentralen Ideen des Werkzeug & Material-Ansatzes, die Gegenstände und Konzepte des (in diesem Falle selbst gestellten) Anwendungsbereiches als Grundlage des softwaretechnischen Modells nehmen sowie der Rückgriff auf die zentralen Leitbilder und Entwurfsmetaphern. Außerdem beziehe ich mich in dieser Arbeit auch auf die Erweiterung der WAM-Entwurfsmetaphern für die Unterstützung kooperativer Arbeit. Deswegen gibt es in dieser Arbeit mehrfach Verweise auf [Zül98], „Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- und Materialansatz“.

Auch die folgenden grundlegenden Definitionen sind diesem Buch entnommen:

Definition: *Leitbild*

Ein Leitbild in der Softwareentwicklung gibt im Entwicklungsprozess und für den Einsatz einen gemeinsamen Orientierungsrahmen für die beteiligten Gruppen.

Es unterstützt den Entwurf, die Verwendung und die Bewertung von Software und basiert auf Wertvorstellungen und Zielsetzungen.

Ein Leitbild kann konstruktiv oder analytisch verwendet werden.

Definition: *Entwurfsmetapher*

Eine Entwurfsmetapher ist eine bildhafte, gegenständliche Vorstellung, die ein Leitbild fachlich und konstruktiv „ausgestaltet“, d.h. konkretisiert

Eine Entwurfsmetapher strukturiert die Wahrnehmung und trägt zur Begriffsbildung bei. Sie leitet die Vorstellung und Kommunikation über das, was fachlich analysiert, modelliert und technisch realisiert werden soll.

Eine Entwurfsmetapher dient der Gestaltung von Softwaresystemen, indem sie Handhabung und Funktionalität für die Beteiligten verständlicher macht.

Eine Entwurfsmetapher hat im WAM-Ansatz immer auch eine technisch konstruktive Interpretation in Form von Konstruktionsanleitungen und Entwurfsmustern

1.2 Das Leitbild vom Arbeitsplatz für qualifizierte menschliche Arbeit

Die Methode WAM bildet eine Grundlage für die Entwicklung von Software, die qualifizierte menschliche Arbeitstätigkeit unterstützen soll.

Damit sind folgende Grundannahmen verbunden:

- Die Benutzer bzw. zukünftigen Benutzer der Software besitzen Fachkompetenzen, die sich in einer gewissen Selbständigkeit und Eigenorganisation ihrer Arbeit niederschlagen.
- Die Entscheidungskompetenz der Benutzer betreffend die Organisation ihrer Arbeit befähigt sie, über Art und Umfang des Einsatzes von Softwarewerkzeugen zur Unterstützung ihrer Arbeit für jede Aufgabe immer wieder neu zu entscheiden. Daraus ergibt sich die Forderung nach individuell einstellbaren Systemen, deren Komponenten hohe Flexibilität bieten.

Ein beachtlicher Teil z.B. der Büroarbeitsplätze in Deutschand entspricht diesen Annahmen.

Diese Beschreibung des Einsatzgebietes dient als Leitbild, auf das ausgerichtet Software entwickelt werden soll.

Es gibt natürlich auch ganz andere Leitbilder, z.B. das einer Fabrik, bei dem es nicht darum geht, qualifizierte Arbeit durch einen Arbeitsplatz geeignet zu unterstützen, sondern darum, menschliche Arbeit zu automatisieren und zu kontrollieren.

1.3 Werkzeug und Material

Die zentralen Entwurfsmetaphern in WAM bilden Werkzeug und Material . (Die dritte namensgebende Metapher, der Automat, wird in dieser Arbeit kaum eine Rolle spielen). Dabei sind diese Gegenstände in intuitiver Weise aus der realen Welt zu übertragen:

Dem Benutzer soll Software bereitgestellt werden, die ihn bei der Bearbeitung von Aufgaben mit Funktionalität und Bedienbarkeit unterstützt, die nahe bei „natürlichen“ Konfigurationen wie „Hammer und Nagel“, „Pinsel und Leinwand“ usw. einzuordnen sind.

Die Gegenstände, an deren Erstellung und Bearbeitung der Benutzer primär interessiert ist, bilden in den WAM-orientierten Entwürfen die Materialien, während die Dinge, die die Funktionalität zur Erstellung und Bearbeitung der Materialien bieten, als Werkzeuge betrachtet werden. Im Gegensatz zu Werkzeugen sind Automaten Arbeitsmitttel, die nach dem Starten ein Material in einer definierten Folge von Arbeitsschritten (und ohne weitere äußere Eingriffe) bearbeiten und so Routinetätigkeiten erledigen.

Die Werkzeug-Material-Metapher soll bei allen Tätigkeiten im Wege der Softwareentwicklung beachtet werden. Als Konsequenz ergibt sich beispielsweise, daß niemals von einer Bijektion (also einer eineindeutigen Abbildung im mathematischen Sinne) zwischen Software-Werkzeugen und -Materialien ausgegangen werden sollte (ein Messer kann schließlich Gemüse schneiden oder Holz schnitzen, und auf eine Leinwand kann man auch mit einem Spachtel Farbe aufbringen statt mit einem Pinsel - genauer besehen erhalten man hier also nicht einmal eine Abbildung zwischen Werkzeugen und Materialien, gleich in welcher Richtung, sondern nur eine Relation). Die mit dieser Sichtweise verbundene Entkoppelung von Werkzeug und Material wird durch *Aspekte* modelliert (s. Abschnitt 1.4).

Anmerken möchte ich, wie gut sich die zentralen Entwurfsmetaphern mit dem Leitbild vertragen: Der *qualifizierte* Benutzer wird nicht als Be-Diener einer Maschine gesehen, die *ihm* Vorgaben

macht, sondern als jemand, der für die zu erledigenden Aufgaben die passenden Materialien bestimmt und aus einem großen Werkzeugkasten die angemessenen Werkzeuge für jeden Arbeitsschritt herausgreift.

Diese Sichtweise legt eine partizipative (also die Anwender beteiligende), evolutionäre Vorgehensweise bei der Softwareentwicklung nahe. Und genau in diesem Prozeß sind die Entwurfsmetaphern Werkzeug und Material noch einmal von großem Nutzen: Sie bilden eine begriffliche Basis, auf der Entwickler und Benutzer miteinander kommunizieren können, denn die Klassifizierung von Gegenständen nach einem Werkzeug-Material-Schema ist tief in unserer Kultur verankert und somit allgemeinverständlich.

1.4 Aspekte

Wenn man davon ausgeht, daß Werkzeuge und Materialien in der physischen Welt konzeptionell in keiner bijektiven Beziehung stehen und es also auch in unserer Modellierung nicht sollen, dann stellt sich die Frage, wie sich diese flexible Art des Zueinander- und Nichtzueinander-Passens konzeptuell formulieren lässt. WAM schlägt hierzu sogenannte*Aspekte vor.*

Ein Aspekt beschreibt Bearbeitbarkeitseigenschaften eines abstrakten Materiales. Wenn ein konkretes Material, die durch einen Aspekt definierten Eigenschaften besitzt, also die durch den Aspekt festgelegte Schnittstelle unterstützt, dann sagt man, daß es diesem Aspekt*genügt.* Ein Material kann mehreren, kann beliebig vielen Aspekten genügen.

Werkzeuge werden nun im Grundsatz nicht auf Materialklassen, sondern auf Aspekte hin entworfen, also so, daß sie Materialen bearbeiten können, die einem oder mehreren bestimmten Aspekten genügen. Damit ist die Hoffnung verbunden, in den formulierten Aspekten die richtigen Abstraktionen vorzunehmen, um die zu konstruierenden Werkzeuge und Materialien möglichst vielseitig miteinander verwenden zu können.

Auf der Implementationsseite finden Aspekte ihren Ausdruck als Schnittstellendefinitionen, wann immer dies eine Programmiersprache erlaubt. So werden Aspekte in C++ als rein abstrakte Klassen definiert und in Java als Interfaces.

Da es bei dieser Studienarbeit um eine Implementation in Smalltalk handelt, möchte ich an dieser Stelle die hier vorhandenen Probleme mit Aspekten nicht unerwähnt lassen: Leider bietet Smalltalk keine statische Typprüfung, so dass es nicht möglich ist, innerhalb einer Klassendefinition auszudrücken, daß eine Klasse mehrere Schnittstellendefinitionen erfüllt. In [Zül98] wird als eine Möglichkeit zur Unterstützung von Aspekten in Smalltalk ein*Aspect Browser* vorgeschlagen, dessen Realisierung in [Gat96] ansatzweise dargestellt wird.

Man kann Aspekte aber in jedem Fall bei der objektorientierten Modellierung verwenden, auch wenn sie dann bei der Implementation keinen Ausdruck in eigenen programmiersprachlichen Objekten oder Prüfmechanismen finden. Dies erfordert dieselbe (erhebliche) Sorgfalt, die ganz allgemein im Umgang mit dieser Sprache notwendig ist, um Laufzeitfehler zu vermeiden.

1.5 Konstruktion von Werkzeugen: Trennung von Interaktion und Funktion

Ein Werkzeug stellt sich selbst und die Materialien, die es bearbeitet, in einer Form dar, die es dem Benutzer erlaubt, interaktiv sondierende und verändernde Aktionen an den Materialien vorzunehmen.

Dabei werden beim WAM-Ansatz die Teile eines Werkzeuges, die mit der Benutzer-Interaktion

befaßt sind, und jene, die die Funktionalität implementieren, also auf den Materialien operieren, voneinander getrennt: Ein Werkzeug nach WAM besteht aus einer (evtl. sogar mehreren) Interaktionskomponente(n) *(IAK)* und einer Funktionskomponente *(FK)*. Dies kann als Fortentwicklung des MVC-Designs ("Model-View-Controller") betrachtet werden (s. Darstellung von MVC in [Gol89]).

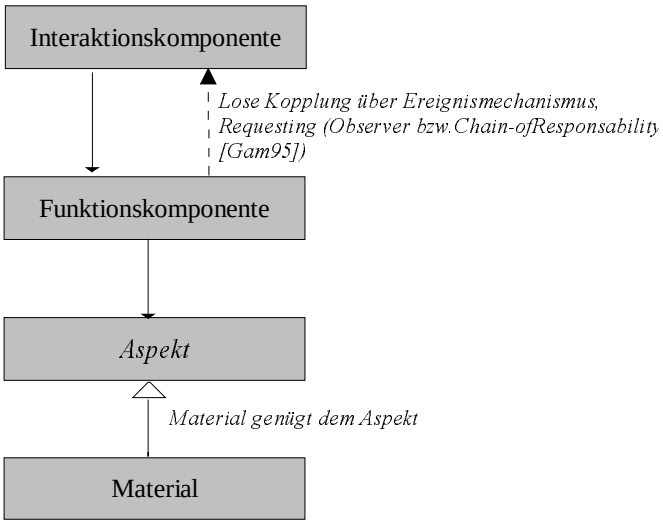

Abbildung 1 Trennung von Interaktion und Funktion (Klassendiagramm und Benutzt-Beziehungen)

Dabei kennt eine IAK stets ihre FK, umgekehrt ist dies jedoch nicht der Fall. Die notwendigen Informationen über Veränderungen an den Materialien gelangen durch einen Ereignismechanismus von der FK zu den IAKs (s. 6.1). Dieses Vorgehen verhindert, daß in der FK Annahmen über die Art der Darstellung des Werkzeugs gemacht werden und garantiert erst die gewünschte Entkopplung von Interaktion und Funktion, die angestrebt wird.

Eine weitere Ausdifferenzierung der Konstruktionsweise sieht vor, Werkzeuge aus Subwerkzeugen zu kombinieren. Dies führt zu einer Differenzierung in (Blatt-)IAK/FK, Kontext-IAK/FK und Wurzel-IAK/FK, die nach dem*Kompositum-Muster* zusammengesetzt werden können (s. [Gam95]).

Hier kann zusätzlich zum Ereignismechanismus ein Requesting-Mechanismus Verwendung finden, der ein Stellen von Anfragen der Subwerkzeuge an ihre Kontextwerkzeuge nach dem Muster der Chain-of-Responsibility (ebenfalls [Gam95]) vorsieht.

Ausführliche Darlegungen zur Konstruktion interaktiver Benutzeroberflächen im WAM-Kontext finden sich in [Rie97a] und [Zül98].

1.6 Abschließende Bemerkung

Bei der Konstruktion der Pausenplaner-Anwendung, die mit dieser Arbeit beschrieben wird, fanden die in diesem Kapitel beschriebenen Prinzipien und Vorgehensweisen Anwendung. Außerdem wird im weiteren Verlauf der Arbeit auch auf die vorgestellten Begriffe zurückgegriffen.

2 Kooperationsmodelle

Die Pausenplaner-Anwendung, deren Konstruktion im Rahmen dieser Arbeit dargestellt wird, ermöglicht eine kooperative Arbeitsteilung zwischen der Verwaltung des Lehrkörpers (also auch der Verwaltung der Informationen über jeden einzelnen Lehrer eines Kollegiums) und der eigentlichen Erstellung des Pausenplans.

Während der Ausgangspunkt der Softwareentwicklung nach WAM historisch einmal der Einzelarbeitsplatz war, behandelt der WAM-Ansatz seit geraumer Zeit auch die Abbildung und Unterstützung von kooperativer Arbeit sowie deren Koordinierung durch Software. Die entsprechenden Begriffsbildungen werden im folgenden vorgestellt und gehen im wesentlichen auf [Zül98] zurück.

2.1 Kooperative Arbeit

Unter *kooperativer Arbeit* wird zunächst ganz allgemein verstanden, daß verschiedene Personen geplant und koordiniert zusammenarbeiten, um ein bestimmtes Ergebnis zu erreichen.

Die wichtigsten Merkmale solcher Kooperationen sind:

- Die beteiligten Personen oder Stellen müssen sich *bestimmte Ressourcen teilen*. Im einfachsten Fall ist dies zum Beispiel das Dokument oder Produkt, das nach einer Reihe von Bearbeitungsschritten schließlich das Ergebnis der kooperativen Arbeit darstellen wird.
- Die *Arbeitshandlungen* der Beteiligten müssen unter Umständen *zeitlich und/oder inhaltlich aufeinander abgestimmt* werden
- Um die beiden vorigen Ziele zu verwirklichen, müssen sich die Beteiligten darüber *verständigen, was wann wie von wem getan wird.*

Die obige Definition von kooperativer Arbeit und die Nennung ihrer Merkmale haben noch keinen Bezug zur Softwaretechnik. Bei der Entwicklung von Anwendungssoftware in diesem Umfeld muß nun entschieden werden, wieweit die Kooperationssituation in der Software abgebildet und unterstützt wird. Tatsächlich kann man eine Kategorisierung dahingehend vornehmen, welche der Merkmale kooperativer Arbeit in der Software sichtbar werden.

Die obige Definition von kooperativer Arbeit und die Nennung ihrer Merkmale haben noch keinen Bezug zur Softwaretechnik. Bei der Entwicklung von Anwendungssoftware in diesem Umfeld muß nun entschieden werden, wieweit die Kooperationssituation in der Software abgebildet und unterstützt wird. Tatsächlich kann man eine Kategorisierung dahingehend vornehmen, welche der Merkmale kooperativer Arbeit in der Software sichtbar werden.

Man unterscheidet darum zwischen zwischen zwei Kooperationsmodellen, zwischen *impliziter* und *expliziter Kooperation:*

- Implizite Kooperation: Der konkurrierende Zugriff auf Materialien von verschiedenen Arbeitsplätzen aus wird deutlich (mehr nicht).
- Explizite Kooperation: Die zeitliche und inhaltliche Koordination wird unterstützt und vergegenständlicht, z.B. durch „elektronische“ Laufzettel, Vorgangsmappen oder Archive.

Natürlich gibt es fließende Übergänge zwischen den beiden Modellen, was auch im folgenden deutlich werden wird.

2.2 Implizite Kooperation

WAM orientiert sich stets an bewährten Vorgehensweisen aus der Arbeitswelt. Wenn eine Anwendungssoftware nun nur implizite Kooperation unterstützen soll (in ihrem Rahmen also zwar die Materialien und ihre Bearbeitung durch Werkzeuge modelliert werden, die kooperativen Umgangsformen in Hinblick auf dieses Material aber außerhalb des Systems bleiben), so muß demnach darauf geachtet werden, daß die Arbeitsmaterialien weiterhin mindestens diejenigen Kooperationsformen zulassen, wie ihre physischen Vorbilder auch.

Eine wichtige Eigenschaft physischer Materialien (Aktenordner, Karteikarte usw.) ist, daß sie zu einer Zeit nur an einem Ort sein können. Es gibt also keine gleichzeitige, voneinander unabhängige Bearbeitung solcher Gegenstände. Außerdem kann man, wenn ein Ordner entliehen ist, annehmen, daß jemand anderes im Moment (vor kurzem, sehr bald) damit arbeitet, vielleicht ist sogar ersichtlich, wer dieser jemand ist. Beides sind jedenfalls wichtige implizite Informationen in Hinblick auf kooperative Tätigkeiten.

Vielfach werden Ordner, Listen und ähnliche Dinge auch zum Zweck der Synchronisation von Arbeitsabläufen eingesetzt: etwa, wenn ein bestimmter Brief erst dann an eine Firma versendet werden darf, wenn ein Formular zu dieser Firma in einem speziellen Ordner eingeheftet worden ist.

Wenn man den Rahmen solcher Kooperationssituationen nicht zerbrechen will, und wenn man dem Anwender ein vertrautes Benutzungsmodell als Ausgangsbasis anbieten möchte, muß also sichergestellt werden, daß

- ein Material zu einer Zeit immer nur von einem Benutzer resp. nur an einem Arbeitsplatz bearbeitet werden kann
- die Materialien im Rahmen der Softwäre ähnliche Eigenschaften im Hinblick auf primitive Kooperationsunterstützung aufweisen wie ihre realen Vorbilder (Beispiel: ein Benutzer hat die Möglichkeit, eine von ihm veränderte Mappe in einem Aktenschrank zur Kenntnisnahme „nach vorne“ zu hängen)

Die Koordination der kooperativen Arbeit aber findet in jedem Fall außerhalb der Software statt, z.B. über Telefon, „Flurfunk“ oder andere Konvention.

2.3 Explizite Kooperation

Da ich mich in dieser Arbeit hauptsächlich mit impliziter Kooperation beschäftige, möchte ich die explizite Kooperation im Wesentlichen nur gegen die implizite abgrenzen:

Im Gegensatz zur impliziten Kooperation wirden bei der expliziten die zeitliche und inhaltliche Koordination vergegenständlicht. Das kann z.B. bedeuten, daß Materialien (evtl. mit (elektronischen) Bemerkungen versehen) mit Hilfe des Systems zwischen den Bearbeitern hin- und hergeschickt werden. Dabei kann die Software bestimmte Arbeitsabläufe oder Bearbeitungsphasen vorsehen, die schließlich zum gewünschten Ergebnis führen. Charakteristisch ist, daß hierbei die Beteiligten im System einander explizit sichtbar gemacht werden und sogar Möglichkeiten direkter Interaktion vorgesehen werden. Das genau ist bei der impliziten Kooperation nicht der Fall.

Explizite Kooperation kann beispielsweise über elektronische Laufzettel oder Umlaufmappen, Archive mit erweiterten Fähigkeiten oder Postfächer erfolgen. Laufzettel sind eine Vergegenständlichung von Prozeßmustern. Eine ausführliche Begründung dieses Konzepts in Abgrenzung zu der in Workflow-Systemen implementierten Kooperationsmodelle findet sich in [Gry96] beschrieben.

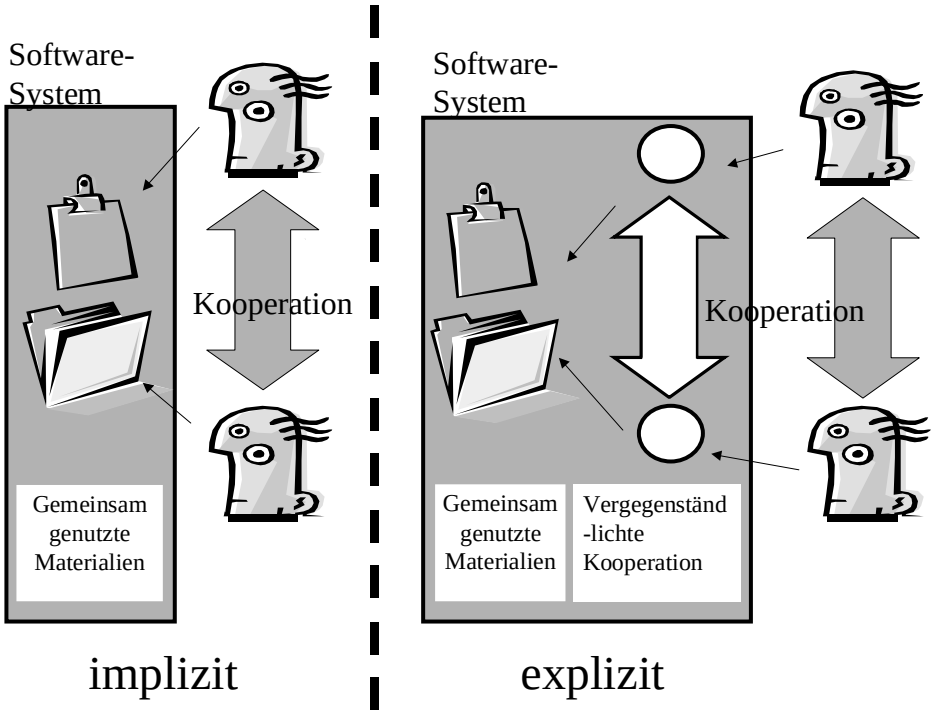

Abbildung 2 Implizite vs. explizite Kooperation

Die Grafik verdeutlicht den Unterschied: Im impliziten Fall spielen sich die für die Kooperation entscheidenden Absprachen und Koordinierungen außerhalb der Software ab; allein die so entstandenen Konventionen ermöglichen das Funktionieren der Kooperation im Software-System.
Im expliziten Fall dagegen werden Teilaspekte der kooperierenden Benutzer im System abgebildet, und damit findet auch ein Teil der Kooperationssteuerung im System statt und wird explizit.

2.4 Das Archiv als Medium impliziter Kooperation

Ein Konzept für die Softwareunterstützung von impliziter Kooperation, das sich an (nicht softwareunterstützten) bereits vorhandenen Kooperationsformen orientiert, ist das des (einfachen) gemeinsamen Archivs. Genau dieses Konzept wird in der Pausenplaner-Anwendung zur Kooperationsunterstützung verwendet:

Ein Archiv dient dazu, darin Materialien aufzubewahren und zu verwalten. Zu diesem Zweck gehört zu einem Archiv eine Inventarliste. Das Entleihen eines Materials aus dem Archiv führt zu einem entsprechenden Eintrag in dieser Liste, ebenso wie die Rückgabe. Ein gemeinsames Archiv ist nun eines, das von mehreren Benutzern gemeinsam verwendet wird, gemessen am physischen Vorbild also der Normalfall

Das Archiv ist auch als Objekt in einem Software-Entwurf als Medium zur impliziten Kooperation geeignet, denn es garantiert die eindeutige Zuordnung von Zeit und Ort bzgl. der Materialien und ermöglicht über die Entleihinformationen in der Inventarliste eine Koordinierung der Arbeit außerhalb des Systems. Dabei liefert es dem Anwender ein fachlich nachvollziehbares Benutzungsmodell, daß sich am physischen Vorbild orientiert.

Abgesehen von den rein mimetischen Aspekten des Archivs erlaubt das in der Pausenplaner-Anwendung verwendete Archiv, Kopien eines entliehenen Materials zu erhalten. Außerdem signalisiert das Archiv einem Benutzer, der die Kopie eines Materials entliehen hat, die Rückgabe des Originals.

3 Beispielanwendung: der Pausenplaner

In dieser Arbeit soll die Implementation eines auf einem Archiv basierenden Kooperationsmodells mit VA vorgestellt werden. Als für Beispielzwecke gut geeignete Aufgabenstellung hat sich im WAM-Kontext schon mehrfach der Pausenplaner erwiesen (so in [Fel97] und als immer wieder auftauchendes Beispiel in [Zül98]).

Da es sich hier nicht um einen konkreten Anwendungsfall handelt, sondern um eine problembezogene Abstraktion, verzichte ich auf künstliche Detaillierungen, die für die Verwendung der in [Zül98] vorgestellten WAM-Dokumenttypen wie Szenarios und Systemvisionen nötig wären, und beschreibe die verschiedenen Aspekte der Aufgabenstellung in angemessen reduzierter Form.

3.1 Die Problemstellung im Kern

Im deutschen Schulalltag sind Pausen zwischen den Unterrichtsstunden vorgeschrieben. Die Schüler können diese Pausen an verschiedenen ausgezeichneten Orten und Plätzen auf dem Gelände ihrer Schule verbringen, und diese Orte müssen währenddessen von Lehrern (oder anderen berechtigten Personen) beaufsichtigt werden.

Damit steht eine jede Schule vor der Aufgabe, über die Vergabe der Aufsichtspflichten an die Lehrer zu entscheiden. Da sich normalerweise der Stundenplan im wöchentlichen Rhythmus wiederholt, ist es sinnvoll, diesen Turnus auch auf den Aufsichts- bzw. Pausenplan anzuwenden. Es muß also eine Wochenschablone für die Vergabe der Aufsichtspflichten erstellt werden. Und genau dies soll der Pausenplaner erleichtern.

3.2 Überblick: Bisherige Pausenplanung

Einige Zeit vor Beginn des neuen Schuljahres sind die Lehrer aufgefordert, diejenigen Zeiten zu benennen, in denen ihnen eine Pausenaufsicht unmöglich ist. Außer individuellen Gründen spielt hier natürlich auch der neue Stundenplan eine Rolle, so daß ein Teil dieser Ausschlußzeiten gleich von demjenigen angemeldet werden, der den Stundenplan entwirft.

Die Ausschlußzeiten werden jedenfalls im Sekretariat in einem Karteikasten gesammelt und geordnet. Dort werden auch Mappen für jeden Lehrer geführt, die über Stundenzahl, Krankmeldungen etc. Auskunft geben.

Der Pausenplan-Verantwortliche - der mit seiner Arbeit natürlich nicht warten kann, bis sich der letzte Kollege bequemt hat, seine Wünsche anzumelden - holt sich, wann immer er seinen Plan weiterentwickeln will, den Karteikasten mit den Ausschlußzeiten aus dem Sekretariat. Bei dem Plan, der nun Schritt für Schritt an einer großen Pinwand entsteht, versucht er sicherzustellen, daß sich die Last gerecht auf die Kollegen verteilt. Wann immer sich Auschlußzeiten ändern oder neue dazukommen, werden die entsprechenden Karteikarten von der Schulsekräterin markiert, so daß der Pausenplan-Verantwortliche dies nicht übersehen kann.

3.3 Pausenplanung mit der Pausenplaner-Anwendung

Der Planer soll durch Software dabei unterstützt werden, die Pausen in Übereinstimmung mit den Ausschlußzeiten zu belegen. Dabei soll insbesondere auch eine gerechte Verteilung der Aufsichten durch Software erleichtert werden.

Die Erfassung der Ausschlußzeiten und die eigentliche Planungstätigkeit sollen wie bisher von zwei (oder ggf. mehr) Personen durchführbar sein, auch wenn diese sich nicht automatisch persönlich begegnen. Die Verwaltung der Lehrer und die Erfassung von Ausschlusszeiten soll also weiterhin

getrennt von der Erstellung der Aufsichtsplanung vor sich gehen können. Das bedeutet, daß Änderungen in den Ausschlußzeiten, Veränderungen in der Zusammensetzung des Kollegiums usw., die von der Schulsekretärin eingegeben werden, für den Planenden deutlich sichtbar werden müssen, so daß er sein Vorgehen tatsächlich darauf einstellen muß.

Zudem sollte die Tätigkeit von Sekretärin und Planer gleichzeitig an verschiedenen Orten, also verteilt, möglich sein. Die Sicherung der Konsistenz des Planes und die implizite Kooperation soll über ein (im Bezug auf Kooperationsunterstützung) einfaches, aber für Pausenplan- und Lehrkörperobjekte (s. nächsten Abschnitt) spezialiertes Archiv (Pausenplaner-Archiv) erfolgen. Diese implizite Kooperation durch den Zugriff auf das Archiv wird in einer Client/Server-Architektur realisiert. Damit ist gemeint, daß das Pausenplaner-Archiv auf einem ausgezeichneten Rechner (dem Server) gestartet wird und dann von anderen Rechnern (den Clients) aus, zugreifbar ist, auf denen Lehrkörper und Pausenpläne bearbeitet werden können.

Um die Vorstellung der zukünftigen Arbeit mit dem Pausenplaner zu entwickeln, müssen nun die Materialien identifiziert und die benötigten Werkzeuge definiert werden:

3.4 Die Materialien (die anwendungsfachlichen Klassen)

Ein detaillierter Vorschlag für die anwendungsfachlichen Klassen zum Pausenplaner findet sich in [Fel97]. In der von mir implementierten Version finden sich Abweichungen in Details, die den grundsätzlichen Charakter dieses Entwurfs jedoch nicht verändern und hier nicht diskutiert werden sollen. Um dem Leser ein Kernverständnis zu ermöglichen, will ich die wichtigsten drei Material-Klassen hervorheben:

Lehrer

Exemplare dieser Klasse enthalten die für die Anwendungen entscheidenden Informationen zu einem Lehrer. Das sind unter anderem sein Name, Vorname, sein Kürzel (das im Pausenplan angezeigt wird, wenn er durch den Benutzer in eine Pausenaufsicht eingetragen wird) und die Liste der Ausschlußzeiten.

Lehrkörper

Ein Lehrkörper enthält die Information über die an einer Schule beschäftigten Lehrer. Der Lehrkörper verwaltet also eine Liste von Lehrern und bietet Methoden, um dort Lehrer einzufügen, sie zu erhalten oder herauszulöschen. Dabei garantiert der Lehrkörper, daß die Kürzel der Lehrer stets eindeutig innerhalb des Lehrkörpers sind. Ein Lehrkörper hat einen Namen.

Pausenplan

Ein Pausenplan hält die Zuordnung von Pausenaufsichten zu den Lehrern fest. Dabei bezieht sich ein Pausenplan fest auf einen bestimmten Lehrkörper und ordnet den Pausenaufsichten Lehrer nur aus diesem Lehrkörper zu. Auch ein Pausenplan trägt einen Namen.

3.5 Die Werkzeuge des Pausenplaners

Der Pausenplaner bietet dem Benutzer vier Werkzeuge, mit denen er die Materialien bearbeiten und verwalten kann: Zu jedem der oben benannten Materialien gibt es genau ein Werkzeug; hinzu kommt

das Archivwerkzeug, welches dem Benutzer über das Archiv den Zugriff auf die Materialien (auf Lehrkörper und Pausenpläne) gestattet. Der durch die Verwendung von Aspekten erreichbare Wiederverwendungseffekt wird leider bei der vorliegenden Systemgröße noch nicht sichtbar.

3.5.1 Das Lehrer-Werkzeug

Das Lehrer-Werkzeug erlaubt es, die Daten für einen Lehrer einzugeben und zu editieren. Dabei können sein Name, Vorname, sein Kürzel, sein Arbeitspensum, ob er aufsichtsberechtigt ist oder nicht eingegeben werden. Außerdem können Auschlußzeiten in eine Liste eingetragen oder dort gelöscht werden.

Abbildung 3 Das Lehrer-Werkzeug

Das Lehrerwerkzeug ist ein Subwerkzeug des im folgenden Abschnitt kurz beschriebenen Lehrkörper-Werkzeugs. Wird es von dort aus für einen bestehenden Lehrer geöffnet, sind selbstverständlich dessen Attributwerte vorgeblendet. Soll ein neuer Lehrer eingefügt werden, sind die Einträge im Lehrer-Werkzeug leer.

3.5.2 Das Lehrkörper-Werkzeug

Das Lehrkörper-Werkzeug listet die im Lehrkörper vorhandenen Lehrer. Von hier aus kann ein Lehrer-Werkzeug geöffnet werden, um einen Lehrer zu editieren oder um die Eigenschaften eines neuen Lehrers einzutragen.

Ein neuer Lehrkörper kann erstmals ins Archiv gestellt, ein bereits im Archiv vorhandener kann dort aktualisiert oder gleich ganz zurückgestellt werden.

Abbildung 4 Das Lehrkörper-Werkzeug

Die Lehrer aus der Lehrkörper-Liste können per Drag'n Drop auf dem Pausenplan im Pausenplan-Werkzeug plaziert werden.

3.5.3 Das Pausenplan-Werkzeug

Das Pausenplan-Werkzeug ermöglicht es, aus einer Lehrer-Liste per Drag'n Drop Lehrer auf den Pausen des Pausenplanes zu plazieren. Es ist nicht möglich, Lehrer auf einer Pause abzulegen, zu der bei ihnen eine Ausschlußzeit vermerkt ist. Die Lehrer können auch von einem Lehrkörper-Werkzeug her gezogen werden, das sich über den Button „Lkp editieren“ öffnen läßt.

Das Pausenplan-Werkzeug kann so eingestellt werden, daß in der Lehrer-Liste stets nur eine Auswahl der fünf Lehrer erscheint, die im Sinne der Gerechtigkeit am ehesten eine weitere Aufsicht übernehmen müßten (wobei die Liste dann auch in diesem Sinne sortiert ist).

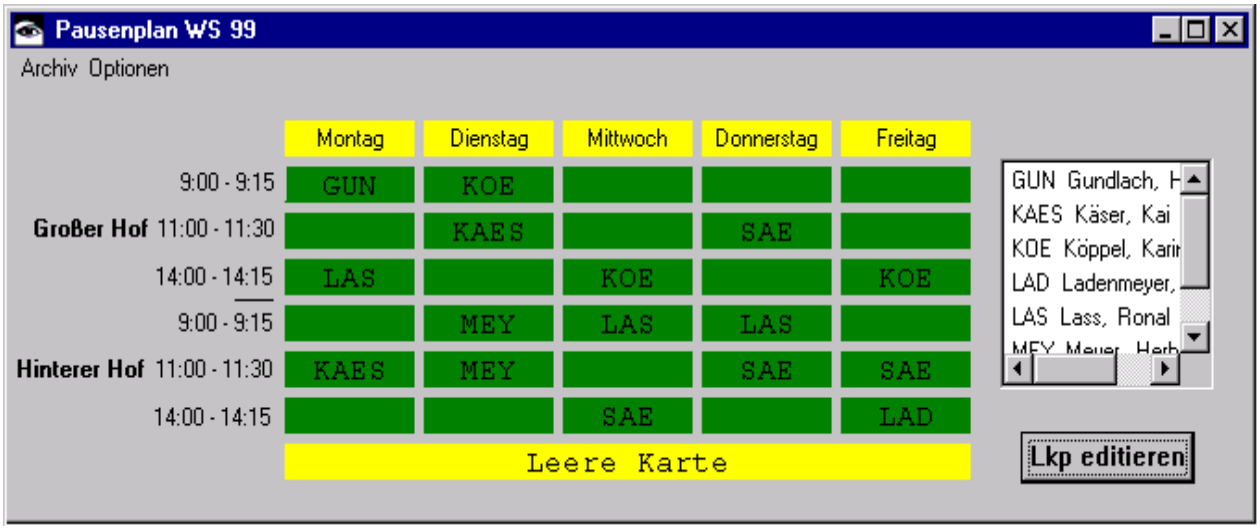

Abbildung 5 Das Pausenplan-Werkzeug

Das Protokoll zum Umgang mit dem Archiv ist wie bei dem Lehrkörper-Werkzeug.

3.5.4 Das Pausenplaner-Archiv-Werkzeug

Solange die Pausenplaner-Anwendung läuft, ist das Zugriffswerkzeug sichtbar. Nur von ihm aus kann die Anwendung beendet werden. Das Zugriffswerkzeug listet die vorhandenen Lehrkörper und Pausenpläne, die als Original oder Kopie (diese Begriffe sind noch genauer zu klären, s. dazu 5.1) entnommen werden können. In diesem Fall werden sie in den zugehörigen Werkzeugen geöffnet.

Das Archivwerkzeug kann zudem jederzeit anzeigen, welche Pausenpläne sich auf einen bestimmten Lehrkörper beziehen, und natürlich auch umgekehrt, auf welchen Lehrkörper sich ein bestimmter Pausenplan bezieht.

Genaueres zum Zugriffsmodell, welches das Archiv implementiert, in Abschnitt

3.6 Abschließende Bemerkungen

In diesem Kapitel wurde die Aufgabenstellung bei der Konstruktion der Pausenplaner-Anwendung formuliert, es wurden die zentralen Materialklassen vorgestellt und die Werkzeuge, welche die Anwendung im Zusammenspiel realisieren sollen. Ausserdem wurde verdeutlicht, daß das Pausenplaner-Archiv auf einem als solchen ausgezeichneten Server laufen wird.

Mit der (für diese Arbeit angemessen stark verengten) Einführung in den WAM-Ansatz, der Diskussion von Kooperationsmodellen und der Vorstellung der Aufgabenstellung „Pausenplanung“ sind die konzeptionellen Grundlagen für die weitere Detaillierung der zu gestaltenden Anwendungen gelegt. Es fehlt noch eine Vorstellung der technischen Gegebenheiten im Zusammenhang mit der Verwendung von VisualAge for Smalltalk und seinem Distributed Feature - und die folgt im nächsten Kapitel.

4 IBM VisualAge for Smalltalk und sein „Distributed Feature“

IBM VisualAge for Smalltalk ist eine Entwicklungsumgebung für Smalltalk, die vor allen Dingen für ihre Fähigkeit zu „visueller Programmierung“ bekannt ist. Das ist im Wesentlichen die Möglichkeit, in einem dramatisch erweiterten GUI-Builder über das Ziehen und Erzeugen von „Connections“ (die als Pfeile sichtbar gemacht werden) Objekte so miteinander zu verknüpfen, daß Benachrichtigungen im Sinne des Beobachter-Musters (s. [Gam95]) Methoden (die als „Aktionen“ ausgezeichnet sind) an ereignisempfangenden Objekten, aufrufen.

Mit VA for Smalltalk (im folgenden nur noch meist nur VA genannt) wird aber auch ein sogenanntes „Distributed Feature“ ausgeliefert. Dieses kann in das Image geladen werden und ergänzt alle Objekte um Fähigkeiten zu verteilten Methodenaufrufen. Mit der Installation des „Distributed Feature“ sind massive Eingriffe in grundlegende Klassen verbunden, so wird zum Beispiel das Protokoll von „Object“ um eine Reihe von Methoden erweitert.

So wird die Verteilung von Software-Objekten auf mehrere Rechner durch das „Distributed Feature“ realisiert:

- Image-globale Objekte und Objekte in Image-globalen Dictionaries können über sogenannte „Object References“ (welche sich aus TCP/IP-Adresse, Imagename und Objektname zusammensetzen) eindeutig identifiziert werden.
- Jedes Image kann einen eigenen Name Server verwenden, in dem Object References unter eindeutigen symbolischen Namen registriert werden.
- Für durch eine Object Reference eindeutig bezeichnete Objekte kann ein Shadow, das ist die VA-Bezeichnung für einen Proxy, erzeugt werden. Es handelt sich hierbei also i.a. um ein Stellvertreterobjekt im lokalen Prozessraum, das für ein Objekt in einem anderen Prozessraum bzw. auf einem anderen Rechner steht. Dieser Shadow kann alle Methodenaufrufe verstehen, die auch das Originalobjekt versteht und leitet sie an dieses weiter.
- Für Argumente wie auch Rückgabeobjekte von solcherart entfernt aufgerufenen Methoden werden (bis auf diejenigen eingebauten Klassen, deren Exemplare Wertsemantik besitzen) automatisch Shadows generiert. Das bedeutet ein enormes Maß an Transparenz (also Unsichtbarkeit des unterliegenden Mechanismus').

Im folgenden beschreibe ich die grundlegenen technischen Konzepte des „Distributed Feature“.

4.1 Genaue Darstellung des „Distributed Feature“

4.1.1 Object Spaces und der Activator

Object Spaces

Das *Distributed Feature* basiert auf dem Konzept der „Object Spaces“. Ein Object Space ist ein laufendes Smalltalk Image, oder, wie es IBM ausdrückt, „a collection of active Smalltalk objects loaded from a single Smalltalk image file“ (diese Formuierung ist aus [IBM97a]; durch die Unterscheidung des Begriffes „Image“ von einer laufenden Smalltalk-Umgebung soll deutlich werden, daß mehrere verschiedene Object Spaces auf Basis derselben Image-Datei erzeugt werden können).

Object Spaces können in zwei Modi laufen: im *Client-Modus* und im *Server-Modus*. Der

Unterschied hierbei ist folgender: Im Client-Modus kann der Object Space Verbindungen zu anderen Object Spaces initiieren, aber nur im Server-Modus kann er solche Anfragen auch beantworten.

Die Möglichkeit, eine Verbindung zu einem Object Space zu initiieren, bedeutet nichts anderes als die Möglichkeit, ein erstes Stellvertreterobjekt für ein in dem entfernten Object Space existierendes Objekt erzeugen zu können. Genau dazu muß der entfernte Object Space im Server-Modus laufen.

Wenn auf diese Art eine Verbindung hergestellt ist, kommunizieren entfernte Objekte aber direkt miteinander (transparent über die Stellvertreterobjekte, aber jedenfalls ungeachtet der Natur der Object Spaces). Das beinhaltet dann auch die Möglichkeit, Stellvertreterobjekte für solche Objekte zu erhalten, die in einem entfernten Objekt Space leben, der im Client-Modus läuft (etwa, wenn die Objekte als Parameter beim Methodenaufruf übergeben werden).

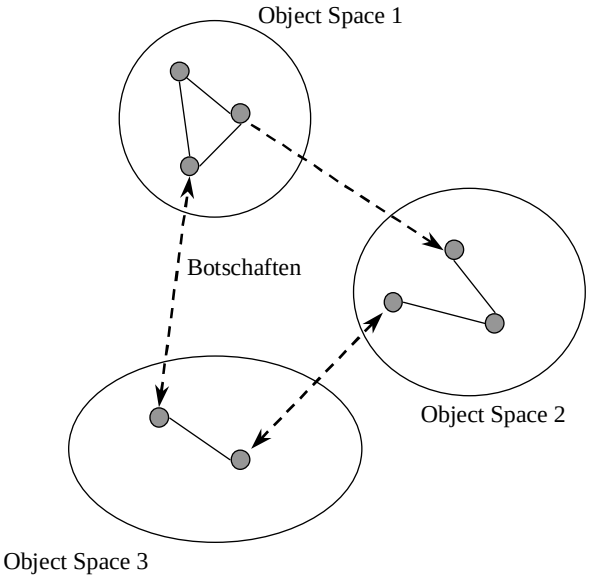

Abbildung 6 Object Spaces

Objekte in einem Object Space können nach der Initiierung der Verbindung zu einem anderen Object Space miteinander kommunizieren, ohne sich der Tatsache, daß sie sich in verschiedenen Object Spaces befinden, bewußt zu sein.

Der Activator

Der *Activator* ist ein Hintergrundprozeß, der sich auf jedem Rechner in Ausführung befinden muß, auf dem ein Object Space im Server-Modus ablaufen soll. Auch wenn mehrere solche Object Spaces gleichzeitig auf einem Rechner laufen, genügt *ein* Activator.

Der Activator ist dazu da, Anfragen anderer Object Spaces an den richtigen Object Space auf dem Rechner weiterzuleiten. Er kann sogar auch so konfiguriert werden, daß er einen angefragten Object Space neu startet, wenn er noch nicht läuft.

Um dies zu ermöglichen, besitzt der Activator eine Konfigurationsdatei, in der verschiedenen Image-Dateien (also Object Spaces) Namen zugeordnet werden. Anmerkung: Die 1-zu-1-Beziehung zwischen Image-Dateien und Object Spaces wird nur dann zerbrochen, wenn der Activator in der Konfigurationsdatei dazu angehalten wird, mit jeder Anfrage an eine bestimmte Image-Datei einen

neuen Object Space zu starten.

4.1.2 Object References

Eine Object Reference stellt eine Art Adresse für ein entferntes Objekt dar und ist die Vorstufe zu einem Stellvertreterobjekt. Man kann eine Object Reference durch einen Aufruf der folgenden Form erhalten (am Beispiel eines im globalen Dictionary bekannten Objekts, auf die ich mich im ganzen Abschnitt beschränken):

```
|anObjectReference|
anObjectReference := DsObjectReference dtFromRepString: aRepString.
```

Dabei ist `aRepString` ein String, welcher so aufgebaut ist:

```
'DtGlobalReference<<TCP:hostname:oskey>Global>'
```

Hierbei ist *`hostname`* zu ersetzen durch den Namen / die IP-Adresse des Hosts, *`oskey`* durch den passenden Object Space Key, welcher in der Konfigurationsdatei des Activators auf der Zielmaschine festgelegt ist, und *`Global`* durch den Namen des Objekts im globalen Dictionary.

4.1.3 Name Server

Jeder Object Space besitzt seinen eigenen Name Server (tatsächlich kann er sogar mehrere haben, diesen Fall betrachte ich hier aber nicht). In diesem können mit Hilfe von entsprechenden Browsern, die aus der VA-Entwicklungsumgebung heraus geöffnet werden können, die oben beschriebenen Object References einem festen Namen zugeordnet werden (nach dem Key-Value-Prinzip).

Um den Default Name Server zu erhalten, muß man zunächst dem globalen Objekt `NameServerList` die Message `default` senden:

```
|defaultNameServer|
defaultNameServer := NameServerList default.
```

Um vom Name Server nun die gewünschte Object Reference zu erhalten, geht man so vor:

```
|ORef|
ORef := (NameServerList default) referenceAt:
                              'RemoteName>>Reference(Global)'.
```

Dabei muß *`RemoteName`* durch den Namen ersetzt werden, der der Schlüssel für die Object Reference im lokalen Name Server ist.

4.1.4 Shadows

Eine Object Reference beschreibt nur die Lokalisation eines Objektes. Um an ein entfernten Objekt aber tatsächlich Nachrichten versenden zu können, braucht man ein Stellvertreterobjekt, von IBM *Shadow* genannt. Diesen kann man unter Verwendung einer Object Reference so erhalten:

```
|ShadowObject|
ShadowObject := ORef shadow.
```

Mit dem Shadow eines entfernten Objekts kann man nun so umgehen, als manipuliere man das Objekt direkt; der Shadow sendet alle Nachrichten, die das entfernte Objekt versteht, an dieses weiter, wobei eventuelle Argumente im entfernten Object Space als Shadows der hier lokalen Objekte auftreten (sofern es sich nicht um Exemplare eingebauter Klassen mit Wertsemantik handelt) und ein eventuelles zurückgegebenes Objekt in der Regel ebenfalls ein Shadow ist.

4.1.5 Lokale Kopien

VA bietet die Möglichkeit, lokale Kopien eines Objektes zu erzeugen. Es kann auch dafür gesorgt werden, daß ein von einer Methode zurückgegebenes Objekt lokal ist oder die an eine Methode übergebenen Parameter aus deren Sicht lokal.

Dieser Mechanismus wird durch Erweiterungen des Protokolls von*Object* erreicht. Dabei macht die Dokumentation von VA keine Aussage darüber, was genau eine lokale Kopie ist. Da auch kein Hinweis erfolgt, wie man den Kopiervorgang für eigene Objekte anpassen kann (um etwa die Tiefe einer Kopie festzulegen, vgl. [LaL94]), wird dieser Mechanismus von mir nicht verwendet.

4.1.6 Im Fehlerfalle...

Exception Handling

Visual Age for Smalltalk bietet ein Exception-Handling-Konzept, das im Ergebnis dem in C++ und Java ähnlich ist. Exceptions (das sind hier Unterklassen von`ExceptionalEvent`) können erzeugt werden und werden geworfen, indem man ihnen die Nachricht `signal` schickt. Um die geworfenen Exceptions fangen zu können, muß der Methodenaufruf in einen Block gekapselt werden, den man dann mit (potentiell mehreren) Nachrichten der Form `when:do:` beschickt. Ein Beispiel folgt nach dem nächsten Absatz

Remote Exceptions

Das *Distributed Feature* signalisiert eine`unavailableObjectException`, wann immer ein entferntes Objekt nicht erreichbar ist, ob der Grund nun der Absturz des Rechners ist, auf dem das Objekt beheimatet war, oder Fehler im Netzwerk.
Es folgt ein Codebeispiel für solches Exception Handling:

```
|oRef|
“Object Refernce holen“
oRef := (NameServerList default) referenceAtFirstIdentifier: 'MyObject'.
```

```
“Methode im Block koennte Exception werfen, falls das entfernte“
“Objekt nicht erreichbar ist“
[oRef shadow printString]

“Exception Handling“
when: (DsDistributedSystem unavailableObjectException)
do: [:signal | Transcript show:  'The remote Object is not available'.
      signal exitWith: nil].
```

Jeder der entscheidenden Abschnitte im Beispielcode ist kommentiert. Die Parallele zum Exception Handling in C++/Java wird deutlich: Der Aufruf an die möglicherweise eine Exception werfende Methode wird in einen Smalltalk-Block geklammert (bei C++/Java wäre es ein try-catch-Block) und mit einem oder mehreren when:do: werden die auftretenden Exceptions behandel (dies wären in C++/Java ein oder mehrere catch-Blöcke).

4.2 Vergleich mit CORBA

CORBA (Common Object Request Broker Architecture) ist eine von der Object Management Group (OMG) spezifizierte offene Architektur für verteilte Objekte. Ziel dieser Architektur ist es, die Interaktion von Objekten zu ermöglichen, die auf verschiedenen Rechnern, in verschiedenen Betriebssystemen laufen, in verschiedenen Programmiersprachen implementiert sind. Um die Kommunikation zwischen diesen zu ermöglichen, müssen die anzubindenden Objekte Schnittstellen implementieren, die zuvor in der von der OMG festgelegten Schnittstellendefinitions-Sprache (Interface Definition Language, IDL) erstellt worden sind. Die Kommunikation zwischen den verschiedenen Rechnern vollzieht sich dann über sogenannte ORBs (Object Request Broker). Eine kurze Einführung zu CORBA findet sich in [Fel97], außerdem eine ausführliche Darstellung einer CORBA-Implementation für Smalltalk. Ich möchte im Nachfolgenden die Unterschiede und Gemeinsamkeiten im Vergleich zum Distributed Feature von VisualAge for Smalltalk darstellen

Generierung

Die zur Verwendung mit CORBA gedachten Klassen müssen Schnittstellen implementieren, die in der IDL geschrieben und dem CORBA-Mechanismus zur Verfügung gestellt worden sind. Dazu werden aus der IDL mittels angepasster Compiler verschiedene Klassen generiert, unter anderem *Skeleton*- und *Stub*-Klassen zu jeder mit CORBA verwendeten Klasse. Dabei beschreibt eine Stub-Klasse die Stellvertreterobjekte zu einem entfernten Objekt auf der Client-Seite, während die Skeleton-Klasse die (Server-seitige) Superklasse der tatsächlichen Implementation darstellt.

CORBA bietet allerdings auch die Möglichkeit, dynamische Methodenaufrufe abzusetzen. In diesem Fall entfällt die Notwendigkeit der Generierung einer Stub-Klasse. Der Nachteil ist eine umständlichere Programmierbarkeit sowie der Verlust an Performance und statischer Typsicherheit.

Das Distributed Feature verlangt im Gegensatz dazu keine Bereitstellung zusätzlicher Klassen, um Objekte einer Klasse verteilt zugänglich zu machen

Code-Verteilung

Für den Regelfall statischer Methodenaufrufe müssen bei CORBA die Stub-Klassen zu jenen Klassen, welche auf Client-Seite entfernt verwendet werden sollen, auch dort vorhanden sein.

Aufgrund der dynamischen Natur des gesamten Smalltalk-Systems ist dagegen eine Code-Verteilung auf den Client beim Distributed Feature nicht erforderlich.

Interoperabilität

Das Distributed Feature von VisualAge for Smalltalk erlaubt nur und ausschließlich die Kommunikation zwischen VisualAge-for-Smalltalk-Umgebungen.

Implementationen der CORBA-Spezifikation gibt es hingegen für Smalltalk, C, C++, Ada, COBOL, Java, Objective C,..., wobei die Spezifikation garantiert, dass die ORBs verschiedener Hersteller miteinander kommunizieren können, auch wenn sie auf verschiedenen Rechnern mit verschiedenen Betriebssystemen laufen.

Fazit

Die beiden Verteilungsmodelle unterscheiden sich deutlich allein schon deshalb, weil mit ihnen ganz unterschiedliche Ziele verfolgt werden. Während CORBA die Integration verschiedenster objektorientierter oder sogar Legacy-Plattformen (z.B. COBOL-Applikationen) anstrebt und dabei sinnvollerweise (man bedenke die vielen unterschiedlichen Betriebssysteme und Programmiersprachen, für die eine ORB-Implementation wünschenswert wäre) als offener Standard konzipiert ist, ergänzt das Distributed Feature überzeugend die Rapid-Development-Philosophie von VisualAge: es ist eine perfekt auf die Programmiersprache zugeschnittene Smalltalk-Erweiterung, die sich nahezu unbemerkt einfügt und der Verteilung keinerlei nennenswerte technische Hürden aufbaut oder das Erlernen und Benutzen Dutzender Tools erforderte; es verdient wirklich die Bezeichnung transparent (im Sinne von unsichtbar) - ist aber eben nur für Smalltalk und dort auch nur für VisualAge for Smalltalk verfügbar.

Damit ist das Distributed Feature bestens geeignet für all jene verteilten Smalltalk-Anwendungen, welche mit Sicherheit in einer geschlossenen VisualAge-for-Smalltalk-Welt verbleiben und deren Funktionalität nicht anderer Software ausserhalb dieser Welt zur Verfügung gestellt werden muß. CORBA hingegen empfiehlt sich für Anwendungen, die „alten“ Code einbinden sollen oder aber unter Umständen in Zukunft wieder selbst in andere Applikationen eingebunden werden sollen.

4.3 Bewertung des Distributed Feature

Betrachtet man die vorangehenden Abschnitte, wird schnell deutlich, dass das „Distributed Feature“ von VA tatsächlich eine Möglichkeit zur Konstruktion verteilter Anwendungen bietet, die von vielen der technischen Hürden befreit ist, welche andere Lösungen in diesem Bereich aufbauen. Code-Generierung fällt beispielsweise weg: VA erbaut in generischer Weise für jedes Objekt Shadows, Stellvertreter also, die sich in Hinsicht auf die Schnittstelle als Obermenge des Originalobjektes, als Subtyp mithin, darstellen.

Damit erfüllt das Distributed Feature ein Versprechen, das man häufiger hört: „Sie können Ihre Anwendung erst lokal konzipieren, und danach verteilen Sie sie einfach, und im Handumdrehen haben Sie die gewünschte verteilte Anwendung“. Dieses Vorgehen ist technisch möglich, und dies charakterisiert die Fähigkeiten des Distributed Feature. Trotzdem ist das in dem Versprechen beschriebene Vorgehen so nicht zu empfehlen:

Auch wenn sich lokale und entfernte Objekte bezüglich ihrer Schnittstelle (also statisch) gleich verhalten, unterscheiden sie sich doch zur Laufzeit sehr: Zugriffe auf ein entferntes Objekt dauern länger und können aufgrund von Netzwerkfehlern fehlschlagen. Damit ist für viele Zwecke die Verwendung entfernter Objekte zu vermeiden. Außerdem müssen Wartezeiten oder Fehler aufgrund von Netzwerkproblemen einem Anwender auch signalisiert werden. Das bedeutet, dass bei der Verwendung entfernter Objekte in den jeweiligen Methoden meistens bekannt sein muss, dass die entsprechenden Objekte nicht lokal sind, um Exception Handling zu ermöglichen.

Für eine Anwendung, die zeiteffizient und robust sein soll, ist also ein bewußter Einsatz des Distributed Feature von VA nötig, der die von der Entwicklungsumgebung ermöglichte Transparenz (Unsichtbarkeit des Verteilungsmechanismus) um Wissen bzgl. der Verteilung ergänzen muß. Diese Schlussfolgerung habe ich bei der Konstruktion der Pausenplaner-Beispielanwendung umgesetzt.

4.4 Design-Konsequenzen

Wie im vorigen Abschnitt dargestellt, halte ich es für notwendig, dass in der Applikation entfernte Objekte nur mit dem Bewusstsein ihres nicht-lokalen Charakters verwendet werden, um die in 4.1.6 vorgestellten Mechanismen zur Fehlerbehandlung an allen diesen Verwendungsorten einsetzen zu können.

Da an dieser Stelle noch keine Implementationsdetails vorgestellt werden, fasse ich die Konsequenzen dieses Grundsatzes für die Pausenplaner-Anwendung leicht vorgreifend, aber kurz zusammen:

- Zugriffe auf entfernte Objekte (und das sind immer Zugriffe auf das Pausenplaner-Archiv, das auf einem Server-Rechner läuft) werrden an möglichst wenigen Stellen im Programmtext gekapselt und dort mit entsprechendem Exception Handling geklammert.
- Das Archiv bietet einen Ereignismechanismus an, der es interessierten Objekte erlaubt, sich bei Ereignissen anzumelden. Die Benachrichtigung dieser (potentiell entfernten) Objekte wird ebenfalls durch Exception Handling geklammert.
- Die in einem entfernten Archiv abgelegten Materialien (Pausenplan, Lehrkörper) werden durch einen selbstimplementierten Mechanismus als lokale Kopien entnommen. Beim Zurückschreiben werden die neuen Versionen wiederum lokale Objekte des Rechners, auf dem das Archiv existiert.
- Die Verwendung von Name Servern erlaubt es, Anwendungen ohne Eingriffe in den Programmcode auf örtliche Verschiebungen von Objekten oder Änderungen der Netztopologie einzustellen.

5 Verteilung und verteilter Zugriff

Bei der Art von Anwendung, wie sie mit dem Pausenplaner realisiert werden soll, spielt das zugrundeliegende Modell für den verteilten Zugriff auf die Materialien eine entscheidende Rolle. Dazu möchte ich zunächst die Begriffe Original und Kopie diskutieren

Wenn man sich mit diesen Begriffen beschäftigt, wird schnell klar, daß es sich um ein kompliziertes Thema handelt. Versucht man einen an physischen Gegenständen orientierten, allgemeinen Original-Kopie-Begriff zu finden, erhält man leider wenig, das eine direkte Übertragbarkeit auf Software-Objekte zuließe: Ein Original ist in der physischen Welt von Schriftstücken, Dokumenten, Kunstwerken etc. ein einziges, durch die Art seiner Entstehung ausgezeichnetes Objekt. Eine Kopie von einem Original ist demgegenüber ein durch Nachbildung des Originals konstruiertes Ding. Was immer die Bedeutung der Objekte im Einzelnen sein mag, das Original wird stets als das maßgebliche und entscheidende Objekt angesehen. Meist kann man Original und Kopie anhand von Merkmalen unterscheiden, die mit der Vielfalt von Merkmalen physischer Gegenstände zusammenhängen.

Bei Software-Objekten hingegen können Kopien digitaler Originale den Originalen selbst ununterscheidbar gleichen; die Unterscheidung zwischen Original und Kopie kann dann nur in der Bedeutung liegen, die man ihnen*von aussen* zumisst, und in dem daraus resultierenden, verschiedenen Umgang mit ihnen).

Aus diesem Grund behandele ich diese Fragestellung, indem ich eine Original-Kopie-Begriff für die Pausenplaner-Anwendung definiere, der sich nur an den speziellen Gegebenheiten dieser Anwendung orientiert.

5.1 Original und Kopie beim Pausenplaner

Die Identität eines Pausenplanes / eines Lehrkörpers gegenüber dem Benutzer ist dadurch gegeben, daß er ihn unter einem bestimmten Namen aus dem Pausenplaner-Archiv entnehmen kann. Der Begriff von Kopie ist demnach der eines vom Zustand her identischen Objektes, das sich allein in seiner Benennung vom Original unterscheidet. Da eine Kopie bearbeitbar ist, kann sie auch wieder als Original angesehen werden, als ein eigenes Original mit einem eigenen Namen und einer eigenen Entwicklung entlang der Zeit.

Um so etwas wie exklusive Bearbeitung zu ermöglichen, müssen sich die formaleren Beschreibungen von Original und Kopie, weil sich der Originalbegriff sehr auf den Namen eines Pausenplanes / Lehrkörper im Pausenplaner-Archiv bezieht, stark daran orientieren, unter welchen Umständen Pausenpläne / Lehrkörper unter welchen Namen ins Pausenplaner Archiv gelegt werden können (mit Material ist im folgenden immer Pausenplaner oder Lehrkörper gemeint):

Original

Wenn ein Material zur Bearbeitung entliehen wird und nach der Bearbeitung wieder unter dem gleichen Namen (also unter Wahrung der fachlichen Identität) ins Pausenplaner-Archiv zurückgestellt werden kann, möchte ich es als Original bezeichnen. Zusätzlich möchte ich dem Anwender aber die Möglichkeit geben, die gemachtenÄnderungen zu verwerfen, das Original also unverändert zurückzustellen.

Gültige Version

Im Gegensatz zum physischen Archiv ist beim Archiv der Pausenplaner-Anwendung ein entliehenes Material nicht einfach weg, sondern es wird eine*gültige Version* vorgehalten, das heißt: Das

Pausenplaner-Archiv merkt sich, welchen Zustand das Material hatte, als es zuletzt zurückgestellt wurde, und dieser Zustand darf als die gültige Version angesehen werden.

Kopie

Wenn ein Material nicht mehr als Original zugreifbar ist, kann vom Pausenplaner-Archiv immer noch eine technische Kopie der gültigen Version des Materials angefordert werden. Dieses Material kann zwar verändert, aber nicht unter dem Namen des Originals ins Pausenplaner-Archiv zurückgestellt werden (das heißt, daß die fachliche Identität nicht die des Originals ist). Eine solche technische Kopie möchte ich auch fachlich als Kopie ansehen.

5.2 Modelle für den verteilten Zugriff

Modelle für den verteilten Zugriff beschreiben, wann und mit welchen Berechtigungen Benutzer Zugriff auf Materialien erhalten können, in Abhängigkeit von Aktionen der anderen Benutzer.

Einige denkbare Modelle:

a) **Beliebig viele Originale, regellos**
Jeder Benutzer kann zu jeder Zeit ein Original des Materials erhalten. Veränderungen können jederzeit übernommen werden. Für eine konkurrente Situation bedeutet das: Der letzte Benutzer, der zurückschreibt, „gewinnt".
Dieses Modell ist angebracht, wenn sehr geringe Sicherheitsanforderungen an den Umgang mit den Materialien gestellt werden (weil z.B. die jeweils gemachten Änderungen nur marginal sind) und überhaupt sehr selten konkurrente Situationen auftreten. Es handelt sich hierbei um nicht mehr, als ein mehreren Benutzern verteilt zugängliches Dateisystem realisiert. Wenn dann doch Problemsituationen auftreten, müssen diese außerhalb des Systems von den Benutzern gelöst werden.

b) **Beliebig viele Originale, („optimistic locking")**
Jeder Benutzer kann zu jeder Zeit ein Original des Materials erhalten. Jedoch können die Veränderungen nur übernommen werden, wenn das Original*nach* der letzten Veränderung am Material entliehen wurde. Das heißt also für eine konkurrente Situation: Der erste Benutzer, der zurückschreibt, „gewinnt".
Hier gilt das gleiche wie für das obige Modell, nur daß dieses hier besser geeignet ist für Fälle, in denen im Unglücksfall der „Verlierer" der konkurrenten Situation auch informiert werden muß, daß seine Änderungen zunächst nicht einfließen können. Dieses Modell ist analog zu dem, das auch viele Datenbanken anbieten dort häufig mit dem Begriff „optimistic locking" bezeichnet wird, (oder auch „Optimistisches Synchronisierungsverfahren", vgl. [Loc87])

c) **Ein Original, beliebig viele Kopien**
Es kann nur ein Original entliehen werden. Solange dieses entliehen ist, kann kein anderer Benutzer ein Original erhalten. Es können aber beliebig viele Kopien von anderen Benutzern entnommen werden.
Mit diesem Modell wird dem Benutzer, der ein Original erhält, eine exklusive Bearbeitung wie bei einem Einzelbenutzersystem garantiert. Es kann hier auch im ungünstigen Fall nicht zur Vernichtung von Arbeitsergebnissen durch chaotische konkurrente Zugriffe kommen. Gleichzeitig wird anderen Benutzern die Möglichkeit gegeben, Kopien zu erhalten. Dies ist

dann sinnvoll, wenn mit denen in einer Kopie enthaltenen Informationen noch etwas anzufangen ist, während das Original zur Bearbeitung entliehen ist.

d) **Nur ein Original, keine Kopien**
Es kann nur ein Original entliehen werden. Solange dieses entliehen ist, kann kein anderer Benutzer auf das Material zugreifen.
Dieses Modell ist dem vorigen sehr ähnlich; es ist dann angemessener, wenn, falls das Original entliehen ist, Arbeit auf Basis einer Kopie als problematisch angesehen wird.

Der Pausenplaner setzt Modell c) ("Ein Original, beliebig viele Kopien) um. Es bietet dem Benutzer eine Zusicherung (kein aufwendiges Zusammenführen konkurrent geänderter Materialien nötig), die allerdings nicht unabdingbar ist: auch das „optimistic locking" wäre möglich gewesen, aber c) ist eben komfortabler. Die gewählte Möglichkeit c) ist aber in jedem Fall d) ("ein Original, keine Kopien) vorzuziehen, da Kopien von Pausenplänen und Lehrkörpern, die sich gerade in Bearbeitung befinden, immer noch als aussagekräftig betrachtet werden dürfen (das entspricht dem Vorgehen, wie es auch in 3.2 dargestellt wird: Der Plan-Verwantwortliche nimmt in der Schilderung dort zwar den Zettelkasten mit, tatsächlich können aber, während er an dem Plan arbeitet, weitere Lehrer im Sekretariat anrufen und Ausschlußzeiten bekanntgeben u.ä.)

5.3 Zugriff auf Materialien

Bei der Formulierung eines Modells des verteilten Zugriffs auf Materialien für den Pausenplaner ist zu beachten, daß das Zusammenspiel der zentralen Materialien Pausenplan und Lehrkörper den Entwurf aus 5.2 kompliziert: Wenn jemand nämlich einen Pausenplan bearbeitet, so muß er immer zumindest „lesend" auf den zugehörigen Lehrkörper zugreifen, mithin wenigstens eine Kopie des Lehrkörpers entleihen.

Dies ist die angemessene und konkretisierte Anpassung des gewählten Modells auf die komplexere Situation:

- Pausenpläne und Lehrkörper werden über eindeutige Namen identifiziert
- eine Kopie hat initial den Namen „Kopie von <Name des Originals>" Dieser Name kann beim Einfügen ins Archiv natürlich noch verändert werden. Die Kopie kann nach Ablage ins Archiv selbst wieder als ein neues, anderes Original betrachtet werden.
- die Benutzer haben Zugriff auf die im Archiv abgelegten Pausenpläne und Lehrkörper. Dabei können sie grundsätzlich wählen, ob sie ein Original oder eine Kopie erhalten möchten.
- tatsächlich erhalten kann ein Benutzer ein Original aber nur dann, wenn im Moment kein anderer Benutzer ein solches besitzt.
- wenn ein Benutzer einen Pausenplan entleiht, so kann er auch das Original des zugehörigen Lehrkörpers für sich reservieren (ohne, daß ein Lehrkörper-Werkzeug geöffnet sein müßte). Es ist konfigurierbar, ob eine solche Reservierung bei jedem Ausleihen eines Pausenplanes versucht wird, oder ob nachgefragt wird, oder ob in jedem Fall nicht reserviert wird.
- wenn ein Benutzer die Kopie eines Pausenplanes entleiht, wird er gefragt, ob sich dieser auf denselben Lehrkörper beziehen soll wie das Original oder auf eine Kopie desjenigen Lehrkörpers. Im zweiten Fall muß der Benutzer sofort einen Namen für diese Lehrkörper-Kopie festsetzen, sie wird dann erzeugt und im Archiv abgelegt.
- wenn ein Benutzer eine Kopie bearbeitet, wird er informiert, wenn sich das Original verändert hat (das heißt: wenn das Original nach Bearbeitung ins Archiv zurückgestellt wurde).

- wenn ein Benutzer einen Pausenplan bearbeitet (ohne den zugehörigen Lehrkörper reserviert zu haben), und währenddessen der Lehrkörper geändert wird, dann wird der Benutzer über die aufgetretenen Veränderungen informiert und bei den Anpassungen an diese Veränderungen unterstützt (s. hierzu 7)

6 Implementation

6.1 Ereignismechanismus

Wenn von Materialien die Rede ist und davon, daß Funktionskomponenten diese modifizieren, dann ergibt sich sofort die Frage, zumal bei interaktiven Anwendungen, auf welchem Wege der aktualisierte Zustand des Materials bekanntgemacht und schließlich nach außen repräsentiert wird. Eine wichtige Rolle hierbei spielt der Ereignismechanismus, der beim WAM-Ansatz in Anlehnung an das Observer-Pattern (vgl. [Gam95]) implementiert wird.

Der Ereignismechanismus dient dazu, das zu beobachtende Objekt und das Beobachter-Objekt, im Fall von WAM meist Funktions- und Interaktionskomponente, lose aneinander zu koppeln. Zu diesem Zweck bietet die Funktionskomponente für die verschiedenen interessierenden Zustandsänderungen jeweils ein Ereignisobjekt, ein Exemplar der Klasse `Event` an. Bei diesem können Exemplare der Klasse `EventStub` registriert werden, die konfiguriert sind, bei Benachrichtigung eine bestimmte Methode an einer IAK zu rufen. Daszu beobachtende Objekt erfährt also nicht, wer sich für die Benachrichtigungen über Zustandsänderungen interessiert.

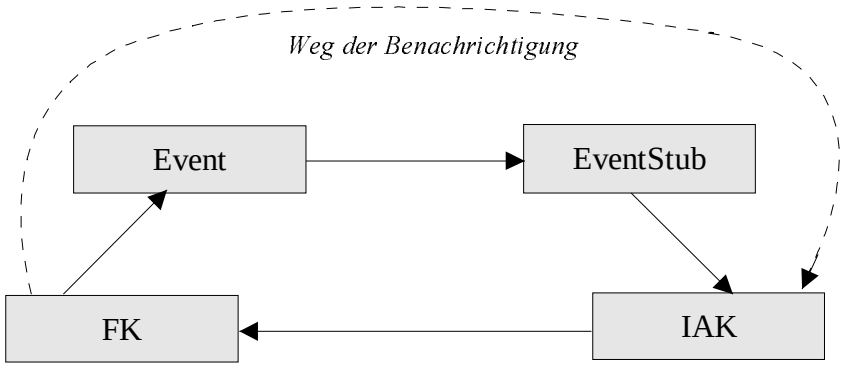

Abbildung 7 Ereignismechanismus (Benutzt-Beziehung)

Der von mir implementierte Mechanismus entspricht weitgehend dem, der ausführlich in [Gat96] beschrieben ist, weswegen ich mich hier kurz fasse. Es gibt allerdings einen gravierenden Unterschied: Der von mir implementierte Ereignismechanismus versendet Informationen über die gemachten Änderungen mit der Ereignisnachricht, eine Variante, die in [Gam95] als *Push-Modell* bezeichnet wird und im Zusammnhang mit asynchronen Ereignisnachrichten unverzichtbar ist.

6.2 Überlegungen zur Asynchronizität

Synchron/Asynchron

Wenn in einem System nebenläufige Prozesse / Prozeßfäden (Threads) auftreten, ist man bemüht, deren Ausführung zu synchronisieren. Damit ist gemeint, daß man bei der zeitlich überlappenden Manipulation derselben Objekte durch mehrere Threads / Prozesse sicherstellt, daß keine inkonsistenten Zustände entstehen können. Dies geschieht meist durch die Verwendung von

Semaphoren (oder ähnlicher Konstrukte wie „critical sections“, die letztlich aber Semaphoren verwenden).

Synchrone / Asynchrone Kommunikation

Die folgenden Betrachtungen lehnen sich wiederum an [Zül98] an.

Auf dem Paradigma der Objektorientierung aufbauend möchte ich den Begriff Kommunikation im folgenden als das Senden von Nachrichten zwischen Objekten verstanden wissen.

Man kann diese Kommunikation dabei in drei Kategorien unterteilen:

- Ein Objekt sendet eine Nachricht an ein anderes Objekt und erwartet einen Resultatwert zurück
- Ein Objekt sendet eine Nachricht an ein anderes Objekt ohne Erwartung eines Resultatwertes
- Ein Objekt kommuniziert mit einem anderen Objekt über einen Ereignismechanismus, also über eine weitere Indirektion (das geschieht natürlich durch mehrere Kommunikationsaktionen der vorigen beiden Kategorien)

Man spricht von *synchroner Kommunikation*, wenn der Kontrollfluß erst dann wieder zu einem Objekt, welches eine Nachricht versendet hat, zurückkehrt, wenn das empfangende Objekt die Verarbeitung der Nachricht bestätigt hat.

Von *asynchroner Kommunikation* spricht man, wenn ein Objekt, das eine Nachricht an ein anderes Objekt versendet hat, sogleich mit der Ausführung fortfährt, ohne abzuwarten, bis das Empfängerobjekt die Nachricht verarbeitet hat.

Wenn Kommunikation asynchron stattfindet, hat das auf die oben definierten drei Kategorien sehr unterschiedliche Auswirkungen:

- Im ersten Fall, wenn das Objekt, welches die Nachricht versendet hat, einen Resultatwert vom empfangenden Objekt erwartet, muß spätestens, wenn dieses Resultat weiterverarbeitet werden soll, eine Synchronisation stattfinden: Es muß gegebenenfalls gewartet werden, bis das Resultat zur Verfügung steht. Wenn das Resultat nach einer festzulegenden Zeit nicht zur Verfügung gestellt worden ist, muß eine Fehlerbehandlung stattfinden.
- Wenn das sendende Objekt jedoch keinen Rückgabewert erwartet, ist die Situation etwas unkomplizierter. Dann nämlich entspricht der Kommunikationsvorgang „nur“ dem Starten eines neuen Threads (im Einprozeßraum) bzw. dem Starten eines neuen Threads auf einem entfernten Rechner (verteilt).
- Bei dem Ereignismechanismus findet durch Einführung von Asynchronizität eine semantische Verschiebung statt. Ereignisse informieren über Statusänderungen, dabei entkoppeln sie den Auslöser einer Statusänderung von den Objekten, die darüber informiert werden sollen. Bei synchronen Ereignissen (s.6.1) aber kann das benachrichtigte Objekt sicher sein, das die mit dem Ereignis transportierte Information während seiner Reaktion auf dieses Ereignis gültig ist und bleibt, wenn es nicht selber etwas daran ändert. Ein asynchron ausgelöstes Ereignis hingegen teilt dem benachrichtigten Objekt nur mit, daß zu einem Zeitpunkt in der jüngsten Vergangenheit eine bestimmte Information gültig war. In vielen Fällen macht das zwar keinen großen Unterschied (Beispiel: Ein Ereignis teilt mit, daß ein Fensterobjekt der Benutzeroberfläche zerstört wurde), in anderen aber ist die Bedeutung der Nachricht dadurch stark verändert (Beispiel: Ein Ereignis teilt mit, daß eine bisher blockierte Ressource frei geworden ist).

•

Vor- und Nachteile

Der zentrale Vorteil der synchronen Kommunikation gegenüber der asynchronen liegt darin, daß der Ablauf streng sequentiell ist, was eine größere Übersichtlichkeit bedeutet und den Entwurf sehr erleichtert. So können in einem System, das nur mit synchroner Kommunikation arbeitet, weitreichende Aussagen über den Ist-Zustand des Systems gemacht werden, eben weil ein eine Nachricht erhaltendes Objekt davon ausgehen kann, daß das Restsystem stillsteht, solange der Kontrollfluß bei ihm verweilt. Kommt dagegen asynchrone Kommunikation vor, können im übrigen System Veränderungen vor sich gehen, so daß ein eine Nachricht empfangendes Objekt daraus nur eine Information über den Ist-Zustand einer nahen Vergangenheit ableiten kann.

Gerade bei einer verteilten Anwendung hat aber synchrone Kommunikation entscheidende Nachteile:

- Es ist denkbar, daß ein synchroner, entfernter Methodenaufruf das rufende Objekte*unbegrenzt warten läßt*, auch ohne daß ein Fehler beim entfernten Aufruf selbst vorgefallen ist: Kommt es nach erfolgreichem Aufruf nämlich auf dem entfernten Rechner zu einem Blockieren oder Abstürzen des von außen gestarteten Threads, wird die Verarbeitung der Nachricht nie bestätigt werden. Ein asynchroner Aufruf hingegen, der die Verteilung deutlich werden läßt, könnte auf eine Zeitüberschreitung angemessen reagieren (das ist im übrigen natürlich auch sinnvoll für Fälle, in denen der entfernte Rechner momentan einfach nur überlastet ist). Dies muß deutlich unterschieden werden von einem Timeout auf der Ebene der Netzwerkkommunikation, denn dieser würde nur aktiv, wenn der *entfernte Aufruf selbst* mißlingt.
- Bei der Pausenplaner-Anwendung, bei der bei einem Server (auf dem das Pausenplaner-Archiv läuft) Methodenaufrufe von verschiedenen Client-Rechnern aus stattfinden, ist es im Sinne der gleichmäßigen Verteilung von Antwortzeiten sinnvoll, die Anfragen der Clients an den Server, wenn möglich, nicht sequentiell, sondern parallel abarbeiten zu lassen. Damit hat man aber schon asynchrone Kommunikation, nämlich mehrere Prozeßfäden zumindest Server-seitig. Vom Server aus lassen sich damit Ereignisse nur noch dann wirklich synchron (hiermit ist gemeint: mit einer Zusicherung der Gültigkeit einer Aussage über den Zustand der Server-seitigen Objekte) auslösen, wenn man mit einigem Aufwand zusätzliche Synchronisationsvorkehrungen trifft. Mithin ist hier rein synchrone Kommunikation schlicht ungeeignet.

•

Asynchrone Kommunikation bietet als Vorteil, daß mehrere Prozeßfäden gleichzeitig ablaufen können und so die Kommunikation nicht den weiteren Programmverlauf blockiert. Wie schon dargestellt wurde, ist dies für das Versenden von Nachrichten an entfernte Objekte besonders wichtig. Je nach fachlichem Kontext kann ein Anwender unter Umständen schon weiterarbeiten, während seine vorigen Aktionen noch Vorgänge auf einem anderen Rechner bewirken.

Die Nachteile der asynchronen Kommunikation:

- der Erhalt von Resultatwerten durch asynchrone Methodenaufrufe muß durch Synchronisation sichergestellt werden (ggf. Fehlerbehandlung).
- Die Semantik von Ereignissen verkompliziert sich erheblich
- Alle Objekte, die potentiell von mehreren Threads gleichzeitig verwendet werden, müssen Synchronisationsmechanismen besitzen, um das Auftreten von dadurch verursachbaren Inkonsistenzen zu verhindern. Kurz: Sie müssen thread-safe sein.

•

Konsequenz

Die vorhergehenden Feststellungen führen mich zu folgender Ansicht: Man benötigt asynchrone Kommunikation, aus technischen und aus Performanzgründen. Andererseits ist der auf synchroner

Kommunikation basierende Entwurf übersichtlicher und weniger fehlerträchtig.

Die Folgerung hieraus sollte sein, in einer Anwendung möglichst Bereiche zu bilden, innerhalb derer die Kommunikation rein synchron stattfindet, die untereinander aber möglicherweise asynchron kommunizieren.

Ich werde diese Strategie nun anhand des Pausenplaners verdeutlichen.

Umsetzung beim Pausenplaner

Obgleich man für das Versenden von Ereignisnachrichten über Prozeß- bzw. Rechnergrenzen andere Mechanismen als den in6.1 vorgestellten vorschlagen könnte (die geeigneter sein könnten, den Netzwerkverkehr gering zu halten), wird dieser von mir mit geringer Anpassung (nämlich Ermöglichung asynchron versendeter Ereignisnachrichten) eingesetzt, da die Zahl der über Rechnergrenzen hinweg kommunizierenden Objekte recht gering ist, genauer:

Wie in Abschnitt6.3 ("Der verteilte Zugriff: Realisierung in drei Schichten") dargelegt, versendet nur ein einziges Objekt (nämlich das Archiv) aus einem jeden Client Object Space Nachrichten an die im Server Object Space existierende Materialverwaltung (s. auch hier 6.3), auch der Empfang von deren Ereignisnachrichten wird nur über dieses einzige Objekt abgewickelt. Deshalb kann dieses für Synchronisation und Multiplexing (also für eine Reduzierung der Netzbelastung) sorgen.

Die Serverkomponente der Applikation, das ist im Wesentlichen ein Materialverwaltung genanntes Objekt, welches vom Pausenplaner-Archiv verwendet wird , bearbeitet die von entfernten Objekten gesendeten Nachrichten asynchron. Die Benachrichtigungen durch die von ihm angebotenen Ereignisse (die die entsprechenden Ereignisobjekte des Pausenplaner-Archivs auslösen) wiederum treten für die Clients asynchron auf. Sie sind die einzigen Quellen asynchroner Kommunikation für die Clients. Dies verdeutlicht die folgende Grafik:

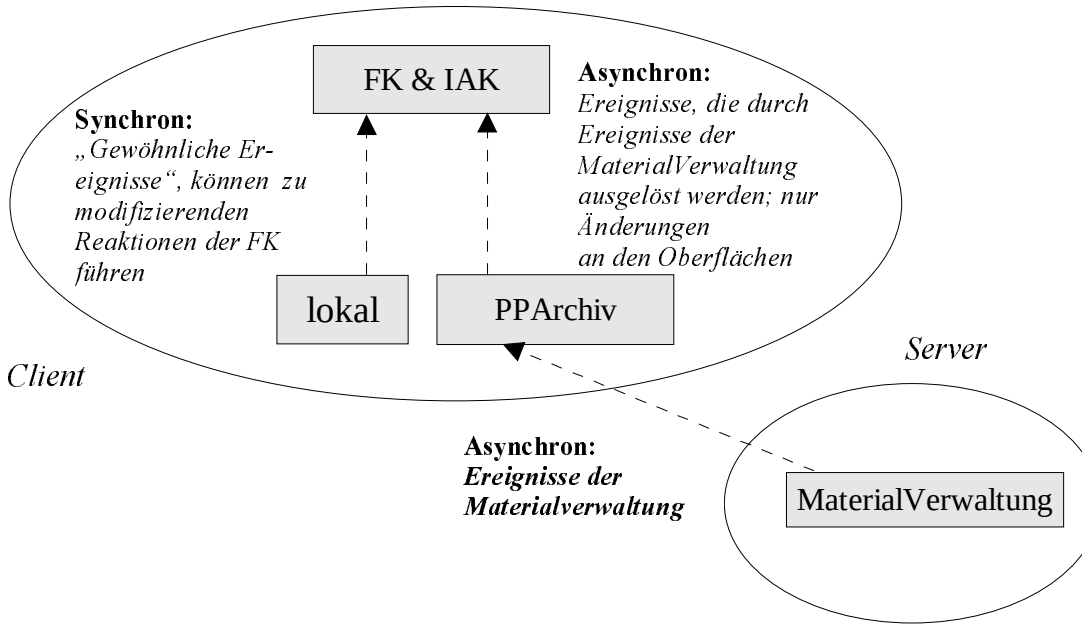

Abbildung 8 Trennung asynchroner und synchroner Benachrichtigungen durch Konvention (Schema)

Hierbei ist allerdings zu beachten: Die Ereignisse, die ausgehend von der Materialverwaltung in den

Client wirken, führen allesamt nur zu veränderten Anzeigen an den Werkzeugoberflächen (wenn zum Beispiel ein entliehener Lehrkörper wieder freigegeben wurde), nicht zu automatischen Reaktionen, die ein in Bearbeitung befindliches Material manipulieren würden. Eine entsprechende Reaktion auf z.B. die Veränderung des Leihstatus eine Lehrkörpers wird stets nur vom Benutzer ausgehen oder zumindest von ihm erfragt.

Daraus folgt aber, daß die FKs auf Client-Seite nicht asynchron zum Verändern der von ihnen manipulierten Datenstrukturen verwendet werden und also auf Basis des Modells synchroner Kommunikation entworfen werden können. Damit sind für einen beträchtlichen Teil der Applikation die Vorteile synchroner Kommunikation nutzbar.

Implementation asynchroner Ereignisse

Die grundsätzliche Ereignis-Architektur aus6.1 wurde derart erweitert, daß die Klasse `Event` eine abstrakte Basisklasse für die Unterklassen`EventSync` und `EventASync` wird. Dabei realisiert `EventSync` die bekannte Variante, in der ein Stub nach dem anderen aufgerufen wird, während bei EventASync die verschiedenen Stubs mittels eines`fork` nebenläufig getriggert werden und der Kontrollfluß aus der Methode `teileEreignisMit:` der Klasse `Event`, welche die Benachrichtigungen an die Stubs verschickt, schon zurückkehrt, während das Ereignis noch von den Empfängern bearbeitet wird.

Die Bindung der Entscheidung über synchrone oder asynchrone Ereignismitteilung an das Ereignis selbst erscheint natürlich: Es liegt in der Eigenart des betrachteten Ereignisses und weniger in der der Empfänger (dies hätte eine Bindung an die Stubs nahegelegt), welcher der beiden Modi vorzuziehen ist.

Darüber hinaus gewährleistet diese Lösung eine etwas einfachere Änderbarkeit der getroffenen Entscheidung zwischen synchron und asynchron für jedes einzelne Ereignis (die Verwendung eines synchronen Ereignis macht natürlich auch nur dann Sinn, wenn nicht ein anderer Thread die mit der Ereignisnachricht verbundene Information noch während der Versendung ungültig werden lassen kann). Dabei ist allerdings unbedingt die veränderte Semantik zu betrachten, so daß häufig auch auf Seiten des Ereignisempfängers Änderungen nötig werden.

So kann es bei asynchronen Ereignissen vorkommen, daß

- sie sich gegenseitig überholen und in falscher Reihenfolge beim Empfänger eintreffen.
- das Ausleihen eines laut Ereignis freigegebenen Materials fehlschlägt, weil es inzwischen erneut von anderer Seite ausgeliehen wurde.
- usw.

Diese Bedeutungsveränderung muss durch den Empfänger berücksichtigt werden, wenn eine Ereignisnachricht nicht mehr synchron, sondern asynchron versendet wird.

6.3 Der verteilte Zugriff: Realisierung in drei Schichten

Nun komme ich zur Implementation desjenigen Bereiches, der das Kernstück dieser Arbeit ist: des verteilten Zugriffs. Die Anforderungen hierzu habe ich in Abschnitt5 beschrieben.
Ich habe mich entschieden, diese Anforderungen mit einem dreischichtigen Modell zu realisieren.

Zum einen ist die Unterteilung in Client- und Server-Komponente naheliegend, um einen einfachen verteilten Zugriff auf das Pausenplaner-Archiv zu ermöglichen. Zum anderen benötigt der Archiv-Mechanismus zur Koordinierung des verteilten Zugriffs selbst eine einfache Persistenz- und Ereignisfunktionalität, die aber nicht in der Schnittstelle sichtbar sein sollte, auf der die

Funktionskomponenten der Anwendung aufsetzen. Es ist darum günstig, die Realisierung speziellerer Dienste auf Basis einfacherer durch Schichtenbildung umzusetzen.

Ich werde die drei Schichten und ihr Zusammenspiel im folgenden genauer (aber nicht vollständig, sondern in exemplarischen Ausschnitten) vorstellen:

Oberste Schicht: Das Archiv für den Pausenplaner

Das Archiv für den Pausenplaner ist diejenige Schicht, deren Schnittstelle es ermöglicht, die in Abschnitt beschriebene Funktionalität für den verteilten Zugriff auf die Pausenpläne und Lehrkörper einfachst zu realisieren.

Das Pausenplaner-Archiv setzt auf einem allgemeineren Archiv auf und liefert im Wesentlichen auf Basis von dessen Archivschnittstelle eine speziell auf die Pausenplaner-Materialien abgestimmte Schnittstelle. Zudem hält das Pausenplaner-Archiv die Information vor, welche Pausenpläne sich auf welche Lehrkörper beziehen.

Mittlere Schicht: Das Archiv

Das Archiv realisiert das Konzept von Original und Kopie, das in Abschnitt 5.2 unter c) aufgeführt ist (“ein Original, beliebig viele Kopien“) unter Verwendung einer Server-seitig laufenden Materialverwaltung. Dabei weiß das Archiv nichts von Pausenplänen oder Lehrkörpern und ihren Abhängigkeiten.

Außerdem implementiert das Archiv das „Lokalitätsprinzip“: Die entliehenen Originale/Kopien werden dem Entleiher als lokale Objekte in seinem Image zur Verfügung gestellt.

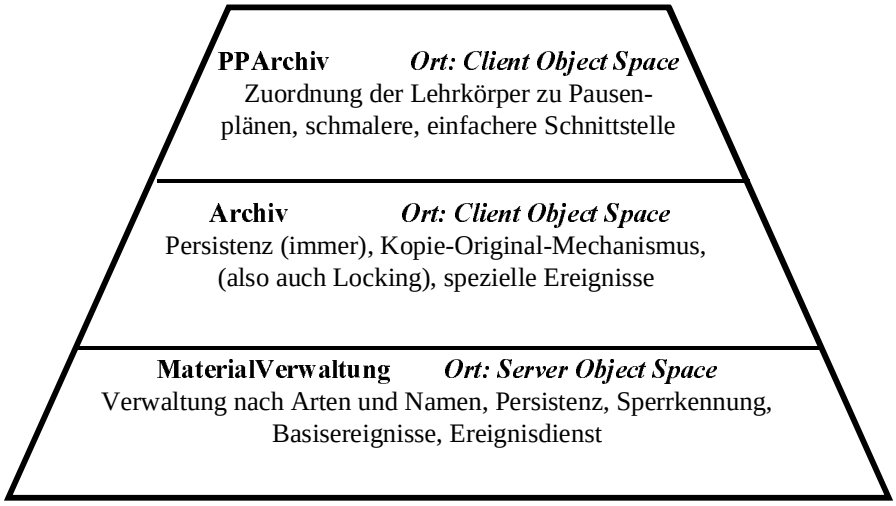

Abbildung 9 Die drei Schichten und die von ihnen bereitgestellten Dienste

Unterste Schicht: Die Materialverwaltung

Die Materialverwaltung stellt die Basisdienste für die Aufbewahrung von Materialien zur Verfügung, die Ablage unter einem bestimmten Namen, den Zugriff auf diese Materialien sowie Persistenz.

Örtliche Verteilung

Die folgende Abbildung verdeutlicht, daß die oben aufgeführten Schichten Pausenplaner-Archiv, Archiv und Materialverwaltung sich auf unterschiedliche Rechner/Prozeßräume verteilen. Dabei läuft die Materialverwaltung auf einem als Server ausgezeichneten Rechner. Das Archiv, welches die Materialverwaltung benutzt und ja auch die Bereitstellung lokaler technischer Kopien leistet, läuft bereits auf dem Client-Rechner, genauso wie das Pausenplaner-Archiv.

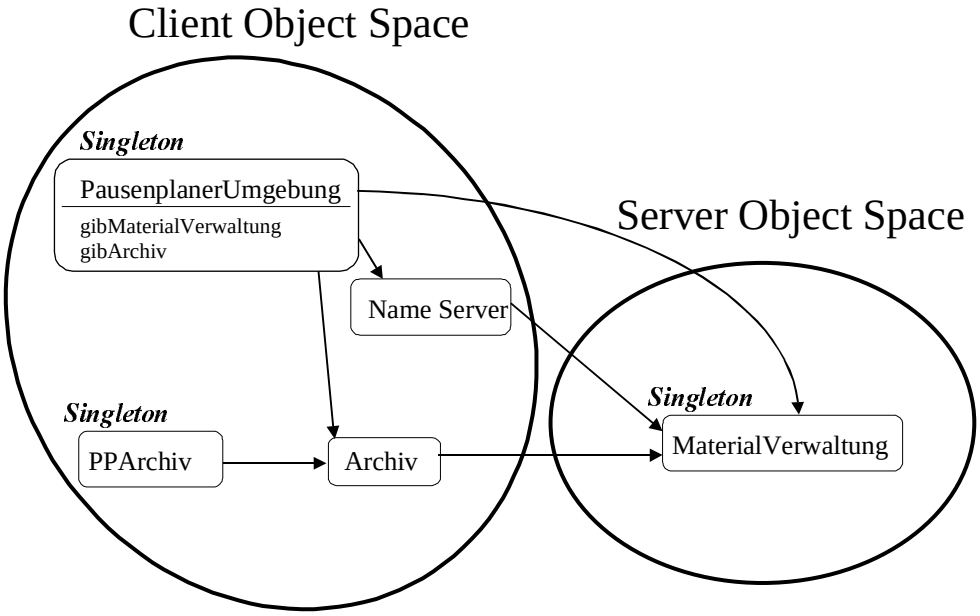

Abbildung 10 Topologische Verteilung der drei Schichten des Pausenplaner-Archivmechanismus

Genauer besehen

Es folgt eine genauere Darstellung der einzelnen Schichten und ihrer Schnittstellen. Dabei habe ich mich bemüht, das Pausenplaner-Archiv nicht allzu technisch zu beschreiben, da es den Umgang des Benutzers mit der Archivkomponente modelliert und hier vornehmlich auch so gesehen werden soll. *Das führt unter anderem dazu, dass eine deutlich technische Information wie „das zu speichernde Material muss den Aspekt <Speicherbar> implementieren", die für alle drei Schichten Gültigkeit besitzt, erst ab der zweiten vermerkt ist.*

6.3.1 Das Archiv für den Pausenplaner

Das Archiv für den Pausenplaner existiert genau einmal lokal in jedem Image und benutzt das Archiv, das es von der Pausenplaner-Umgebung erhalten kann, um seine Aufgaben zu erfüllen.

Aufgaben des Pausenplaner-Archivs:

- ermöglicht es, Pausenpläne und Lehrkörper persistent zu machen. Dabei werden sie unter dem Namen abgelegt, der ihrem Namensattribut entspricht.
- gibt nach dem schon beschriebenen Modell Original oder Kopie eines Pausenplanes oder eines Lehrkörpers heraus
- stellt Listen der Namen aller abgelegten Pausenpläne und Lehrkörper zur Verfügung
- gibt Auskunft, ob ein bestimmter Pausenplan oder Lehrkörper entliehen ist
- bietet Ereignisobjekte an, die über Ausleihen, Rückgabe, Neueinfügung oder Löschung eines Pausenplanes oder Lehrkörpers informieren. An diesen können sich interessierte Objekte anmelden.
- Das Pausenplaner-Archiv kann darüber Auskunft geben, auf welchen Lehrkörper sich ein bestimmter Pausenplan bezieht und welche Pausenpläne sich auf einen bestimmten Lehrkörper beziehen - ohne, daß die Materialien als Original oder Kopie entnommen werden müßten.

•

Die Sondier-Problematik

Im Zusammenhang mit dem Entleihen von Materialien nach dem Konzept aus 5.1 stellt sich die Frage, wie dies geschehen soll. Das Problem ist, daß das Sondieren, ob das Original eines Materials verfügbar ist, auch bei positiver Beantwortung keine Gewähr bietet, daß bei einem darauffolgenden Entleihversuch diese Verfügbarkeit noch immer gegeben ist.

Ich habe folgendes Vorgehen festgelegt: Bevor versucht wird, ein Original zu entleihen, wird zunächst sichergestellt, daß es bisher als nicht entliehen gilt. Wenn dies zutrifft und der Entleihversuch dennoch fehlschlägt, kann nochmals sondiert werden, ob das Original inzwischen entliehen ist. Wenn nein, wird der Versuch wiederholt (eine Maximalzahl von konsekutiven Versuchen ist festgelegt), sonst mit Fehlermeldung abgebrochen.

Dieses Verhalten ist auch beim Archiv (nächster Abschnitt) zu finden.

Schnittstelle des Pausenplaner-Archivs (Ausschnitt):

(Anmelden neuer Materialien)

```
meldeLehrKoerperAn:einLehrKoerper fuer:einBenutzer
meldePausenPlanAn:einPausenPlan fuer:einBenutzer
```

Mit diesen Methoden können Lehrkörper und Pausenpläne im Pausenplaner-Archiv angemeldet werden. Der Lehrkörper/Pausenplan gilt hernach als vom Entleiher (der als zweiter Parameter übergeben wird) ausgeliehen, muss also immer noch „zurückgegeben“ werden.

Im Vorgriff auf die weiteren Schichten (Archiv und Materialverwaltung) möchte ich hier einmal anhand eines Sequenzdiagramms darstellen, wie das Zusammenwirken aussieht. Dabei ist zu beachten, dass das Archiv sich von der Materialverwaltung einen Kopierer geben läßt, um im Prozess des Materialverwaltung eine lokale Kopie des Lehrkörpers zu erstellen:

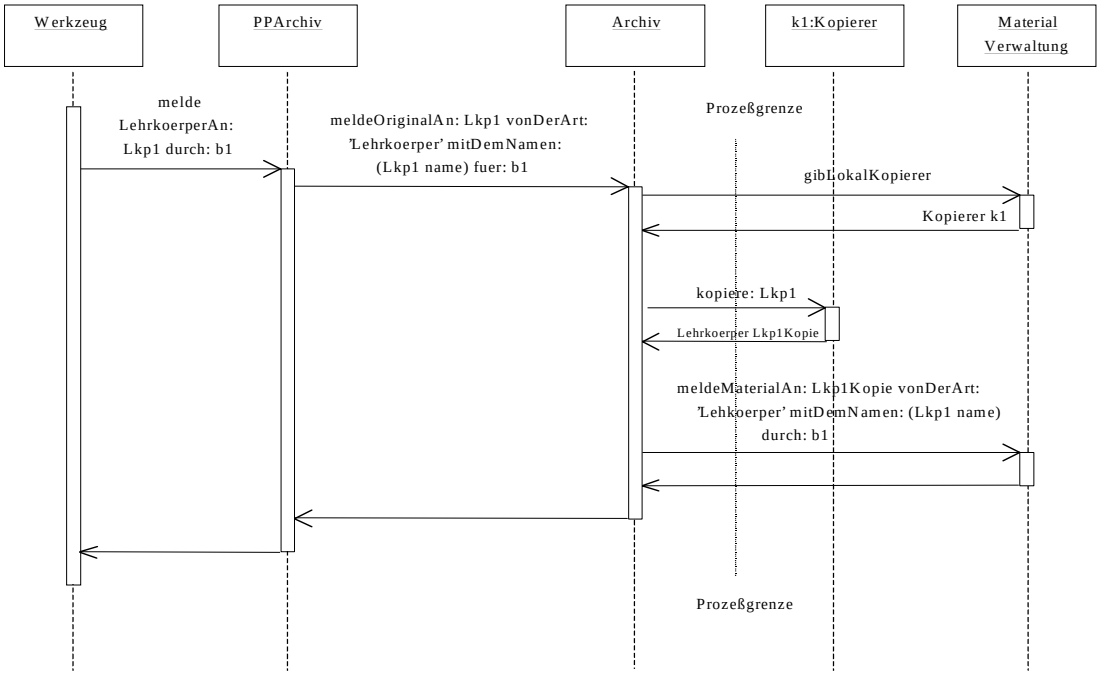

Abbildung 11 Zusammenspiel der Schichten bei Anmeldung eines Lehrkörpers

In dieser Weise läuft das Zusammenspiel von Pausenplaner-Archiv, Archiv und Materialverwaltung auch bei fast allen anderen Vorgängen ab.

(Informationen über den Entleihstatus)

```
istLehrKoerperEntliehenMitDemNamen:einString
entleiherVonLehrKoerperMitDemNamen:einString
```

Die Methodennamen dürften selbsterklärend sein.

(Informationen über Inhalte des Pausenplaner-Archivs)

```
beiPausenPlanEingefuegtTrageStubEin:einEventStub
beiPausenPlanEingefuegtTrageStubAus:einEventStub
gibPausenPlanNamen
```

Die erste Methode trägt einen als Parameter übergebenen EventStub, also einen Ereignisempfänger, für das Ereignis „Pausenplan eingefügt“ ein, der dann informiert wird, wann immer ein neuer Pausenplan eingefügt wird. Die zweite Methode ermöglicht es, einen solchen Stub wieder auszutragen Die dritte Methode liefert eine Liste mit den Namen aller im Archiv befindlichen Pausenpläne.
Es gibt weitere solche Ereignisse, und entsprechende Methoden (auch für Lehrkörper).

(Materialien entleihen, zurückgeben etc.)

```
entleiheLehrkoerperMitDemNamen:einString fuer:einBenutzer
gibLehrKoerperUnveraendertZurueck:einLehrKoerper
          mitDemNamen:einString durch: einBenutzer
trageVeraenderungenEinVonLehrkoerper:einLehrKoerper
          mitDemNamen:einName durch:einBenutzer
gibLehrKoerperZurueck:einLehrKoerper mitDemNamen:einString
          durch:einBenutzer
```

Die erste hier aufgeführte Methode liefert den dem übergebenen Namen zugehörigen Lehrkörper zurück, wenn dieser bisher nicht ausgeliehen ist (sonst nil). Der zweite Parameter hierbei ist der einzutragende Entleiher.
Die zweite Methode ändert den Status einees Lehrkörpers auf „nicht entliehen“, ohne aber Veränderungen an dem Lehrkörper einzutragen. Die dritte Methode hingegen trägt Veränderungen an einem Lehrkörper im Archiv ein, der Lehrkörper aber bleibt entliehen. Die dritte aufgeführte Methode benutzt die beiden vorigen, um erst die Veränderungen einzutragen und den Lehrkörper dann freizugeben
Entsprechende Methoden existieren auch für Pausenpläne.

(Relation zwischen Pausenplänen und Lehrkörper)

```
gibLehrKoerperNamenZuPausenPlanMitDemNamen: einString
gibPausenPlanNamenBezogenAufLehrKoerperMitDemNamen: einString
```

Die erste Methode liefert den Namen des Lehrkörpers, der zum Pausenplan mit dem übergebenen Namen gehört.

Dagegen gibt die zweite Methode eine Liste der Namen jener Pausenpläne zurück, die sich auf den Lehrkörper mit dem angegebenen Namen beziehen.

6.3.2 Das Archiv

Das Archiv existiert genau einmal lokal in jedem Image und verwendet die im Server-Prozeß existierende Materialverwaltung

Aufgaben des Archivs:

- Das Archiv ist im Vergleich zum Pausenplaner-Archiv allgemeiner gehalten: Es ermöglicht die Ablage von Materialien getrennt nach Arten. Dieser Mechanismus wird vom Pausenplaner-Archiv genutzt, um unter den Arten „LehrKoerper“ und „PausenPlan“ Materialien abzulegen
- Alle in das Archiv eingetragenen Materialien werden persistent gemacht
- Das Archiv implementiert das Lokalitätsprinzip. Das bedeutet, daß über das Archiv erhaltene Materialien tatsächlich in das lokale Image kopiert werden (hier sollte keine Begriffsverwirrung auftreten: auch eine solche lokale Kopie kann im Sinne von 5.1 ein Original sein).
- Das Archiv implementiert das Konzept „ein Original, beliebig viele Kopien“ multi-threading fähig
- Ereignisse informieren über das Entleihen und Zurückgeben von Originalen

Anzumerken ist, daß sich ein Objekt wie das Archiv in Bezug auf den WAM-Ansatz in einer Grauzone zwischen Automat und Behälter bewegt. Man könnte es auch als einen komplexen Behälter realisieren, wobei der Zugriff auf die Behälterinhalte über einen Automaten erfolgte. Bei einer solchen Konstruktion wäre der Automat für das Versenden von Ereignisnachrichten zuständig. Die hier gewählte Implementierung aber betrachtet das Archiv als ein Objekt, da sich kein einleuchtendes Konzept für eine Aufteilung, insbesondere für die Gestaltung des Behälters, fand.

Schnittstelle des Archivs (Ausschnitt):

(Anmelden von Materialien)

```
meldeOriginalAn:einMaterial vonDerArt:einString
        mitDemNamen:einString fuer:einBenutzer
```

Diese Methode meldet ein Original beim Archiv an. Das Objekt muß den Aspekt *Lokalkopierbar* implementieren (dies ist notwendig, um es zum Server transportieren zu können). Der Kopiermechanismus entspricht dem *Serializer/Atomizer*-Muster, s. [Rie97b])

(Erlangung von Original/Kopie, Veränderung und Rückgabe)

```
entleiheOriginalVonDerArt:einString mitDemNamen:einString
        fuer:einBenutzer
entferneOriginalVonDerArt:einString mitDemNamen:einString
```

Mit diesen Methoden kann man ein Original entleihen bzw. aus dem Archiv entfernen. In beiden Fällen wird das Material als Rückgabewert geliefert. Wenn das Material entliehen ist, wird nil zurückgegeben (s. die Sondier-Problematik). Ein Entfernen ist dann auch nicht möglich.

```
entnehmeKopieVonDerArt:einString mitDemNamen:einString
gibOriginalUnveraendertZurueck:einMaterial
        vonDerArt:einString mitDemNamen:einString
        durch:einBenutzer


trageVeraenderungenEinVonOriginal:einMaterial
        vonDerArt:einString mitDemNamen:einString
        durch:einBenutzer
gibOriginalZurueck:einMaterial vonDerArt:einString
        mitDemNamen:einString durch:einBenutzer
```

Durch den Aufruf der ersten Methode erhält man eine Kopie eines Materials. Wenn es ein Material dieses Namens gibt, sollte dies immer gelingen.
Die weiteren Methoden verhalten sich wie ihre Gegenstücke im Pausenplaner-Archiv

(Sondierung)

```
entleiherVonOriginalVonDerArt:einString mitDemNamen:einString
istOriginalEntliehenVonDerArt:einString mitDemNamen:einString
```

Diese Methoden geben Auskunft über den Entleihstatus eines Materials

(Ereignisse)

```
beiMaterialEingefuegtTrageStubEin:einEventStub
beiMaterialEingefuegtVonDerArt:einString trageStubAus:einEventStub
beiMaterialEntferntVonDerArt:einString mitDemNamen:einString
         trageStubEin:einEventStub
```

Diese Methoden erlauben das Eintragen von Stubs bei Ereignissen, die sich allgemein auf Materialien beziehen oder auf bestimmte Materialarten oder auf ganz bestimmte einzelne Materialien.

6.3.3 Die Materialverwaltung

Die Materialverwaltung existiert genau einmal, auf einem als Server ausgezeichneten Rechner in einem eigenen Prozess. Für ein solches Objekt steht kein passender WAM-Begriff zur Verfügung. Vielleicht kommt der Begriff eines fachlichen Dienstes oder Services (in Anlehnung an Betriebssystemdienste), dem Charakter der Materialverwaltung am nächsten.

Aufgaben der Materialverwaltung

Der Materialverwaltung erfüllt folgende Aufgaben:

- Materialien können, geordnet nach „Arten“, in die Materialsammlung eingefügt, entnommen und wieder aus ihr entfernt werden. Es gibt keine Unterscheidung im Sinne von Original und Kopie.
- Materialien können persistent gemacht werden
- Für jedes einzelne Material kann thread-safe eine „Sperrkennung“ (ein Integer und ein String) gesetzt/umgesetzt werden. Verwender dieses Mechanismus können eine Semantik festlegen und hierauf einen höherwertigen Locking-/Transaktionsservice aufsetzen (genau das macht das Archiv). Dies ist ein sehr einfaches und rudimentäres Vorgehen, um eine Sperrmöglichkeit bereitzustellen und ist nur als minimalistische Implementation zu verstehen
- Die Materialverwaltung bietet verschiedene Ereignisse an, die zum Beispiel darüber informieren, wenn ein neues Material eingetragen wurde oder ein Material bestimmter Art etc.
- Für jedes Material können (wahlweise synchrone/asynchrone) Ereignisse an der Materialverwaltung instantiiert werden (bei denen selbstverständlich Stubs ein- und ausgetragen werden können), diese können von außen angestoßen werden.

•

Schnittstelle der Materialverwaltung (Ausschnitt)

(An- und Abmeldung, Katalogfunktionen)

```
meldeMaterialAn:einMaterial vonDerArt:einString
          mitDemNamen:einString durch:einBenutzer
meldeMaterialAbVonDerArt:einString mitDemNamen:einString
          durch:einBenutzer
gibMaterialVonDerArt:einString mitDemNamen:einString fuer:einBenutzer
gibMaterialArtNamen
gibMaterialNamenVonDerArt:einString
```

Die ersten drei Methoden erlauben es, ein Material an- oder abzumelden, oder ein angemeldetes Material zu erhalten. Beim Anmelden wird das Material übergeben, und es wird sein Name und seine Art festgelegt. Außerdem wird ein Objekt übergeben, das den Besitzer kennzeichnet. Die Materialverwaltung hält nämlich eine Liste der Objekte, welche das Material verwenden. Wenn das Material nicht persistent ist (s.u.), „vergißt“ es die Materialverwaltung, wenn alle Interessenten ihren Zugriff auf das Material wieder abgemeldet haben.
Die letzten beiden Methoden listen alle Materialarten bzw. alle Materialnamen einer Art.

(Arten persistent machen)

```
macheArtUnloeschbar:einString
macheArtWiederLoeschbar:einString
meldeArtDauerhaftAn:einString
```

Diese Methoden verhindern/erlauben, daß mit dem Verschwinden aller Materialen einer Art auch die Materialart selbst von der Materialverwaltung „vergessen“ wird. Die dritte Methode verwendet die erste.

(Materialien persistent machen)

```
speichereMaterialVonDerArt:einString MitDemNamen:einString
          schreiber:einSchreiber leser:einLeser ort:einString
entspeichereMaterialVonDerArt:einString mitDemNamen:einString
```

Mit dieser Methode wird ein Material gespeichert. Ein solches Material muß den Aspekt *Speicherbar* implementieren. Zudem muß ein zum*Speicherbar*-Protokoll passendes Schreiber/Leser -Paar übergeben werden (hier findet das *Serializer/Atomizer*-Muster Verwendung, s. [Rie97b]). Als Ort wird ein String übergeben, der dem Leser/Schreiber als Information gelten kann. Bei dem von mir meist verwendeten CfsSchreiber / -Leser (Cfs steht für „Common File System“) ist dies ganz einfach der Pfad im Dateisystem.
Ein persistentes Material wird von der Materialverwaltung nicht vergessen, wenn keine Interessenten mehr angemeldet wird. Zwar wird die Referenz verworfen und der Garbage Collector gibt den Speicher frei. Wenn aber das Material erneut nachgefragt wird, wird es durch den entsprechenden Leser wieder instantiiert und herausgegeben.
Die zweite Methode lädt nicht etwa das Material, sondern hebt den Persistenzstatus auf.

(Sperr-Unterstützung)

Diese folgenden Methoden bilden das Rückgrat für darauf aufsetzende Locking-Mechanismen. Dies ist ein sehr einfaches und rudimentäres Vorgehen, um eine Sperrmöglichkeit bereitzustellen und ist nur als minimalistische Implementation zu verstehen: Es werden zwei Attribute zur Kennzeichnung

verschiedener Sperrzustände vorgesehen: Die Sperrkennung, ein positiver Integer ohne vordefinierte Semantik, außer der, das 0 als „nicht-gesperrt“ gelten soll, und der Sperr-String. Sehr einfache Mechanismen werden nur die Kennung verwenden, während für kompliziertere Modelle mit der Verwendung eines Strings alle Möglichkeiten offen stehen. Alle Methoden führen ihre Aktion thread-safe aus.

```
sperreMaterialVonDerArt:einString
          mitDemNamen:einString sperrKennung:einInteger
          sperrer:einString
sperreMaterialVonDerArt:einString mitDemNamen:einString
          einSperrBlock:einBlock
entsperreMaterialVonDerArt:einString mitDemNamen:einString
sperrInfoVonMaterialVonDerArt:einString mitDemNamen:einString
```

Die erste Methode versucht, bei dem Material eine neue Sperrkennung zu assoziieren. Dies gelingt nur dann, wenn die bisherige Kennung 0 ist. Nur dann wird auch der als letzter Parameter übergebene „Sperrer“ (ein String) gesetzt. Rückgabewert ist die neue Kennung bei Erfolg, sonst 0.

Die zweite Methode ist differenzierter. Hier wird als dritter Parameter ein Block übergeben, der als Parameter ein Objekt erhält, welches Sperrkennung und Sperr-String enthält. Dieses Objekt kann im Block verändert werden (Kennung und String werden hier also gewissermassen „per Referenz“ in den Block übergeben). Damit ist also eine genau unterscheidende Transition zwischen allen denkbaren Sperrzuständen möglich.

Die dritte Methode setzt die Kennung auf 0 zurück und den „Sperrer“ auf einen Leerstring. Sie darf nur von demjenigen gerufen werden, der weiß, daß er in der jeweiligen Sperrsemantik das Recht dazu hat.

Die vierte Methode liefert als Rückgabewert eine Sperr-Info, bestehend aus Sperrkennung und Sperrstring.

(Ereignisse: Stubs ein- und austragen)

```
beiMaterialEingefuegtTrageStubEin:einEventStub
beiMaterialEingefuegtTrageStubAus:einEventStub
beiMaterialEntferntTrageStubEin:einEventStub
...
```

Mit diesen und weiteren Methoden kann man sich über das Eintreten bestimmter Ereignisse an der Materialverwaltung informieren lassen (ein Material wurde eingefuegt, gelöscht etc.)

```
beiMaterialEingefuegtVonDerArt:einString trageStubEin:einEventStub
beiMaterialEingefuegtVonDerArt:einString trageStubAus:einEventStub
beiMaterialUmbenanntVonDerArt: einString trageStubEin:einEventStub
...
```

Für diese Methoden gilt das gleiche wie für den vorigen Block. Sie leisten dasselbe, nur mit Blick auf bestimmte Materialien.

(Ereignisse: Erzeugung und Auslösen)

```
erzeugeEreignisSyncMitName:einString beschreibung:einString
          parameterbeschreibung:einString
          fuerMaterialMitName:einString VonDerArt:einString
teileMitEreignisMitName:einString fuerMaterialMitName:einString
          vonDerArt:einString parameter:einObject
          aberNichtAn:einEventStub
gibEreignisNamenFuerMaterialVonDerArt:einString  mitDemNamen:einString
trageStubEin:einEventStub beiEreignisMitName:einString
          fuerMaterialMitName:einString vonDerArt:einString
```

Die erste Methode dient dazu, ein synchrones Ereignis einem Material zuzuordnen, das dann mit der zweiten Methode ausgelöst werden kann (die entsprechende Methode existiert auch für asynchrone Ereignisse).
Der letzte Parameter in der zweiten Methode dient dabei dazu, einen Stub von der Benachrichtigung auszuschließen. Diese Möglichkeit ist vorgesehen, da konstruktionsbedingt häufig auch das ereignisauslösende Objekt selber einen Stub angemeldet hat.
Die beiden letzten Methoden listen die den einzelnen Materialien zugeordneten Ereignisse und erlauben das Anmelden von Stubs bei ihnen.

7 Zusammenfassung und Ausblick

Ich habe in dieser Arbeit eine Aufgabenstellung (Bau einer Pausenplaner-Anwendung) und eine Programmiersprache bzw. Entwicklungsumgebung (VisualAge für Smalltalk + Distributed Feature) als Ausgangspunkt genommen, um die Realisierung einer verteilten Anwendung im WAM-Kontext auf dieser Basis vorzustellen.

Zu diesem Zweck habe ich zunächst in bescheidenem Umfang WAM-Grundbegriffe erläutert und mich dann mit den Begriffsbildungen zu kooperativer Arbeit befaßt, die sich im Rahmen von WAM entwickelt haben. Bei der anschliessenden Vorstellung der Problemstellung, die der Pausenplaner bearbeiten soll, ist deutlich geworden, daß sich für die Pausenplaner-Anwendung ein einfaches Archiv als implizites Kooperationsmedium gut eignet. Ich habe an dieser Stelle noch die wichtigsten Materialien un die Werkzeuge der Anwendung beschrieben.

Um die technischen Grundlagen der Konstruktion zu klären, habe ich in einem eigenen Abschnitt die Eigenschaften von VisualAge for Smalltalk, insbesondere des „Distributed Feature“ dargestellt.

Nach diesen Vorarbeiten konnte ich nun die durch die Verteilung aufgeworfenen oder zumindest stärker ins Blickfeld geratenen Fragen behandeln: Zunächst (als unabdingbare Voraussetzung) die Klärung der Begriffe Original und Kopie im Kontext dieser Anwendung, und im Anschluss die Frage, in welcher Form und mit welchem Benutzungsmodell Anwender verteilt auf die Materialien zugreifen.

Der letzte Abschnitt schließlich stellte einige Details der Implementation dar, die mir besonders wichtig erschienen, insbesondere den schichtartigen Aufbau der Archiv-Funktionalität und die Aufteilung der einzelnen Objekte auf verschiedene Rechner.

Durch den Modellcharakter der Anwendung und damit verbundene, recht „weiche“ Anforderungen ist ein Teil der Implementationsentscheidungen sicher nicht ausreichend am realen Betrieb einer solchen Anwendung gemessen und optimiert worden. Dennoch sind einige Aspekte, so z.B. die Idee der schichtenartigen Konstruktion eines Archivs, als Konzeption tragfähig auch für andere, verwandte Aufgabenstellungen.

Anhang: Komplexe Materialien und deren Synchronisation beim Pausenplaner

Dieser Abschnitt behandelt eine Problematik, welche in der Pausenplaner-Anwendung zutage tritt: den Umgang mit komplexen Materialien. Dieses Problem ist, wenn man einen großen, allgemeinen Lösungsansatz versuchen wollte, sehr schwierig und umfangreich. Ich werde im folgenden nur mein Vorgehen beim Pausenplaner erläutern, ohne den Anspruch einer vielseitigen Wiederverwendbarkeit dieser Lösung zu erheben.

Als *komplexes Material* soll ein Material verstanden werden, welches auf kleinteiligeren Materialien aufbaut und von diesen abhängt.

Im konkreten Beispiel ist der Pausenplan ein solches Material, denn er verwendet einen bestimmten Lehrkörper. In seine Pausen werden Lehrer aus diesem Lehrkörper eingetragen, und Restriktionen bei diesen Einfügungen entstehen aus den Ausschlußzeiten, die vom Lehrkörperobjekt erfragt werden. Das bedeutet, daß fast alle Methoden des Pausenplanes einen validen Lehrkörper voraussetzen; ein Pausenplan ist nur in einem gültigen Zustand, wenn er einen gültigen Lehrkörper enthält.

Der Lehrkörper ist in diesem Fall das kleinteiligere Material, auf welches aufgebaut wird. Er selbst ist unabhängig von den Pausenplänen, die sich auf ihn beziehen. Die Erstellung und Änderung eines Lehrkörpers setzt nicht einmal die Existenz irgendeines Pausenplanes voraus.

Die Problematik scheint (für den recht einfachen Fall der Pausenplaner-Anwendung) immerhin einigermaßen beherrschbar, solange man sich ein System nur mit „lebendigen“, das heißt: im Laufzeitsystem existierenden Objekten vorstellt: Die Integrität kann allein dadurch weitgehend gesichert werden, daß man die Lehrer in den Pauseneinträgen direkt referenziert (allerdings führt das Löschen eines Lehrers aus dem Lehrkörper dabei nicht automatisch zum Löschen aus den Pauseneinträgen). Doch es geht eben nicht nur um solche Objekte, da die Pausenplaner-Anwendung ja Persistenz der Pausenpläne und Lehrkörper bietet. Mit dieser zusätzlichen Eigenschaft wird die Fragestellung komplizierter.

Besonders problematisch ist es nämlich, wenn das komplexe Material gespeichert werden soll, obwohl seine kleineren Bestandteile unhabhängig von ihm bearbeitet werden können: Wenn man das eben dargestellte Verfahren fortschreibt, würde mit der Änderung eines Lehrkörpers die Situation entstehen, daß man sämtliche auf ihn bezogenen persistenten (und nicht in Bearbeitung befindlichen) Pausenpläne laden und abgleichen müßte. Dies ist ein sehr aufwendiges Verfahren. Die Alternative dazu ist natürlich, die im vorigen Absatz angedeutete Referenzierung des Lehrkörpers auch in der Persistenz abzubilden.

Vom relationalen Modell her besehen könnte man sich die Verweise des Pausenplanes auf Lehrer etwa als *Foreign Keys* vorstellen (die z.B. auch noch dem Constraint *On Delete Cascade* unterliegen - das hieße also, daß ein Löschen eines Lehrers aus dem Lehrkörper zum Löschen aller entsprechenden Verweise in Pausenplänen führen würde).

Die Pausenplaner-Anwendung ist aber keine datenbankzentrierte Software und soll es auch nicht werden. Dennoch bietet es sich an, dem relationalen Modell die Schlüssel entlehnen: Man könnte für die Lehrer in einem Lehrkörper eindeutige Schlüssel über die Kürzel hinaus vergeben, also Schlüssel, die nicht fachlich gegeben sind, sondern rein technisch motiviert, und die Einträge im Pausenplan über diese Schlüssel vornehmen. Damit würde z.B. eine Namensänderung bei einem Lehrer, die auch eine Kürzeländerung nach sich zieht, weiterhin völlig korrekt in jedem der sich auf ihn beziehenden Pausenpläne reflektiert.

Dieses Vorgehen würde technisch gesehen gut und effizient funktionieren, hat aber zwei Nachteile:

- Mit technischen Schlüsseln wird die rein fachliche Gestaltung der Schnittstellen für die Materialklassen verunreinigt.
- Die Schnittstelle des Lehrkörpers weist Methoden auf, die mit technischen Schlüsseln für Lehrer hantieren, obgleich diese Schlüssel nur im Zusammenhang mit Pausenplänen gebraucht werden. Der Lehrkörper aber ist ein eigenständiges Material und seine Schnittstelle verliert so die anzustrebende Minimalität.
- Die Anpassung des Pausenplanes an den veränderten Lehrkörper ist auf diesem Wege ein reiner Automatismus, denn der Pausenplaner hat keine Wahl, als sich ohne jede Nachfrage beim Benutzer den Änderungen am Lehrkörper anzupassen: beim nächsten Laden eines Pausenplanes ist mit einem Mal ein Lehrer einfach verschwunden . Tatsächlich könnte aber ja ein Anwender auch den Wunsch haben, seinen Pausenplan weiter auf einer alten Version des Lehrkörpers basieren zu lassen (dann müßte dieser als eigenständiges Material in das Pausenplaner-Archiv eingefügt werden).
-

Deswegen habe ich das Problem etwas anders gelöst. Der Lehrkörper vergibt für die in ihm vorhandenen Lehrer zwar technische Schlüssel, aber nur*intern.* Damit gewinnt er die Möglichkeit, sich mit einem anderen Lehrkörper zu vergleichen . Der Lehrkörper bietet nun an seiner Schnittstelle Methoden an, die einen anderen Lehrkörper als Parameter haben und ausweisen, welche Lehrer in einem anderen Lehrkörper nicht mehr vorhanden sind oder sich geändert haben, und welche dazu gekommen sind. Außerdem kann er, welcher Lehrer des anderen Lehrkörpers einem seiner Lehrer entspricht.

Ein weiterer Vorteil dieses Vorgehens ist, daß ein persistenter Pausenplan mit seinem Lehrkörper nicht gleich abgeglichen werden muß, sobald sich an diesem etwas ändert. Es reicht, wenn dies geschieht, sobald der Pausenplan wieder ins Leben gerufen wird. Die Aufgabe, diesen Abgleich durchzuführen, wird durch einen*PausenPlanBauer* erfüllt. Dieses geht so vor:

- Der PausenPlanBauer erhält den Pausenplan. Mit einem Pausenplan wird auch der jeweilige Lehrkörper serialisiert, das heißt also, daß nach dem Laden des Pausenplanes dieser in einem konsistenten Zustand ist, wenn man davon absieht, daß sein Lehrkörper unter Umständen veraltet ist.
- Der PausenPlanBauer erfragt vom Pausenplan den Namen des Lehrkörpers und holt sich vom Pausenplaner-Archiv die aktuelle Version.
- Das PausenPlanBauer erfragt vom „alten“ Lehrkörper in einzelnen Schritten, wie er sich von der aktuellen Version unterscheidet. Entsprechend entfernt er die inzwischen gelöschten Lehrer aus dem Pausenplan und paßt die Lehrer im „alten“ Lehrkörper an. Auch die neu hinzugekommenen Lehrer werden eingefügt.
- Danach ist der Pausenplan mit dem aktuellen Stand des Lehrkörpers synchronisiert.

Von Vorteil ist bei diesem Vorgehen insbesondere, daß der Synchronisationsprozeß mit dem Benutzer abgestimmt werden kann. Man kann den Benutzer schrittweise über die gefundenen Veränderungen informieren und fragen, ob er all diese Änderungen übernehmen will. Wenn nicht, muß er allerdings den nicht synchronisierten Lehrkörper unter einem neuen Namen speichern; der Pausenplan bezieht sich von nun an auf diesen Lehrkörper. Der Nachteil bei diesem Vorgehen ist, daß der Synchronisationsprozeß sehr aufwendig ist (was in diesem Fall zum Zwecke der Benutzerinteraktion nicht unerwünscht ist) und die Frage nach der Natur von Referenzen nicht wirklich geklärt, sondern umschifft wird. Das schränkt die Übertragbarkeit dieses Vorgehens für ähnliche Probleme mit komplexen Materialien ein.

Abbildungsverzeichnis

Literaturverzeichnis

[Fel97] Andreas Felten, Christian Langmann.*Entwicklung einer CORBA-basierten vertielten Anwendung mit Distributed Smalltalk am Beispiel eines Pausenplaners.* Studienarbeit, Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1997.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides.*Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gat96] Michael Gatzhammer, Valentino Kyriakides.*Konstruktion interaktiver Software nach der Methode WAM in Smalltalk am Beispiel von VisualAge und VisualWorks.* Studienarbeit. Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1996.

[Gol83] A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[Gol89] A. Goldberg, D. Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989.

[Gry96] G. Gryczan. *Prozeßmuster zur Unterstützung kooperativer Tätigkeit.* Gabler-Verlag, 1996.

[IBM97a] IBM. *VisualAge for Smalltalk Distributed User's Guide, Version 4.0.* IBM Corporation, 1997.

[IBM97b] IBM. *VisualAge for Smalltalk User's Guide, VisualAge for Smalltalk User's Reference, Version 4.0.* IBM Corporation, 1997.

[LaL94] Wilf LaLonde, John Pugh.*Just Cloning Around* in: *Journal of Object-Oriented Programming,* Sept. 1994.

[LaL96] Wilf LaLonde, John Pugh.*Preparing to use the distributed facility in IBM Smalltalk / Using the distributed facility in IBM Smalltalk to implement a distributed bank account browser* in *Journal of Object-Oriented Programming,* May / June 1996.

[Loc87] P.C. Lockemann, J.W. Schmidt. *Datenbankhandbuch.* Springer Verlag, 1987.

[Oes97] Bernd Oesterreich. *Objektorientierte Softwareentwicklung mit der Unified Modeling Language.* R. Oldenbourg Verlag, 1997.

[Rie97a] Dirk Riehle. *Entwurfsmuster für Softwarewerkzeuge.Gestaltung und Entwurf von Anwendungen mit grafischer Benutzeroberfläche.* Addison-Wesley, 1997.

[Rie97b] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, Heinz Züllighoven*The Serializer Pattern* in: Robert C. Martin, Dirk Riehle, Frank Buschmann.*Pattern Languages of Program Design 3.* Addison-Wesley, 1997, Chapter 17.

[Zül98] Heinz Züllighoven.*Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Materialansatz.* dpunkt-Verlag, Heidelberg, 1998.