Studienarbeit

Entwicklung einer CORBA-basierten verteilten Anwendung mit Distributed Smalltalk am Beispiel eines Pausenplaners

Prof. Heinz Züllighoven
Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

07. Dezember 1998

Andreas Felten
Brunsberg 14
22529 Hamburg
Matr.-Nr: 4555777

Christian Langmann
Kiwittsmoor 21
22417 Hamburg
Matr.-Nr.: 4527372

Wir bestätigen hiermit, daß wir die hier vorgelegte Studienarbeit ausschließlich mit den aufgeführten Mitteln erstellt haben.

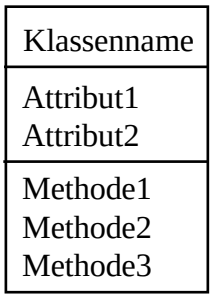
Inhaltsverzeichnis

Abkürzungsverzeichnis

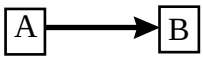
AM	Application Model
AO	Application Object
BOA	Basic Object Adaptor
CORBA	Common Object Request Broker Architecture
COSS	Common Object Service Spezification
DII	Dynamic Invocation Interface
DM	Domain Model
DSI	Dynamic Skeleton Interface
DSOM	Distributed System Object Model
DST	Distributed Smalltalk
ev	Ereignisverwalter
FK	Funktionskomponente
IAK	Interaktionskomponente
IDL	Interface Definition Language
IIOP	Internet Inter ORB Protocol
IR	Interface Repository
LK	Lehrkörper
mv	Materialverwalter
ODBMS	Objektorientiertes Datenbank-Management System
OMG	Object Management Group
OO	Objektorientierung
OQL	Object Query Language
ORB	Object Request Broker
P/S-Split	Presentation/Semantic-Split
PO	Presentation Object
PP	Pausenplaner
RDBMS	Relationales Datenbank-Management System
RFP	Request For Proposal
RPC	Remote Procedure Call
SO	Semantic Object
SQL	Structured Query Language
WAM	Werkzeug/Aspekt, Automat/Material (-Metapher)

Legende

Notation für Klassendiagramme



Klasse mit Attributen
und Methoden



Klasse A erbt von Klasse B

Assoziation

Aggregation

Benutzt-Beziehung

Notation für Interaktionsdiagramme

asynchrones Ereignis

Methodenaufrufe

Vom Entwickler zu implementierender Aufruf

Kontrollfluß

Vom Entwickler zu überschreibende Methode

Objekt

Trennung zwischen Benutzer und Implementation

1 Einleitung

Am Arbeitsbereich Softwaretechnik des Fachbereiches Informatik an der Universität Hamburg steht die Einführung der CORBA-Implementation Distributed Smalltalk von ParcPlace (DST) bevor. Der Arbeitsbereich hat in den letzten Jahren Erfahrungen mit der Konstruktion von unverteilten Einzelplatzanwendungen gesammelt. Diese Arbeit dient dazu, erste Erfahrungen mit der Entwicklung von verteilten Mehrbenutzeranwendungen zu gewinnen.

In den Übungen zu der Vorlesung Objektorientierte Konstruktion (WS 96/97) wurde unter VisualWorks 2.5 zur Vertiefung der in der Vorlesung gehörten Konzepte ein Programm zur Bearbeitung von Pausenplänen erstellt. Diese Aufgabe enthielt das Potential, Problematiken von verteilten Anwendungen beispielhaft zu bearbeiten. Daher wurde sie als Implementationsbeispiel für diese Arbeit verwendet (Kapitel 2).

DST entspricht der von der Object Management Group (OMG) vorgeschlagenen Architektur für verteilte objektorientierte Anwendungsentwicklung. Diese Architektur findet Ausdruck in der CORBA-Spezifikation und wird in Kapitel 3 beschrieben.

In Kapitel 4 wird die Benutzung von DST aus der Perspektive eines Benutzers dargestellt. Distributed Smalltalk bietet ein Programmiermodell an, welches die Grundlage der Anwendungen dieser Arbeit bildet. Dieses und die Eigenheiten der CORBA-Implementation, werden in diesem Kapitel beschrieben.

Darauf aufsetzend werden in Kapitel 5 die Konzepte und die Implementation des Pausenplaners vorgestellt. Dabei wurde Wert auf erweiterbare und wiederverwendbare Konzepte gelegt. Daraus sind die Ereignis- und Materialverwaltung hervorgegangen, die unabhängig von dieser Anwendung sind und auch in anderen CORBA-Implementationen eingesetzt werden können. Eine Vorgabe dieser Arbeit besteht darin, die in DST angebotenen Modelle zu benutzen. In diesem Kapitel werden im Hinblick auf Verteilung Rückschlüsse auf die am Arbeitsbereich bevorzugte Enwicklungsmetapher WAM (Werkzeug/Aspekt-Automat/Material) gezogen.

Anhand der durch die Arbeit mit DST gewonnenen Erfahrung wird eine kritische Auseinandersetzung mit dieser Entwicklungsumgebung vorgenommen (Kapitel 6). Da ParcPlace leider auf Anfragen bzgl. einiger Probleme keine Reaktion gezeigt hat, ist diese Bewertung nur Aufgrund eigener Erfahrungen entstanden.

Schließlich wird in Kapitel 7 ein Ausblick auf Erweiterungen gezeigt, auf die im Rahmen dieser Arbeit nicht näher eingegangen werden soll.

Die in dieser Arbeit entwickelten Klassen und Konzepte werden dem interessierten Leser selbstverständlich zur Verfügung gestellt. Dieser kann entweder das komplette DST-Image verwenden oder die Quelltexte in seine Arbeitsumgebung einbinden.

2 Ein Pausenplaner

Nach dem Gesetz sind Schulen dazu verpflichtet, für jede Pause, die länger als 5 Minuten dauert, Lehrer als Pausenaufsicht abzustellen. Die Anzahl der Lehrer pro Pause hängt dabei von der Größe des zu beaufsichtigenden Gebäudes und des Schulhofes ab. Um dem Pausenplan-Koordinator seine Arbeit zu erleichtern, wurde von uns der dieser Arbeit zu Grunde liegende Pausenplaner entwickelt. Dieser steht modellhaft für eine bestimmte Schule. Die Situationsbeschreibung wird anhand eines Szenarios in Kapitel 2.1 beschrieben. Anschließend wird eine Vision vorgestellt, in der dargelegt wird, wie ein Softwareprodukt die Arbeit mit dem Pausenplan unterstützen kann (Kapitel 2.2).

2.1 Szenario eines Pausenplans

Grundsätzlich dient ein Pausenplan dazu, die Lehrer des Lehrkörpers einer Schule den einzelnen Pausen der Schule zuzuordnen. Jeder Lehrer hat dabei die Möglichkeit, Zeiten anzugeben, zu denen er keine Pausenaufsicht führen kann, da er außerschulischen Aktivitäten nachgeht, nur Teilzeit unterrichtet oder aus anderen Gründen keine Zeit hat. Die Zuordnung der Lehrer zu den Pausen kann sehr komplex werden, da die Lehrer keine weiten Wege zurücklegen dürfen von dem Raum, in dem sie als letztes unterrichtet haben, bis zu dem Ort, den sie beaufsichtigen sollen, da sonst ein zu großer Teil der Pause ohne Pausenaufsicht vergeht.

Am Anfang eines jeden Schulhalbjahres wird vom Pausenplan-Koordinator der Pausenplan festgelegt. Im Normalfall gilt dieser für das ganze Schulhalbjahr. Ausnahmen können auftreten, wenn Lehrer krank werden oder wenn neue Lehrer an die Schule kommen bzw. Lehrer die Schule verlassen. Die Berechnung, wie viele Pausen ein Lehrer beaufsichtigen muß, ergibt sich aus folgender Formel:

$$P(l_x) = \frac{A(l_x) * GP}{GA},$$

mit:
- l_X - Lehrer x
- $P(l_x)$ - Pausenaufsichten für Lehrer X pro Woche
- $A(l_x)$ - Arbeitsstunden von Lehrer X pro Woche
- GP - Gesamtzahl der Pausenaufsichten pro Woche
- GA - Summe der Arbeitsstunden aller Lehrer

Bei der Rundung der Pausenaufsichten auf eine Kardinalzahl muß darauf geachtet werden, daß die Summe der einzelnen Pausenaufsichten nach dem Runden die tatsächliche Anzahl der zu beaufsichtigenden Pausen ergibt.

In der konkreten Schule, für die der Pausenplan entwickelt wird, gelten folgende Randbedingungen, auf die der Pausenplan-Koordinator bei seiner Arbeit achten muß: Es gibt einen kleinen und einen großen Schulhof und drei Pausenzeiten, in denen Pausenaufsicht geführt werden muß. Zur ersten und zweiten Pausenzeit müssen beide Schulhöfe beaufsichtigt werden, in der Spätaufsicht nur einer. Bis auf bestimmte Ausnahmen muß jede Pausenaufsicht von zwei Lehrern gehalten werden. Nur Dienstag bis Donnerstag müssen die Pausen auf dem großen Schulhof von drei Lehrern beaufsichtigt werden. Ein Lehrer kann nicht gleichzeitig zwei Orte beaufsichtigen. Prinzipiell muß mindestens ein Lehrer einer Pausenaufsicht auf jedem Schulhof die Pause verantwortlich beaufsichtigen können. Diese Eigenschaft ist für

jeden Lehrer definiert. Zusätzlich wird für jeden Tag eine Vertretungsliste angelegt, die vier Lehrer aufnimmt. Diese Lehrer sollten nicht am gleichen Tag bereits Aufsicht führen, da sie sich nicht selbst vertreten können. Ein Beispiel für einen ausgefüllten Pausenplan stellt Abb. 1 dar.

		Mo	**Di**	**Mi**	**Do**	**Fr**
Kleiner Schulhof	**1. Pause**	LAN	OTT	TEI	MUE	MEI
		FEL	LAN	LEU	PUT	SUS
	2. Pause	KOW	HAS	FEL	LAN	MEI
		OTT	TEI	LAN	KOW	TEI
Großer Schulhof	**1. Pause**	PUT	LEU	MUE	SUS	HAS
		TEI	PUT	SUS	LAN	FEL
			MEI	KOW	FEL	
	2. Pause	LAN	FEL	GRY	OTT	FEL
		HAS	LEU	TEI	SUS	LAN
			KOW	LEU	GRY	
Spätaufsicht		LAN	OTT	BRA	MUE	HAS
		OTT	FEL	MUE	LAN	FEL
Vertretung		LEU	GRY	PUT	BRA	BRA
		MEI	MUE	MEI	LEU	LEU
		MUE	SUS	HAS	TEI	MUE
		GRY	BRA	OTT	MEI	KOW

Abb. 1: Ein Pausenplan

Für jeden Lehrer existiert eine Lehrerkarte, auf der seine Daten zu finden sind (Name, Kürzel, verantwortlich Pausenaufsichtsberechtigt, Ausschlußzeiten, Beschäftigungsgrad). Der Pausenplan-Koordinator ordnet diese Karten so auf dem Pausenplan an, daß die oben genannten Kriterien möglichst erfüllt werden. Nach jeder Änderung wird die Pausenstatistik aktualisiert, in der die Ist- und die Soll-Pausen von jedem Lehrer enthalten sind.

2.2 Systemvision: Parallele Nutzung des Pausenplaners

Der Pausenplaner dient dazu, den Pausenplan-Koordinator bei seiner Arbeit zu unterstützen: Die Anzeige des Pausenplaners soll eine schnelle Übersicht der Lehrerverteilung auf die Pausen ermöglichen und gleichzeitig mögliche Konflikte, wie eingetragene Lehrer, die nicht dem Lehrkörper angehören oder zur Pausenaufsicht keine Zeit haben, hervorheben.

Der Schwerpunkt dieser Arbeit liegt in der Verteilung des Pausenplaners auf verschiedene Rechner. Es soll möglich sein, daß mehrere Personen auf verschiedenen Rechnern Lehrer erfassen und manipulieren können. Dies ist die Aufgabe des Sekretariats. Weitere Personen auf wiederum anderen Rechnern sollen diesen Lehrern Pausen zuordnen können (Pausenplan-Koordinator). Dabei müssen alle Personen mit aktuellen Daten versorgt werden, so daß Änderungen sofort allen anderen Personen, die die gleichen Daten bearbeiten, angezeigt werden. Wenn sich beispielsweise bei einem Lehrer eine Zeit ergibt, an der er nicht Pausenaufsicht führen kann, er aber zu dieser Zeit bisher Pausenaufsicht hatte, so muß der Pausenplan diesen Lehrer in der betreffenden Pause hervorheben, damit darauf aufmerksam gemacht wird, daß diese Pause konfliktär belegt ist. Bei der Verteilung muß also berücksichtigt werden, daß jedes Tool auf jedem Rechner gestartet werden kann und über Veränderungen der Daten durch andere Benutzer informiert wird.

Um dem Pausenplan-Koordinator in Anlehnung an das Leitbild des qualifizierten Arbeitsplatzes einen möglichst flexiblen Arbeitsablauf zu ermöglichen, müssen fehlerhafte Eingaben zwar zugelassen, jedoch adäquat angezeigt werden.

Der Pausenplan soll folgende Probleme hervorheben:

- ein eingetragener Lehrer gehört nicht mehr der Schule an
- ein Lehrer ist in zwei unterschiedlichen Schulhöfen zur gleichen Zeit eingetragen
- ein Lehrer ist an einem Tag sowohl in einer Pause als auch in der Vertretungsliste eingetragen
- ein Lehrer hat zu einer Pausenaufsicht keine Zeit
- ein Lehrer hat an einem Tag, an dem er in die Vertretungsliste eingetragen ist, zu irgendeinem Zeitpunkt keine Zeit
- eine Pause wird von zuwenig Lehrern beaufsichtigt
- eine Pause wird von keinem verantwortlichem Lehrer beaufsichtigt

3 CORBA

CORBA (Common Object Request Broker Architecture) ist eine offene Architektur für verteilte Objekte. Sie wurde von der OMG (Object Management Group), in der über 500 Unternehmen organisiert sind, spezifiziert. CORBA beschreibt mit einer eigenen Spezifikationssprache - genannt IDL (Interface Definition Language, Kapitel 3.1) - die Schnittstellen von Objekten auf welche die Clients zugreifen können. Die aktuelle CORBA Version ist 2.0 vom Dezember 1994. Diverse CORBA-Implementierungen (z.B. OrbiX von Iona, DSOM von IBM, ORB Plus von HP, ObjectBroker von Digital, DST von ParcPlace) sind bereits erschienen.

Programme, die nach dem objektorientierten Paradigma entwickelt werden, bestehen aus einem Geflecht mehrerer miteinander kommunizierender Objekte. Um solche Objektkompositionen bauen zu können, müssen die beteiligten Objekte kompatibel zueinander sein. Diese Eigenschaft sollte auch gegeben sein, wenn die Objekte in unterschiedlichen Programmiersprachen entwickelt werden oder auf verschiedenen Hard- oder Software-Plattformen laufen. Um dies zu realisieren, ist CORBA als Standard für heterogene Client/Server-Umgebungen entwickelt worden. Die die Architektur von CORBA ausmachenden Komponenten sind in Kapitel 3.2 beschrieben.

„CORBA Objekte sind ein Stück Intelligenz, die sich an beliebigen Stellen im Netzwerk befinden können. Sie existieren in Form von binären Paketen, auf die von entfernten Clients über Methodenaufrufe zugegriffen werden kann“ [Übers. der Autoren aus ORFA96, S. 49]. Für die Clients ist es im allgemeinen nicht notwendig zu wissen, wo und auf welcher Plattform sich die Server-Objekte befinden (im selben Prozeß oder auf einem anderen Rechner). Das Verbergen des tatsächlichen Aufenthaltsorts vor dem Client wird Ortstransparanz genannt. Die Clients brauchen ebenso nicht zu wissen, wie (und in welcher Programmiersprache) das Serverobjekt implementiert ist. Der Client muß ausschließlich das Interface des Servers, das in IDL definiert ist, und eine Referenz auf das (entfernte) Objekt kennen, um von dem Serverobjekt Methoden aufrufen zu können. Das Interface und die Objektreferenz dienen der Verbindung von Client und Serverobjekt. Die bereits in CORBA definierten Dienste (Kapitel 3.3) sind mit einer IDL-Schnittstelle versehen und können als Serverobjekte angesehen werden.

3.1 IDL

Die Interface Definition Language (IDL) ist eine rein deklarative, programmiersprachen- und plattformunabhängige stark getypte Spezifikationssprache. Durch IDL wird die Spezifikation von der Implementation der Objekte getrennt. Alle CORBA-Objekte besitzen ein Interface in IDL, in dem die Attribute, Oberklassen, Exceptions, Datentypen und Methoden mit Input/Output-Parametern beschrieben sind. Auf diese Beschreibung können die Clients zugreifen, um die Dienste der Objekte in Anspruch zu nehmen.

Die Grammatik von IDL ist eine Untermenge von C++ mit Preprozessor Features und Pragmas sowie einiger Schlüsselworte für die Erweiterung um verteilte Konzepte. Es gibt keine prozeduralen Stukturen und Variablen. IDL dient der Unterstützung von Software-Architekten: Aus den IDL-Spezifikationen können mittels eines IDL-Compilers häufig direkt Programmskelette (z.B. Headerfiles in C++) für Client und Server generiert werden.

Abb. 2 zeigt den Aufbau eines IDL-Files, Abb. 3 zeigt ein Anwendungsbeispiel. Module definieren einen Namensraum, um zusammengehörige Klassenbeschreibungen zu gruppieren. Die Interfaces entsprechen den Klassendefinitionen, sie besitzen nur keinen Implementationsteil. Ein Interface kann Attribute besitzen. Aus ihnen wird automatisch eine get- und eine set-Routine generiert. Da IDL Mehrfachvererbung unterstützt, kann ein Interface von mehreren anderen Interfaces erben. Die Interfaces können Methoden (Operations) enthalten. Bei jeder Methode wird der Rückgabetyp (<op_type>) sowie die Parametertypen und -modes angegeben. Es gibt drei Arten von Parametermodi: Der Mode *in* ist für Parameter, die vom Client zum Server übergeben werden, der Mode *out* für Parameter, die vom Server zum Client gegeben werden und der Mode *inout* für beide Richtungen. In der raises-Klausel hinter den Methodendefinitionen wird angegeben, welche Exceptions von der Methode geworfen werden können.

```
module <identifier>
{
  <type declarations>;
  <constant declarations>;
  <exception declaration>;

  interface <identifier> [:<inheritance>]
  {
      <type declarations>;
      <constant declarations>;
      <attribute declarations>;
      <exception declaration>;

      [<op_type>] <identifier> (<parameters>)
           [raises exception] [context];
                 :
      [<op_type>] <identifier> (<parameters>)
           [raises exception] [context];
                 :
  };

  interface <identifier> [:<inheritance>]
                 :
};
```

Abb. 2: Syntax eines IDL-Modules

Die Datentypen von IDL sind in vielen Sprachen bekannt. Es gibt Basistypen (short, long, unsigned long, unsigned short, float, double, char, boolean, octet) und konstruierte Typen (enum, string, struct, array, union, sequence, any). Die im Beispiel benutzten Pragmas sind in IDL definiert und müssen in Distributed Smalltalk benutzt werden, um Widersprüche in der syntaktischen Schreibweise von Methodennamen auszugleichen: In IDL sind Doppelpunkte syntaktisch nicht gestattet und die Parameter werden in Klammern nach dem Methodennamen angegeben. In Smalltalk hingegen wird ein Parameter immer hinter einem Doppelpunkt und innerhalb des Methodennamens angegeben. Daher muß bspw. die Smalltalkmethode *von: param1 bis: param2* in einen IDL-konformen Namen wie *vonBis(param1,param2)* umgesetzt werden.

```
module Zeitmodul
{ ...
  interface ZeitraumInterface : SmalltalkObject
  {
          #pragma selector setzeBis bis:
          void setzeBis( in ZeitInterface eineZeit );

          #pragma selector setzeVon von:
          void setzeVon( in ZeitInterface eineZeit );

          #pragma selector vonBis von:bis:
          void vonBis( in ZeitInterface b, in ZeitInterface e );

          boolean ueberschneidung( in ZeitraumInterface z );
          ZeitInterface von();
          ZeitInterface bis();
          string gibBeschreibung();
  }; ...
};
```

Abb. 3: IDL-Beispiel

Die in IDL spezifizierten Methoden können in einer beliebigen Sprache, die allerdings das CORBA-Binding unterstützen muß, implementiert werden (bisher C, C++, Smalltalk, Cobol,

Ada, Objective C). Wie Abb. 4 ersichtlich, werden die Datentypen in die entsprechenden Typen der Implementationssprache umgesetzt (gemappt).

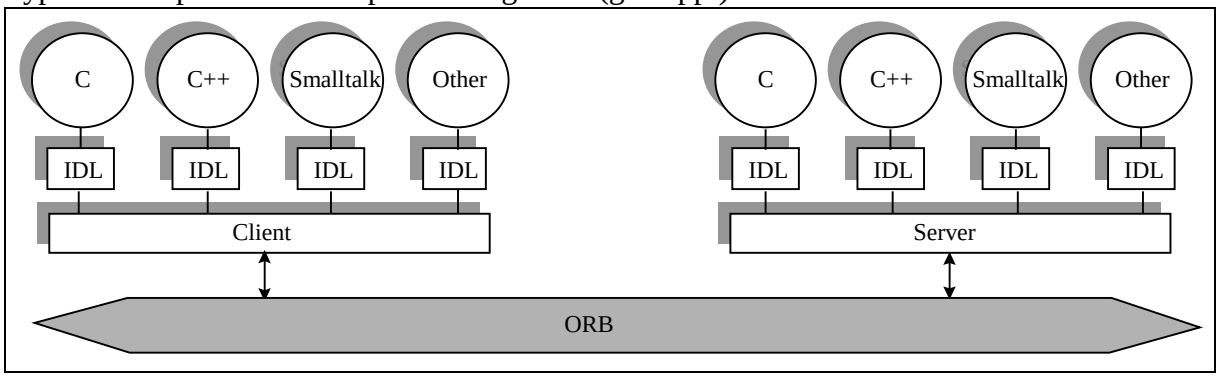


Abb. 4: Sprachunabhängigkeit von CORBA mittels IDL

3.2 Komponenten

Die Komponenten von CORBA sind im Abb. 5 zu sehen. Obwohl die meisten dieser Komponenten gerade bei der Entwicklung mit Distributed Smalltalk kaum in Erscheinung treten, so bilden sie die Grundlage für das Verständnis von CORBA und werden daher in den folgenden Unterkapiteln erläutert.

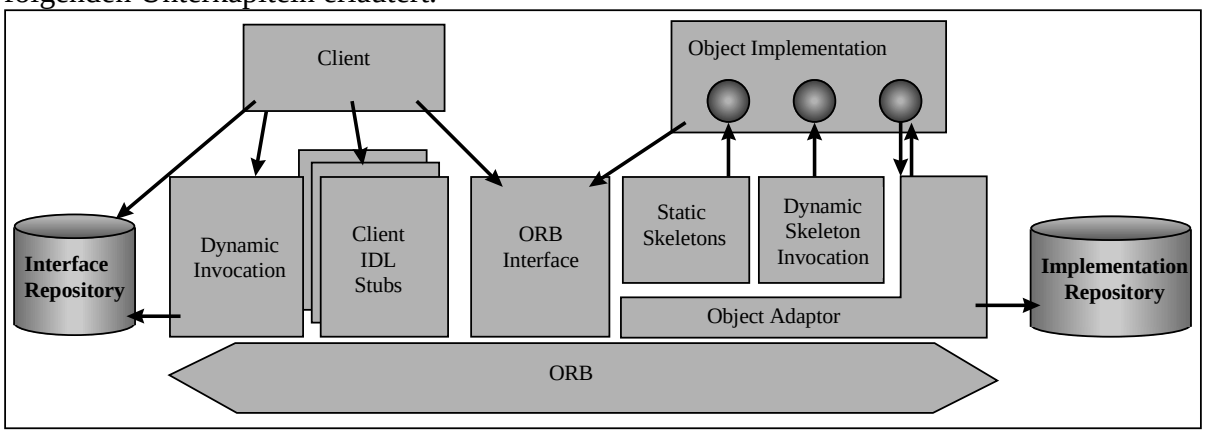


Abb. 5: Komponenten von CORBA

3.2.1 ORB

Der ORB ist eine Middleware, welche die Client/Server-Beziehung zwischen (verteilten) Objekten realisiert. Der ORB ist somit eine Art „open intergalactic object bus“ [ORFA96, S. 48]. Er läßt Objekte transparent Anfragen (Requests) an lokale oder entfernte Objekte senden, indem er die Aufrufe abfängt, ein passendes Objekt, welches eine Implementation für den Aufruf anbietet, sucht, die Parameter weiterreicht, die entsprechende Methode aufruft und das Ergebnis zurückliefert. Der Client braucht sich somit nicht um den Aufenthaltsort des Servers, die Programmiersprache, in der der Server

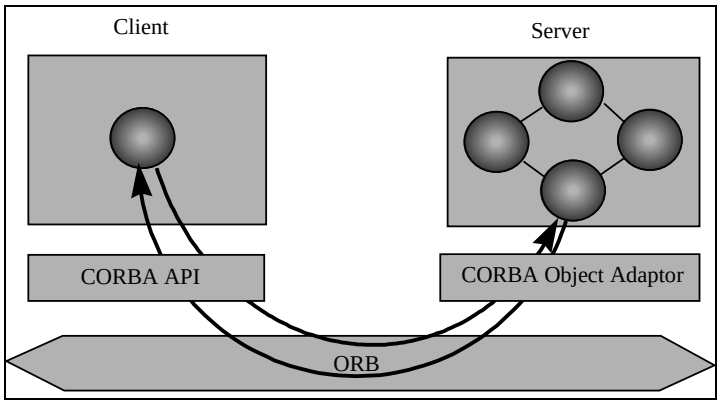


Abb. 6: Client-/Server-Kommunikation über ORB

implementiert ist, das Betriebssystem, auf dem das Serverobjekt läuft oder andere Systemaspekte, die nicht Bestandteil des Interfaces sind, zu kümmern (Abb. 6).

Der ORB unterstützt lokale und entfernte Methodenaufrufe, sowie die gegenseitige Entdeckung von Objekten zur Laufzeit. Im Folgenden werden die Besonderheiten eines ORB´s beschrieben:

- **Statischer und dynamischer Methodenaufruf:** Es besteht die Möglichkeit, Methodenaufrufe statisch zur Compilezeit oder dynamisch zur Laufzeit zu definieren. Bei statischen Methodenaufrufen hält der Client eine (entfernte) Referenz auf das Serverobjekt und ruft eine bestimmte Methode auf. Diese ist zur Compilezeit bekannt, woraus eine strenge Typprüfung möglich ist. Durch Abfrage des Interface und Implementation Repositories (Kapitel 3.2.2) können dynamisch zur Laufzeit existierende Objekte und deren Interfaces erkundet werden. Mit Hilfe der gewonnen Informationen können nun Methodenaufrufe an Objekte deren Klasse zur Compilezeit noch nicht bekannt war, abgesetzt werden. Dadurch entsteht eine hohe Flexibilität. Dieser dynamische Aufruf kann sowohl synchron als auch asynchron erfolgen, während der statische Aufruf ausschließlich synchron erfolgt.
- **High-Level Language Bindings:** Über den ORB können Methoden bei Server-Objekten aufgerufen werden, indem eine durch das CORBA-Binding unterstützte Hochsprache gewählt wird. Es spielt keine Rolle in welcher Sprache der Server implementiert ist, da das Interface von der Implementation getrennt ist und der Client nur das Interface kennen muß. CORBA bietet sprachneutrale Datentypen an, die es ermöglichen, Objekte über Sprach- und Betriebssystemgrenzen hinweg aufzurufen. Im Gegensatz dazu unterstützen andere Typen von Middleware sprachenspezifische API Bibliotheken und trennen nicht zwischen Spezifikation und Implementation.
- **Selbstbeschreibung des Systems:** CORBA unterstützt die Generierung der Metadaten zur Laufzeit, durch die die dem System bekannten Serverinterfaces immer aktuell beschrieben werden. Das Interface Repository enthält Informationen über die Methoden einschließlich ihrer Parameter, die die Serverobjekte anbieten. Die Clients benutzen Metadaten, um Services zur Laufzeit aufzurufen. Diese Metadaten werden automatisch durch einen IDL-Precompiler oder anderen Compilern, die IDL direkt aus einer OO-Sprache erzeugen können, generiert.
- **Lokale/entfernte Transparenz:** Ein ORB kann im Standalone-Betrieb laufen oder er kann mit jedem anderen CORBA 2.0 ORB verbunden sein (Single-Prozeß, Multi-Prozeß, Rechnernetzwerk, unterschiedliche Betriebssysteme). CORBA macht den Transport der Daten, das Auffinden des Servers und das Umwandeln der Daten in Byteströme (marshaling) transparent.
- **Sicherheit und Transaktionen:** Der ORB schließt Kontext-Informationen ein, um Sicherheit und Transaktionen zu behandeln.
- **Polymorphes Messaging:** Ein ORB ruft nicht eine beliebige Methode auf, sondern eine bestimmte Methode eines bestimmten Objektes. Da verschiedene Objekte für die gleiche Methode oder sogar dasselbe Klasseninterface unterschiedliche Implementationen haben dürfen, können gleiche Methodenaufrufe bei unterschiedlichen Objekten verschiedene Effekte haben.

3.2.2 Interface und Implementation Repository

Das **Interface Repository** ist eine dynamische Laufzeit-Datenbank, die alle Interface-Definitionen enthält. Es ist möglich, alle Beschreibungen von allen registrierten Interfaces, den Methoden, die sie unterstützen und den Parametern, die die Methoden benötigen, zu

erhalten und zu modifizieren. Diese Beschreibungen werden bei CORBA *Method Signatures* genannt. Dieser Gebrauch von Meta-Daten erlaubt allen Komponenten, die im ORB leben, selbstbeschreibende Interfaces zu haben. Daher ist CORBA ein sich selbstbeschreibendes System. Jedes Interface bekommt bei Eintragung in das Interface Repository eine eindeutig ID (Repository ID).

Das **Implementation Repository** ist ein Laufzeit Repository mit Informationen über die Klassen, die ein Server anbietet, die Objekte die instanziiert sind und ihre IDs. Es werden auch zusätzliche Informationen, die mit der Implementation des ORBs zusammenhängen, gespeichert (z.B. Verwaltungsdaten und Informationen für Sicherheitsaspekte).

3.2.3 Client/Server IDL Stubs

Die Client IDL Stubs bilden das statische Interface zu den Objekten. Diese Stubs definieren, wie Clients die korrespondierenden Server-Interfaces aufrufen. Aus der Sicht des Clients agiert der Stub wie ein lokaler Aufruf: Er ist ein lokaler Stellvertreter (Proxy) für ein entferntes Serverobjekt. Die Server IDL Stubs bieten das statische Interface zu jeder Methode an, die vom Server exportiert wird. Die Methoden der Serverobjekte werden in IDL spezifiziert. Die Client und Server Stubs werden durch den IDL-Compiler erzeugt. Jeder Client, der eine Methode eines Serverobjekts aufruft, muß für jedes Interface einen IDL-Stub erhalten. Der Stub enthält Code, um das Marshaling auszuführen. Er ist somit für die Umwandlung des Requests und der übergebenen Parameter in einen Bytecode, der über das Netzwerk übertragen werden kann, sowie die Rückcodierung des ankommenden Ergebnis-Bytecodes (Unmarshaling) in das für das Objekt verständliche Format, verantwortlich.

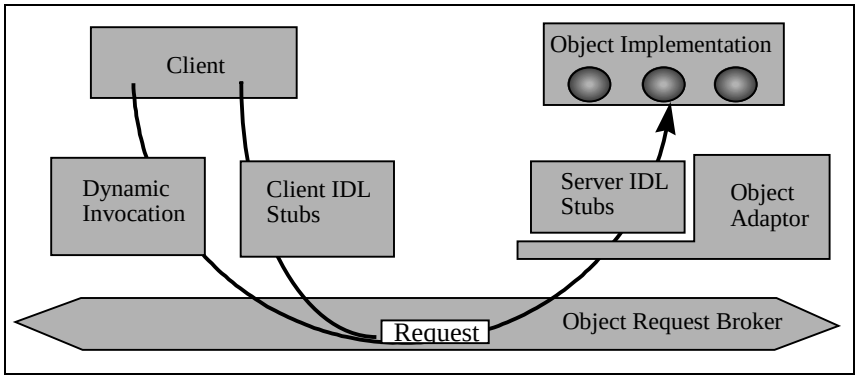


Abb. 7: Statische und dynamische Methodenaufrufe

3.2.4 Object Adaptor

Der Object Adaptor ist für die Registrierung der Server-Klassen im Implementation Repository und für die Erzeugung von Objekten dieser Klassen zuständig. Die Anzahl der Instanzen hängt von der Häufigkeit der eingehenden Anfragen und der Einstellung des Object Adaptors ab. Den neu instanziierten Objekten werden Objekt-Referenzen (eindeutigen IDs) zugeordnet.

Der Object Adaptor empfängt die Requests, die an den Server gesendet wurden, und reicht sie an den Server Stub weiter. Zu diesem Zeitpunkt ist der Object Adaptor dafür zuständig, das Serverobjekt, falls es inaktiv (d.h. persistent ausgelagert) ist, zu aktivieren (Abb. 8). Hierfür gibt es mehrere Möglichkeiten: Es kann ein neuer Prozeß oder ein neuer Thread in einem aktiven Prozeß aktiviert werden, oder es kann ein existierender Thread oder Prozeß wiederverwendet werden. Der Object Adaptor ist außerdem dafür zuständig, Objekte, die lange nicht mehr gebraucht wurden, auszulagern.

Server können unterschiedliche Arten von Object Adaptoren für verschiedene Bedürfnisse unterstützen. Um eine Ausuferung der Anzahl verschiedener Object Adaptoren zu verhindern, gibt es bei jeder CORBA-Implementation den Basic Object Adaptor (BOA). Dieser bietet eine grundlegende Funktionalität, die ebenfalls in CORBA spezifiziert ist.

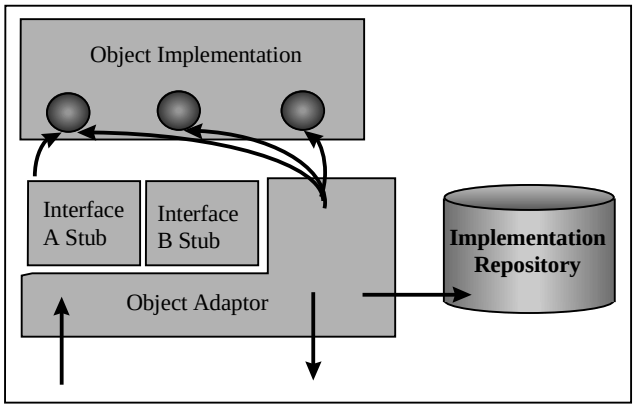


Abb. 8: Object Adaptor

3.2.5 DII/DSI

Mit Hilfe des **Dynamic Invocation Interface (DII)** können Methoden zur Laufzeit bestimmt und aufgerufen werden. CORBA definiert ein Standard API, um 1.) nach Metadaten zu suchen, die das Server-Interface definieren, 2.) Parameter zu generieren, 3.) den entfernten Aufruf auszuführen und 4.) das Ergebnis zu erhalten. Die Gesamtheit des entfernten Aufrufs wird Request genannt. In diesem sind das Serverobjekt, der gerufene Methodenname und die Parameter, nicht aber der Rückgabewert, enthalten.

Das **Dynamic Skeleton Interface (DSI)** bietet einen Laufzeit Binding Mechanismus für Serverobjekte an, deren IDL-Interface zur Kompilierzeit des Clients nicht bekannt sind. Das DSI entnimmt den Parametern des eingehenden Requests das Empfängerobjekt und die aufzurufende Methode und leitet die übrigen Parameter weiter. Das DSI ist das Server-Äquivalent des DII.

Mit Hilfe des DII/DSI sind sowohl synchrone wie auch asynchrone Methodenaufrufe möglich. Rückgabewerte eines asynchronen Requests müssen später selbständig eingesammelt werden.

3.3 Dienste

CORBA stellt drei Klassen von Diensten zur Verfügung: Object Services, Application Objects und Common Facilities (Abb. 9). Sie unterscheiden sich in der Anwendungsnähe und dem Detailierungsgrad der Spezifikation.

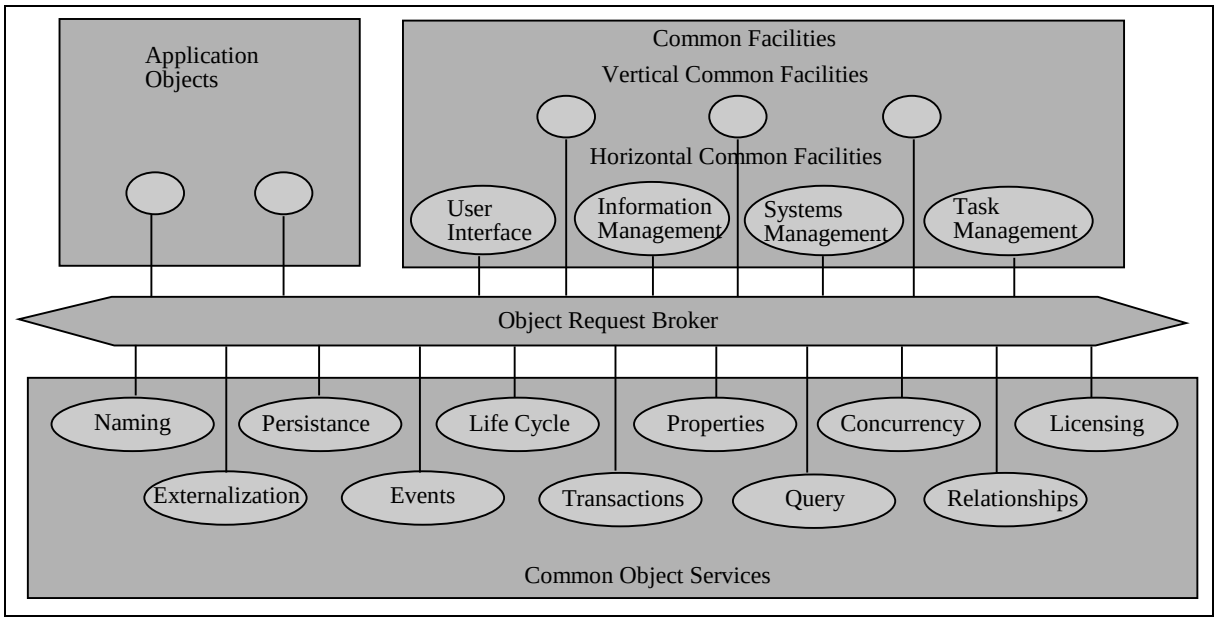


Abb. 9: Dienstklassen von CORBA

3.3.1 Object Services

Die Object Services sind eine Sammlung von systemnahen Diensten mit IDL-spezifiziertem Interface. Sie vergrößern und vervollständigen die Funktionalität des ORB. Die Standardisierung der Object Services erfolgt in fünf Schritten, in denen jeweils mehrere Services enthalten sind. Die einzelnen Schritte werden mit Common Object Service Specification (COSS) oder Request For Proposal (RFP) bezeichnet. COSS 1 und 2 wurden nacheinander 1994, COSS 3 und 4 zusammen Ende 1995 verabschiedet [ORFA96, S.60].

Da CORBA-Implementationen diese Dienste mit der in CORBA definierten Schnittstelle anbieten müssen, steht dem Entwickler ein breites Spektrum an benutzbarer Funktionalität zur Verfügung, auf die er sich bei seiner Arbeit verlassen kann.

Im folgenden werden einige der wichtigsten Object Services kurz erläutert:

- Der **Life Cycle Service** definiert Operationen zum Erzeugen, Kopieren, Bewegen und Löschen von Objekten auf dem Bus. Durch das explizite Bewegen eines Objektes von einem Rechner auf einen anderen ist u.a. eine erhöhte Performanz möglich.
- Der **Persistenz Service** bietet ein Interface an, das der persistenten Speicherung von Komponenten dient. Unterstützte Speicherarten sind dabei ODBMS, RDBMS und einfache Files. CORBA ist dadurch in der Lage, eine datenbankunabhängige Persistenz anzubieten.
- Der **Naming Service** dient zum Auffinden von Objekten auf dem Bus durch ihren Namen. Durch diesen Service können Objekte an einen hierarchischen Namenskontext gebunden werden. Dies ermöglicht die Auffindung von Objekten mit Hilfe natürlichsprachlicher Namen. Beispielsweise könnte ein Objekt, welches Informationen über die Schule X bereitstellt, an den Namen „/Schulen/Schulinfo-X“ aufgefunden werden. Der Naming Service wird ausführlicher in Kapitel 5.4.1 behandelt.
- Der **Event Service** erlaubt Objekten, sich für bestimmte Ereignisse an- und abzumelden. Der Event Service benutzt dazu einen Event Channel. Dieser erzeugt eine lose, asynchrone Koppelung zwischen Sender und Empfänger (siehe auch Kapitel 5.3.1).
- Der **Concurrency Control Service** bietet einen Lock-Manager an, der unterschiedliche Locks für Objekte, Transaktionen oder Threads vergeben kann. Dieser Service ist für

parallele Zugriffe auf Objekte notwendig. Angeboten werden unterschiedliche Read- und Write-Locks sowie Mechanismen zur Vermeidung von Deadlocks.
- Der **Transaction Service** bietet das 2-Phasen-Commit-Protokoll für flache und eingebettete Transaktionen an.
- Der **Relationship Service** dient dazu, dynamische Beziehungen zwischen sich gegenseitig nicht bekannten Komponenten zu erstellen. Dies gestattet eine gegenüber Inkonsistenzen sicherere Bindung als einfache Referenzen auf Objekte an (dangling References werden vermieden).
- Der **Externalization Service** stellt den Import/Export von Daten oder Objekten aus/in Datenströme bereit und dient so dem Verschicken und Sichern der Objekte.
- Der **Query Service** bietet Abfragemöglichkeiten für Objekte an. Es handelt sich dabei um eine Obermenge aus SQL3 und OQL (Object Query Language).
- Durch den **Licensing Service** werden Operationen zum Messen der Benutzung eines Objektes bereitgestellt, um sicherzustellen, daß ein fairer Ausgleich für die Benutzung der Objekte erfolgt. So könnte ein Objekt einen Informationsservice realisieren, der gegen Gebühr benutzbar wäre.
- Der **Properties Service** dient dazu, Bezeichnungen oder Werte (z.B. Titel, Datum) an ein beliebiges Objekt zu binden. Auf diese Art können Objekte kommentiert werden.

Die meisten Implementationen beschränken sich zum gegenwärtigen Zeitpunkt auf die Bereitstellung dieser Dienste, die in COSS 1, 2 und 4 beschrieben sind. Weitere Dienste sind Security, Time, Traders, Collections und Change Management.

3.3.2 Common Facilities

Während Objekt Services low-level Dienste bereitstellen, sind die Common Facilities Dienste, die in vielen Anwendungsgebieten nützlich sind. Im Gegensatz zu den Object Services sind die Common Facilities für ein CORBA-kompatibles System nicht zwingend erforderlich. Wenn sie allerdings vorhanden sind, müssen sie die von der OMG definierte Semantik einhalten [OMAG92, S. 61].

Common Facilities werden in zwei Kategorien unterteilt: Horizontale und Vertikale Common Facilities. Horizontale Common Facilities sind Dienste, die von den meisten Systemen unabhängig von ihrem Anwendungsgebiet benutzt werden, wie (Graphical) User Interface, Hilfesysteme, E-Mail und Systemmanagement. Die Vertical Common Facilities erfüllen dagegen branchenabhängige Aufgabe, die z.B. im Gesundheits-, Finanz- oder Telekommunikationswesen auftreten.

Anwendungsentwickler können die Common Facilities benutzen, aber auch Subklassen davon ableiten, um die Funktionalität anzureichern oder an ihre Applikation anzupassen.

3.3.3 Application Objects

Application Objects sind Objekte, die anwendungsspezifische Aufgaben erfüllen. Application Objects befinden sich auf dem gleichen semantischen Level wie die Common Facilities, Diese sind jedoch universeller einsetzbar, während Application Objects sehr spezielle Objekte sind. Daher werden die Application Objects auch nicht von der OMG standardisiert; die OMG überläßt dies anderen Standardisierungsgremien, stellt aber ein Diskussionsforum.

Am Beispiel Pausenplan wäre es vorstellbar, daß Klassen wie z.B. eine allgemein verwendbare Zeitklasse, die auch Zeiträume unterstützt, als Bestandteil der Application

Objects angeboten würden. Eine andere allgemeine Klasse in diesem Zusammenhang wäre eine Personenklasse, die es erlaubt, eine Person als Lehrer anzusehen.

4 Distributed Smalltalk

Distributed Smalltalk (DST) Version 5.5 ist eine CORBA 2.0 Implementation einschließlich der Mehrheit der CORBA-Services COSS 1 (Life Cycle, Naming, Event Notification) und COSS 2 (Transaction, Concurrency, Relationship - von DST Links genannt). DST erweitert das Image von VisualWorks Version 2.5.1 um die für die Verteilung notwendigen Klassen. Für Anwendungsentwickler, die bisher mit VisualWorks programmiert haben, ergibt sich der Vorteil der schon bekannten Programmierumgebung.

Distributed Smalltalk stellt den Object Request Broker (Kapitel 4.1) mit Remote-Testing und Remote-Debugger und IDL mit IDL-Generator und Interface Repository Browser (Kapitel 4.2) zur Verfügung. Diese Tools können alle über den zusätzlichen Menüpunkt „DST“ im Hauptfenster aufgerufen werden. In DST werden die aus Kapitel 3.2.3 bekannten Stubs Proxies genannt. Sie übernehmen Teile der ORB Aufgaben.

Für Entwickler beinhaltet DST bereits eine Reihe von Beispielanwendungen, die auf einer Klassenbibliothek aufbauen, welche die Verteilung kapselt. Diese Klassenbibliothek sieht eine Trennung in verteilte Semantik- und lokale Präsentationsobjekte vor (Presentation/Semantic-Split, Kapitel 4.3).

Um auf entfernte Objekte zugreifen zu können, müssen verschiedene Einstellungen in DST vorgenommen werden. Diese unterscheiden sich bei Server und Client (Kapitel 4.4). Die Einstellungen müssen vorgenommen werden, bevor Anwendungen verteilte Aufrufe absetzen. Kapitel 4.5 beschreibt, wie Anwendungen von DST aus gestartet werden.

4.1 Object Request Broker

Auf Basis der CORBA-Spezifikation ist der Object Request Broker in Distributed Smalltalk implementiert worden. Wenn ein Client eine Anfrage an das Serverobjekt stellen will, muß der Client-Proxy den Request von Smalltalk oder einer anderen Implementationssprache nach IDL übersetzen. Der Client-Proxy entscheidet, welcher ORB das Serverobjekt verwaltet. Um den Request zum anderen ORB zu senden, muß dieser erst in einen Bytecode übersetzt werden. Dieser Vorgang wird als Marshaling bezeichnet. Nachdem der Server-Proxy diesen RPC-Bytecode empfangen hat, muß er ein Unmarshaling durchführen und überprüfen, ob das entsprechende Interface im Interface-Repository definiert ist. Wenn dies der Fall ist, wird die IDL-Anfrage nach Smalltalk übersetzt und zum passenden lokalen Objekt weitergeleitet. Falls der Methodenaufruf ein Ergebnis zurückliefert, wird das Ergebnis von Smalltalk nach IDL übersetzt, gemarschalt und

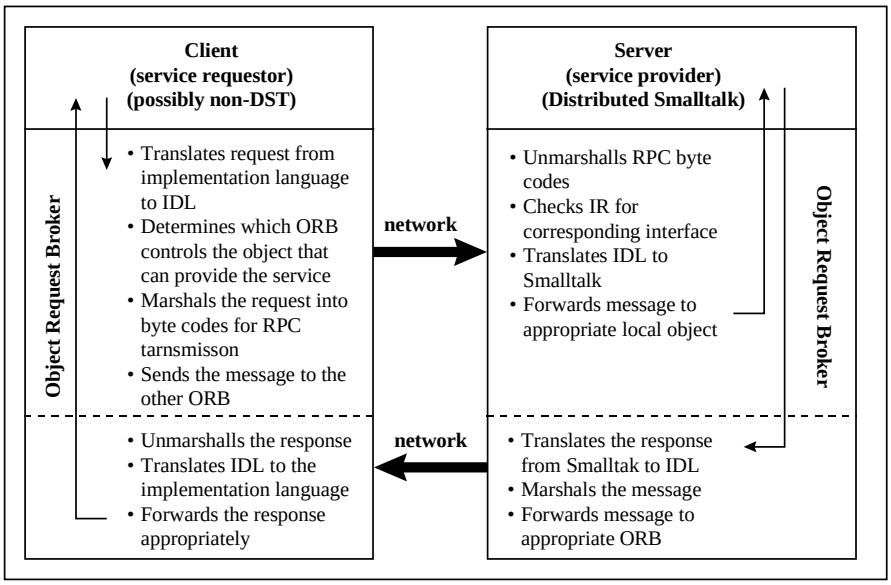


Abb. 10: Methodenaufruf über ORB [DSUG96, S. 39]

zum passenden ORB gesendet. Der Client-Proxy übernimmt anschließend das Unmarshaling, übersetzt die Antwort von IDL in die Implementationssprache und leitet die Antwort an den Client weiter.

Das ORB-Panel (Abb. 11) erhält man über das Menü „DST - Request Broker“. Mit dem Start-Button kann der ORB gestartet und durch nochmaliges Drücken gestoppt werden. Der Debugging-Switch wird benutzt, um auftretende Fehler, die sonst übergangen würden, anzuzeigen. Der Local RPC Testing Switch dient dazu, die Anwendung in einem Single-Image mit simulierten entfernten Objekten laufen zu lassen. Dies ist für einfacheres Debuging sinnvoll. Durch das Local RPC Testing wird die volle Marshaling- und Protokoll-Maschine für alle Messages benutzt, die potentiell verteilt sind. Nach Drücken des Monitor-Buttons erscheint ein Fenster, in dem die Kategorien Type (RPC Client oder Server), State (working, done, final, wait, init), Operation (gesendete Message), Target (abstrakte ClassID des Zielobjektes) und Activity angezeigt werden. Das Message Logging zeigt die RPC-Aktivitäten im System Transcript an. Durch Bewegung des Sliders wird der Umfang des Logging (0 bis 7) beeinflußt. Zur Benutzung wird empfohlen, nur Logging 0 bis 3 zu benutzen, da bei höhereren Einstellungen die Übersicht leicht verloren geht und die Geschwindigkeitseinbußen beträchtlich sind.

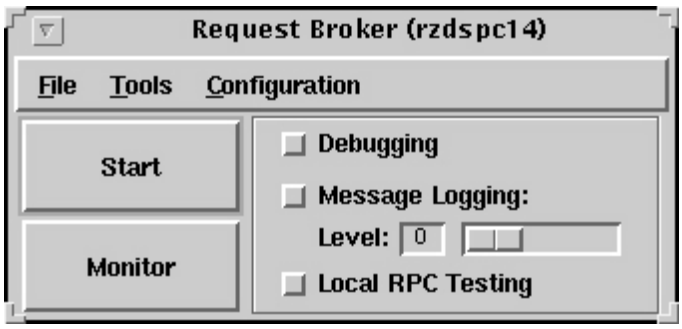


Abb. 11: ORB-Panel

4.2 IDL

Die Syntax von IDL wurde bereits in Kapitel 3.1 beschrieben. Im Folgenden genauer auf das Mapping von IDL nach Smalltalk und den IDL-Generator zur Erzeugung des IDL-Codes eingegangen.

In verteilten Anwendungen müssen alle Klassen, die als Parameter oder Rückgabewert von Nachrichten, die bereits Bestandteil eines IDL-Interface sind, ebenfalls ein der Klasse entsprechendes IDL-Interface bekommen.

Innerhalb eines IDL-Interfaces müssen alle Methoden aufgeführt sein, die entfernt aufgerufen werden sollen. Sie stehen damit dem Client zur Verfügung. Wird versucht, eine Methode entfernt aufzurufen, die nicht im IDL-Interface definiert ist, so tritt je nach Einstellung des ORB-Debuggings (s.o) ein Marshal-Error oder eine ORB-Exception auf.

4.2.1 IDL Mapping

Wie in den vergangenen Kapiteln erläutert, müssen alle Objekte, auf die entfernt zugegriffen werden soll, ein IDL-Interface besitzen. Jeder Methode, die entfernt aufgerufen wird, muß eine Operation in IDL entsprechen. Dieses Kapitel soll erläutern, wie die IDL-Operationsaufrufe in Smalltalkmethodenaufrufe umgesetzt werden.

In IDL sind Zeichen in Operationen erlaubt, die in Smalltalk nicht möglich sind. Umgekehrt sind in Smalltalk Nachrichtennamen erlaubt, die in IDL nicht zulässig sind. Daher gibt es einen Standard, wie nicht zulässige Zeichen nach Smalltalk umgesetzt werden. Beispielsweise wird die IDL-Operation *add_to_copy_map* umgesetzt zu *addToCopyMap*, da in Smalltalk keine Unterstriche erlaubt sind. Dies kann zu Fehlern führen, weil die eindeutige Zuordnung

der Methoden in Smalltalk gefährdet ist. Zum Beispiel werden die beiden IDL-Operationen *sample_op* und *sampleOp* beide zu *sampleOp* gemappt. Um dies zu verhindern, gibt es die #pragma selector Anweisung. Diese kann vor jeder Methode definiert werden um anzugeben, wie die IDL-Operation nach Smalltalk gemappt werden soll. Ein Beispiel:

#pragma selector eineMethodeMitZweiParametern eineMethode:zweiterParameter:
void eineMethodeMitZweiParametern (in short Parameter1, in short Parameter2);

Wird die IDL-Operation *eineMethodeMitZweiParametern* aufgerufen, wird sie in die Smalltalk Nachricht *eineMethode:zweiterParameter:* gemappt. *Parameter1* wird dabei hinter dem ersten Doppelpunkt, *Parameter2* hinter dem zweiten Doppelpunkt übergeben.
Sowohl die IDL als auch die Smalltalk Syntax erlaubt mehrere Input-Parameter. Aber in IDL können Operationen nicht nur einen Output-Parameter haben wie in Smalltalk, sondern die Parameter der Operation können durch die Attribute *inout* oder *out* zusätzliche Rückgabewerte erhalten. Um dies zu ermöglichen muß die Message *asCORBAParameter* an den potentiellen Rückgabewert (Objekt) gesendet werden. Dadurch wird ein Objekt erzeugt, das das CORBAParameter Protokoll unterstützt. Der Wert dieses Objekts kann durch die *value:* Methode geändert werden.

Das Mapping von IDL-Elementen nach Smalltalk geschieht wie folgt:

IDL-Element	Smalltalk
Objekt Referenz	Smalltalk Objekt, das ein CORBA-Objekt repräsentiert
Interface	Smalltalk Objekt, das auf alle Attribute und Operationen des Interfaces und der der geerbten Interfaces reagieren kann
Operation	Smalltalk Message
Attribut	Smalltalk Message
Konstante	Smalltalk-Objekt, das im CORBA-Konstanten-Dictionär enthalten ist
Integer Typ	Smalltalk-Objekt, das der Integer Klasse entspricht
Floating-Point Typ	Smalltalk-Objekt, das der Float Klasse entspricht
Boolean Typ	True- oder False-Objekt
Enumerationstyp	Smalltalk-Objekt, das dem CORBA-Enum Protokoll entspricht
Any Typ	Smalltalk-Objekt, das in einen IDL-Typen gemappt werden kann
Structure Typ	Smalltalk-Objekt, das der Dictionary-Klasse entspricht
Union Typ	Smalltalk-Objekt, das zu den möglichen Typen des IDL-Union gemappt wird oder dem CORBA-Union Protokoll entspricht
Sequence Typ	Smalltalk-Objekt, das der Klasse OrderedCollection entspricht
String Typ	Smalltalk-Objekt, das der Klasse String entspricht
Array Typ	Smalltalk-Objekt, das der Klasse Array entspricht
ExceptionTyp	Smalltalk-Objekt, das der Klasse Dictionary entspricht

4.2.2 IDL-Generator

Der IDL-Generator dient dazu, aus vorhandenen Smalltalk-Klassen IDL-Interfaces zu generieren. Um den IDL-Generator zu starten muß im Menü „DST - IDL Generator“ gewählt werden (Abb. 12). Bei der Generierung des IDL-Codes sollten folgende Schritte eingehalten werden:

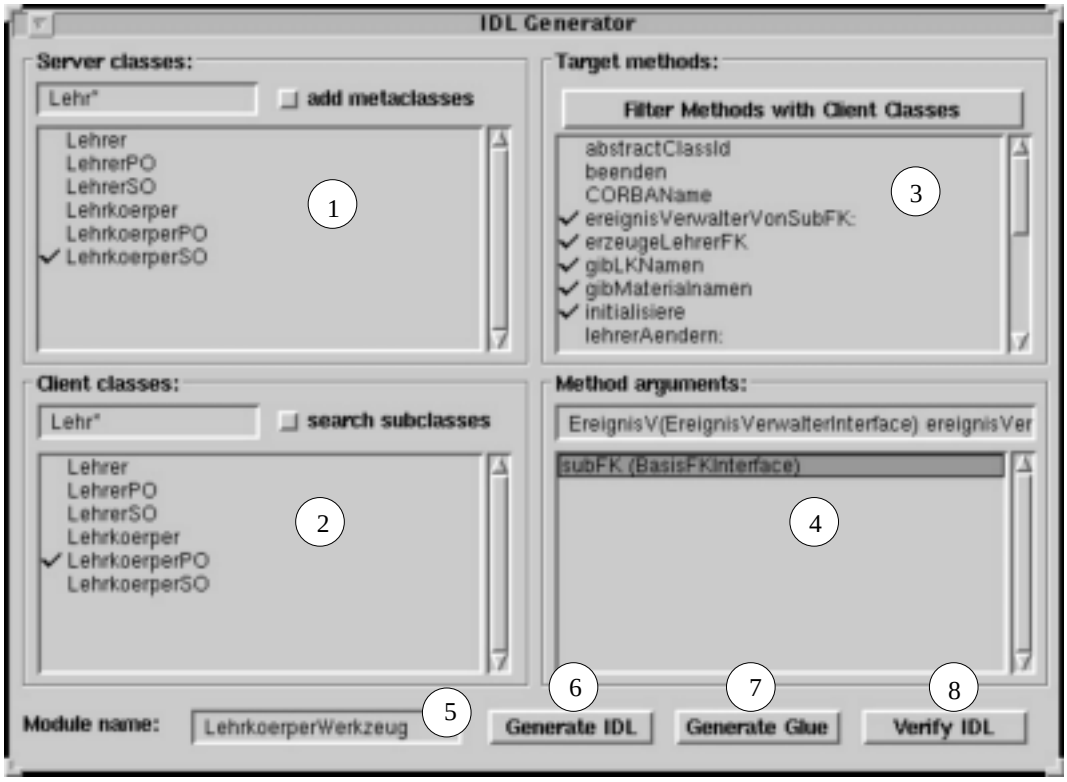


Abb. 12: IDL-Generator

1. Auswählen der Serverklassen, die in das aktuelle IDL-Modul aufgenommen werden sollen. In der Abbildung soll das IDL-Interface für LehrkoerperSO erstellt werden.
2. Auswählen der Clientklassen, die die in 1. markierten Serverklassen benutzen. Dies erleichtert das Filtern der Methoden der Serverklassen (Schritt 3). Die einzige Klasse, die die LehrkoerperSO benutzt, ist LehrkoerperPO.
3. Auswählen der Methoden, die tatsächlich vom IDL-Interface exportiert werden sollen. Durch Drücken des Filter-Method-Buttons werden die Methoden markiert, die Objekte der in Schritt 2 ausgewählten Clientklassen den Objekten der in Schritt 1 ausgewählten Serverklassen senden.
4. Spezifizieren der Argumenttypen für die aktuell ausgewählte Methode. Die kleine Box der Argumenten-Area zeigt den Returntyp an, die große Listbox listet die Argumente und Typen auf. Der Defaultwert für Parameter und Returnwert ist Smalltalk-Objekt. Dieser Typ sollte allerdings möglichst nicht benutzt werden, da er in der Regel nicht speziell genug ist und eine potentielle, schwer zu lokalisierende Fehlerquelle darstellt. Über den Operate-Button der Maus erhält man eine Liste von IDL-Datentypen.
5. Angeben des Modulnamens. Wird ein schon verwendeter Modulname angegeben, so wird dieses Modul beim Compilevorgang (Schritt 6) überschrieben.
6. Drücken des Generate-IDL-Buttons, um das Interface zu generieren. Wenn der ORB läuft (das wird dringend empfohlen) wird das IDL-Modul im Interface Repository auftauchen. Dies sollte genauestens überprüft werden, da hier gelegentlich Fehler eingebaut werden. Vor allem müssen die Pragmas überprüft werden. Falls man als einen Datentyp Sequence von einem Basistyp ausgewählt hat, so meldet der IDL-Generator immer einen Fehler! Dies kann man umgehen, indem man bei der automatischen Generierung auf Sequences verzichtet und diese manuell nachträgt. Dabei ist zu beachten, daß am Anfang des Moduls ein typedef für die Sequence definiert wird, beispielsweise *typedef sequence<string> liste;*

Bei den Operationen muß dann immer *liste* statt *sequence<string>* benutzt werden. Das fertige IDL-Modul wird mittels accept in das Interface Repository übernommen.

7. Drücken des Generate-Glue-Buttons, um die Smalltalk Repository Methoden zu generieren. Generiert wird dabei ein Protokoll *Repository*, das für jede Server-Klasse des Moduls die passende *abstractClassID* und *CORBAName* Methode enthält. Die *abstractClassID*, die für jede Klasse eindeutig ist, wird von DST benötigt, um mit Hilfe sogenannter Factories Instanzen zu erzeugen, wenn entfernte Clients dies fordern; der *CORBAName* ist die Beziehung zwischen Smalltalk-Klasse und IDL-Interface und setzt sich aus Modul- und Interfacenamen zusammen.
8. Drücken des Verify-IDL-Buttons, um eine Konsistenz-Prüfung durchzuführen. Dabei wird die Konsistenz von Typen, Operationen und Interfaces zwischen Smalltalk und IDL überprüft. Entdeckte Fehler werden im System Transcript ausgegeben. Leider werden nicht alle Fehler gefunden. Es kann passieren, das bei der nächsten Initialisierung des ORB oder des Interface Repositories weitere Fehler gefunden werden. Das sind insbesondere solche, die sich aus Namensgleichheit von geerbten und neuspezifizierten Methoden ergeben. Vererbte Methoden sollten im Interface der erbenden Klasse nicht erneut aufgeführt werden, da sonst die Fehlermeldung „ambigious Definitions“ auftritt.

4.2.3 Interface Repository Browser

Der IR-Browser erlaubt den Zugriff auf den Inhalt von lokalen und verteilten Repositories mit Concurrency Control. Um den IR-Browser zu öffnen, kann entweder der entsprechende Button gedrückt oder im Menü „DST - Interface Repository“ gewählt werden. Es erscheint ein Fenster, in dem je nach Wunsch eine grafische oder textuelle Auflistung aller IDL-Module angezeigt wird. Mit einem Klick auf ein IDL-Modul wird dieses geöffnet und es erscheint ein Fenster, in dem die Interfaces dieses Moduls aufgelistet sind. Durch Anwahl des Menüs „Edit - Definition“ erscheint ein Fenster mit dem vollständigen IDL-Code (Abb. 13).

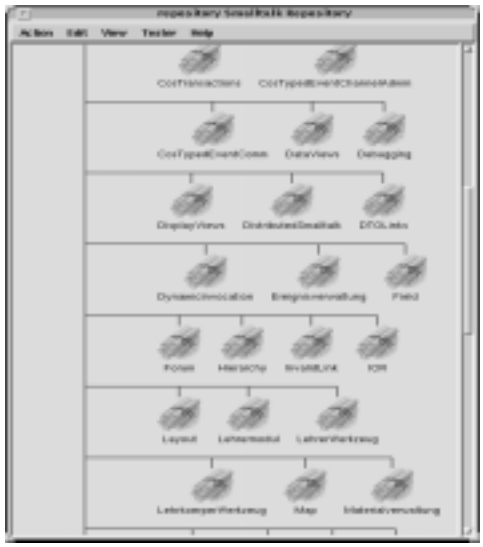


Abb. 13: IR-Browser

4.3 Presentation/Semantic-Split

Nachdem bisher die Funktionsweise von CORBA und die Verteilung von Objekten erklärt wurde, stellt sich die Frage, welche Objekte verteilt sein sollen und welche nicht. Allgemein gilt, daß verteilte Objekte die Performance der Anwendung senken und lokale Objekte die Performance erhöhen. Mit Blick auf die Geschwindigkeit sollten also möglichst wenig Objeke verteilt sein.

In Distributed Smalltalk wurde als Design-Muster der Presentation/Semantic-Split [DSUG96, S. 34f.] zu Grunde gelegt. Prinzipiell sind Presentation-Objekte lokal und für die Darstellung der Daten zuständig, während Semantic-Objekte verteilt oder lokal sein können und für die Datenspeicherung zuständig sind. Die Begriffe Lokal und Verteilt vertreten dabei immer den Blickwinkel des Benutzers. Semantic-Objekte sind verantwortlich für die Wartung der Attribute, die von allen Präsentationen gemeinsam benutzt werden. Presentation- und

Semantic-Objekte stehen in einer n:1 Beziehung: Jedes Semantic-Objekt kann mehrere Presentation-Objekte haben. Der Split erlaubt vielen Benutzern, simultane Sichten auf die Objekte zu haben. Die Sichten können dabei in der Erscheinung unterschiedlich sein, sie spiegeln aber dieselben zugrundeliegenden Informationen des Semantic-Objekts wieder.

Es ist nicht festgelegt, welche Attribute typische Semantic- oder Presentation-Attribute sind. Ziel des Splits ist vor allem, die bestmögliche Performance und Wiederverwendung zu erhalten. Presentationsobjekte sollten daher die persönlichen Aktivitäten des Benutzers regeln und für Aufgaben mit hohem Verkehrsaufwand (Bildschirmaufbau, Benutzereingaben, etc.) verantwortlich sein. In Kapitel 5 wird ein aus softwaretechnischen Gesichtspunkten motivierter Vorschlag erörtert.

Die meisten Klassen, die für verteilte Anwendungen benötigt werden, sollten entweder von der Klasse *DSTSemantic* bzw. deren Unterklasse *DSTapplicationObject* (Semantic-Objekte) oder von der Klasse *DSTPresenter* (Presentation-Objekte) erben. Diese Klassen bieten die Basis für eine effiziente Zusammenarbeit zwischen lokalen und entfernten Objekten. Dazu benutzen sie die CORBAServices Lifecycle, Link, Event Notification. Wenn existierende Anwendungen nach DST portiert werden sollen ist es besser, von den Wrapperklassen *DSTMediaDM* und *DSTMediaAM* zu erben (Abb. 14). Bei der Namensgebung der Klassen sollten die Presentation-Klassen den gleichen Namen wie die dazugehörigen Semantic-Klassen haben; nur die Endung sollte PO oder SO sein.

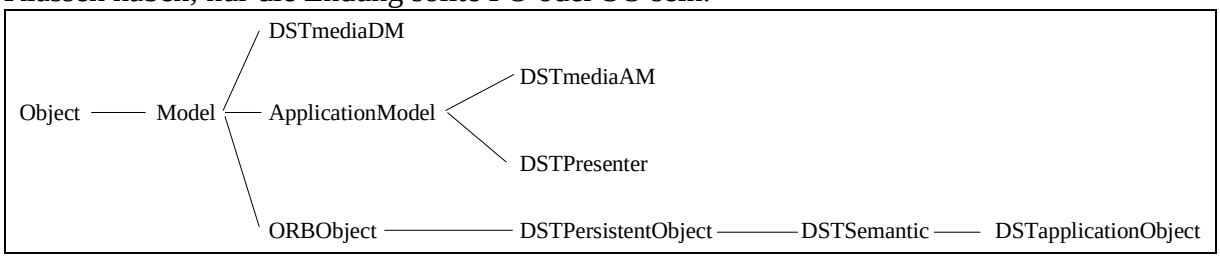


Abb. 14: Ausschnitt der DST-Klassenhierarchie

Im Rahmen dieser Arbeit werden die neuen Klassen BasisFK (erbt von DSTapplicationObject) und BasisIAK (erbt von DSTPresenter) zur Verfügung gestellt (siehe Kapitel 5.2.1), so daß Entwickler sich um die von DST benutzten Konzepte nicht zu kümmern brauchen.

4.4 Einstellungen für den entfernten Zugriff

Um auf entfernte Objekte zugreifen zu können, müssen in Distributed Smalltalk einige System-Einstellungen vorgenommen werden, die den lokalen ORBs erlaubt, mit entfernten ORBs zu kommunizieren. Es gibt ein zentrales, ausgezeichnetes Image (Server für Naming Service und Repository) und beliebig viele lokale Client-Images. Die beiden Image-Typen müssen unterschiedlich konfiguriert werden. Das Eingabefenster für die Einstellungen wird mit dem Menüpunkt „DST - Settings“ geöffnet. (Hinweis: In jedem Teil des Notebooks, in dem Veränderungen vorgenommen werden, muß zur Bestätigung der Accept-Button gedrückt werden!)

Die Server-Konfiguration:

- **Adapter ID settings**: Auswählen von ‘Configured To’ unter Beibehaltung der angegebenden Adaptor id.

- **IIOP Transport settings**: Auswählen von ‘Configured To’ unter Beibehaltung des angegebenen Ports. Wenn das Image mit einem ORB auf einer Windows 3.1 Maschine kommunizieren soll, muß das IIOP Write Increment modifiziert werden.
- **NCS Transport settings**: Auswählen von ‘Configured To’ unter Beibehaltung des angegebenen Ports. Wenn das Image mit einem ORB auf einer Windows 3.1 Maschine kommunizieren soll, muß die Fragment Size modifiziert werden.
- **TI Locator, Naming Service, Repository und Security settings**: Diese Einstellungen müssen alle auf ‘local’ gestellt werden.

Die Client-Konfiguration:
- **Adapter ID settings**: Auswählen von ‘Allocated By System’.
- **IIOP Transport settings**: Auswählen von ‘Dynamically Allocated’. Die Änderung des IIOP Write Increment erfolgt wie bei der Server-Konfiguration.
- **NCS Transport settings**: Auswählen von ‘Dynamically Allocated’. Die Änderung des IIOP Write Increment erfolgt wie bei der Server-Konfiguration.
- **TI Locator**: Auswählen von ‘Configured To’ und Eintragen des Hostnamens des Rechners auf dem das Server-Image läuft.
- **Naming Service**: Auswählen von ‘Adaptor Id’ unter Beibehaltung der voreingestellten Adaptor Id.
- **Repository setting**: Auswählen von ‘Adapter Id’ unter Beibehaltung der voreingestellten Adaptor Id.
- **Security settings**: Diese Einstellung muß auf ‘local’ gesetzt werden.

Nach den Veränderungen der Konfiguration in den Images muß noch sichergestellt werden, daß in den ORB-Panels im Menü „Configuration“ die Checkboxes von IIOP und NCS transport selektiert sind. Anschließend können die ORBs gestartet werden. Im folgenden Kapitel wird beschreiben, wie verteilte Anwendungen gestartet werden können.

4.5 Erzeugen und Starten von Applikationen

Die Sandbox ist ein Bereich, in dem man mit den Beispiel-Applikationen Erfahrungen mit dem Umgang mit DST sammeln kann. Hier können lokale und verteilte Applikationen erzeugt, gestartet und gelöscht werden. Geöffnet wird die Sandbox durch Klicken auf das Sandbox-Item oder durch wählen von „DST - Sample Applications“ im Menü. Innerhalb der Sandbox befindet sich das Office des aktuellen Benutzers. Im Office können durch Wahl von „Object - Create Examples“ die Beispiel-Applikationen und eigene Anwendungen erzeugt werden. Diese können lokal, mit Local RPC Testing oder verteilt gestartet werden.

Ein Office (Abb. 15) stellt eine Sammlung von Anwendungen dar, die ihrerseits wieder hierarchisch in Ordner verteilt sein können. Jedem Office ist ein Benutzer zugeordnet. Es empfiehlt sich daher, die personenbezogenen Applikationen (wie das mit der voreingestellten Beschriftung ‘Guest’ versehenen Kommunikationstool) mit dem entsprechenden Namen zu versehen (z.B. ‘Longo´s Office’). Dazu dient der Menüpunkt „Action - Retitle...“. Es stehen weitere Anwendungen wie ein Papierkorb und ein Waisenhaus (Orphans) zur Verfügung, in dem vorher gelöschte Objekte enthalten sind, auf die

Abb. 15: Ein Beispieloffice

noch Verweise bestehen. Der Anschaulichkeit wegen haben viele Anwendungen den Namen des Materials, das sie bearbeiten, so daß eine intuitive Bedienung möglich ist.

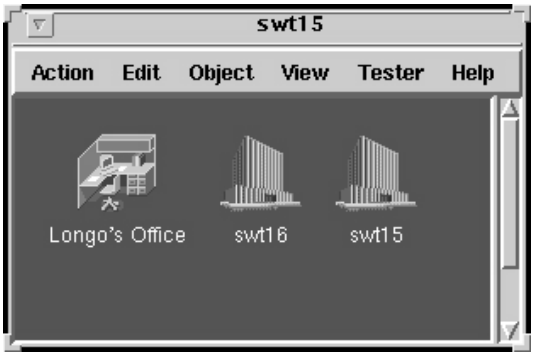


Abb. 16: Ein Building (Host: SWT15)

Ein Building enthält immer alle Offices, zu denen von dem Rechner des Benutzers aus zugegriffen werden kann (Abb. 16). Damit von fremden Buildings aus auf das eigene Office zugegriffen werden kann, muß dieses erst freigegeben werden. Das geschieht über „Action - Add to Building“. Um auf Objekte eines fremden Buildings zugreifen zu können, muß im Office aus dem Menü „Action - Open Building“ gewählt werden. Bevor auf ein entferntes Building zugegriffen werden kann, müssen unter „DST - Settings“ im Hauptfenster einige Einstellungen vorgenommen werden (siehe Kapitel 4.4). Sind diese bereits korrekt eingestellt, so kann im Menü des Buildings durch Anwählen von „Action - Add Remote“ der entfernte Rechner angegeben werden. Dabei ist zu beachten, daß der ORB auf beiden Rechnern laufen muß. Durch Klicken auf das entfernte Building erscheinen die entfernten Offices des angegebenen Rechners. Im Demo-Ordner dieser Offices können Anwendungen gestartet werden.

5 Implementation des Pausenplaners

Das Handbuch von DST empfiehlt zur Entwicklung von verteilten Anwendungen zuerst die lokale Applikation zu erstellen und zu testen. Erst wenn diese fehlerfrei läuft soll sie um die Verteilung erweitert werden [DSUG96, S. 49f.].

Die Beispielapplikationen von DST besitzen alle den in Kapitel 4.3 beschriebenen Presentation/Semantic-Split (P/S-Split). Der Pausenplaner sollte möglichst eng an dieses Designkonzept angelehnt sein. Um eine klarere Aufgabenteilung zu erreichen, haben wir uns entschlossen, einige Konzepte aus der WAM-Metapher (Werkzeug/Aspekt/Material) [KILB94, S. 23ff.] mit dem P/S-Split zu kombinieren.

Im Gegensatz zu den sehr einfachen Beispielen von DST haben wir es für sinnvoll erachtet, die anwendungsspezifischen Daten (Materialien, Kapitel 5.1) strikt von den Semantikobjekten zu trennen, da die Materialien kein Wissen über die sie darstellenden Komponenten haben sollten.

Es besteht eine weitgehende Äquivalenz zwischen Presentationsobjekten (PO) und Interaktionskomponenten (IAK) bzw. zwischen Semantikobjekten (SO) und Funktionskomponenten (FK). Im Sinne von WAM stellt der P/S-Split die Trennung des Werkzeugs in IAK und FK dar (Kapitel 5.2). Im folgenden werden FK und SO bzw. IAK und PO synonym genutzt. Dies gilt insbesondere auch für den Programmcode: FK und IAK betonen die Werkzeugfunktionalität, PO und SO die verteilten Aspekte.

Die Verbindungen von FK und IAK bestehen wie in WAM üblich: Zwischen FK und IAK besteht eine Beobachterbeziehung (Observerpattern) [GAMM95, S. 293ff.], zwischen IAK und FK eine gewöhnliche Benutztbeziehung. Wir haben uns entschlossen, für die Implementation des Beobachtermechanismus weder das Smalltalk Standardprotokoll (mit change/update) bzw. das erweiterte VisualWorks-Protokoll [GAZH96, S.33ff.], noch den in [GAZH96, S. 36ff.] implementierten Beobachtermechanismus mit Events und Event-Stubs zu benutzen, da DST den CORBA Event-Service anbietet. Dieser wird von der Ereignisverwaltung (Kapitel 5.3) benutzt. Eine grundsätzliche Diskussion verschiedener Beobachtermuster ist in [ROOC96] nachzulesen.

Da mehrere Werkzeuge auf denselben Materialien arbeiten können, sie aber gegenseitig unabhängig sein sollen, mußte ein Konzept geschaffen werden, das den Austausch der Materialien übernimmt. Der dafür zuständige Materialverwalter wird in Kapitel 5.4 beschrieben.

Für alle Materialien und Werkzeuge sowie die Material- und Ereignisverwaltung wurden IDL-Interfaces erstellt. Daher besitzt jede Klasse ein Protokoll „Repository“, in dem die Methoden *abstractClassId* und *CORBAName* enthalten sind (siehe Kapitel 4.2.2).

5.1 Materialien

Das Material läßt sich im Diagramm wie in Abb. 17 darstellen:

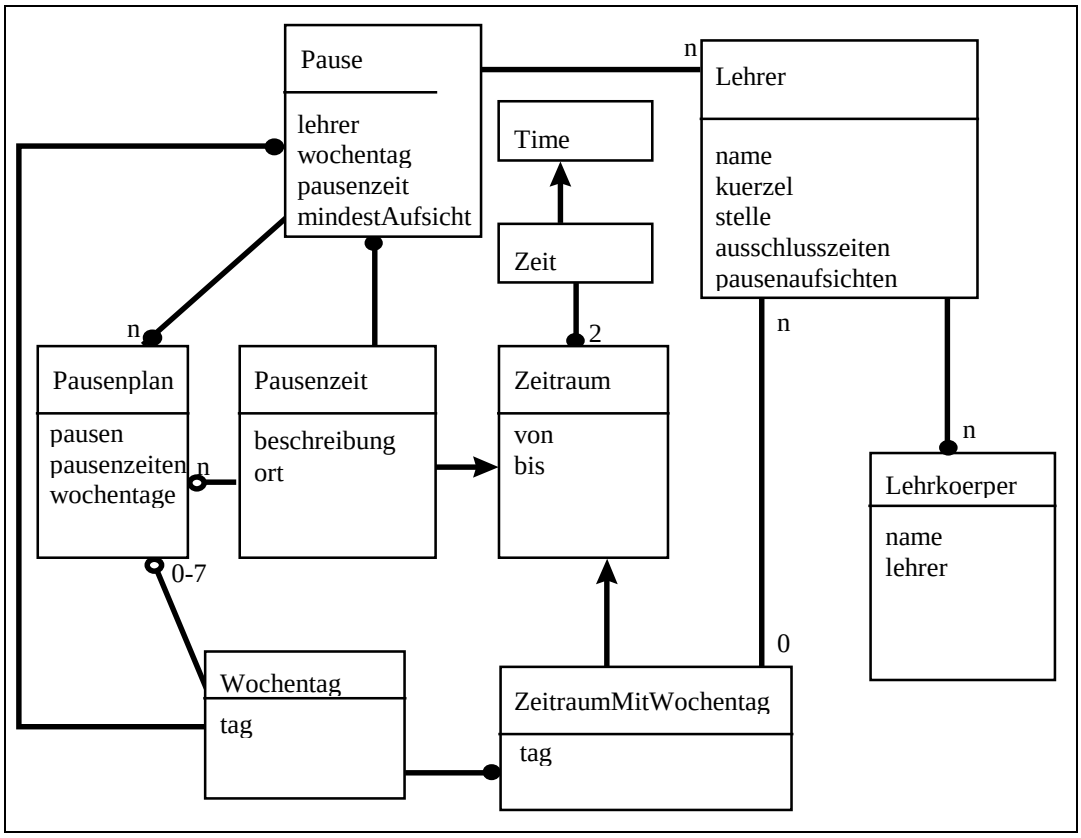


Abb. 17: Materialklassen

Die Instanzen der Klasse **Lehrer** repräsentieren in dieser Anwendung die Pausenaufsicht führenden Personen einer Schule. Es wird also von weiteren Eigenschaften realer Lehrer (Fächer, Anschrift, etc.) abstrahiert. In diesem Anwendungsfall interessieren bei einem Lehrer der Name, ein eindeutiges Kürzel, der Beschäftigungsgrad (Stelle), die zu beaufsichtigenden Pausen und die Ausschlußzeiten.

Lehrer werden zu einem **Lehrkörper** aggregiert, der durch einen Namen bezeichnet wird. Im Lehrkörper können Lehrer nach Kürzel oder vollständigem Namen gesucht werden.

Ein **Zeitraum** besteht aus zwei Zeitpunkten (*von* und *bis*) und ist in der Lage, Überschneidungen mit anderen Zeiträumen zu erkennen. Jeder Zeitpunkt ist ein Objekt der Klasse **Zeit**, die im wesentlichen neue Nachrichten zur einfacheren Erzeugung versteht. Hierbei und bei der Konvertierung in eine Stringrepräsentation wird auf Sekunden verzichtet.

Die Klasse **Wochentag** repräsentiert einen bestimmten Tag in der Woche und geht mit Tagesangaben in der Langform (Montag, Dienstag, usw.) oder in der Kurzform (Mo, Di, usw.) um. Ein **ZeitraumMitWochentag** kann entsprechend auch Überschneidungen oder Gleichheit mit Zeitraum oder ZeitraumMitWochentag überprüfen.

Eine **Pausenzeit** gibt unabhängig vom Tag an, in welchem Zeitraum und an welchem Ort eine Pause stattfindet. Zusätzlich kann jeder Pausenzeit eine Beschreibung zugeordnet werden, wie „erste große Pause“. Es wird angenommen, daß an jedem Tag, an dem überhaupt Pausen vorhanden sind, die Pausen immer zur gleichen Zeit auftreten.

Im Vergleich dazu stehen Instanzen der Klasse **Pause** für eine ganz bestimmte reale Pause, in einem bestimmten Zeitraum, an einem bestimmten Ort und an einem bestimmten Wochentag. Für eine Pause kann angegeben werden, wie viele Lehrer mindestens für eine Aufsicht eingeteilt sein müssen.

In einem **Pausenplan** werden die gültigen Pausenzeiten und die Wochentage, an denen eine Pausenaufsicht notwendig ist, gehalten. Daraus ergeben sich die realen, zu beaufsichtigenden Pausen.

Zusätzlich zu den oben aufgeführten Materialien gibt es die Klasse **Schulinfo**, in der schulspezifische Informationen gehalten werden. Dies sind die Anzahl der zu beaufsichtigenden Pausen und die Summe der Beschäftigungsgrade der Lehrer. Daraus kann ein Objekt dieser Klasse die Sollpausenaufsichten für eine Volltagskraft berechnen.

5.2 Werkzeuge

DST bietet mit dem P/S-Split bereits einen Mechanismus an, um verteilte Anwendungen zu schreiben. Wie bereits in Kapitel 5 erläutert, wurden Materialien und Werkzeuge voneinander getrennt. Im P/S-Modell kennen sich PO und SO wechselseitig. Um dieses zu verbergen und statt dessen das Beobachtermuster bereitzustellen und um einfache Protokolle für die Handhabung mit Werkzeugen zur Verfügung zu stellen, wurden die Klassen BasisFK und BasisIAK (Kapitel 5.2.1) entwickelt. Diese dienen als Grundlage für weitere Anwendungen.

Das in Kapitel 5.1 beschriebene Material kann mit zwei unabhängig voneinander existierenden Werkzeugen bearbeitet werden: Dem Lehrkörpertool (Kapitel 5.2.2) und dem Pausenplaner (Kapitel 5.2.3). Diese bilden aus der Sicht der Office-Fenster mit den ihnen zugrunde liegenden Materialien eine Einheit (Kapitel 4.5). Das bedeutet, daß ein Werkzeugexemplar immer für genau ein Materialexemplar zuständig ist und dieses automatisch erzeugt.

Neue Materialien werden erzeugt, indem im zusammenfassenden Ordner aus dem Menüpunkt „Object“ der Eintrag „Create a New“ gewählt wird. Aus der erscheinenden Liste wird das gewünschte Material ausgewählt. Daraufhin ist noch die Eingabe eines Namens für das Material nötig, welcher zu dem daraufhin erzeugten Icon angezeigt wird. Intern wird gleichzeitig mit dem Icon das zum ausgewählten Material passende Semantikobjekt erzeugt. Dieses wiederum erzeugt das Material. Da das Office bzw. die darin liegenden Ordner und Objekte mit dem Image gespeichert werden, leben die Semantikobjekte solange, bis sie über den Menüpunkt „Edit - Throw away“ explizit gelöscht werden. Mit dieser Technik ist eine automatische, eingeschränkte Persistenz gegeben.

Ein Semantikobjekt stellt in DST eine Factory dar. Ein FactoryFinder ist dafür zuständig, die Factories zu finden, die im System vorhanden sind. Dazu sucht er in allen Klassen nach der Methode *productName*. Damit Materialien bzw. die sie erzeugenden Semantikobjekte in der oben erwähnten Liste aufgeführt werden, müssen sie diese Nachricht implementieren. p*roductName* liefert einen String als Symbol, der in dieser Liste aufgeführt wird. Die Methode *productName* der Klasse LehrkörperSO liefert #’Lehrkoerper’, die der Klasse PausenplanSO #’Pausenplan’.

5.2.1 Oberklassen BasisFK und BasisIAK

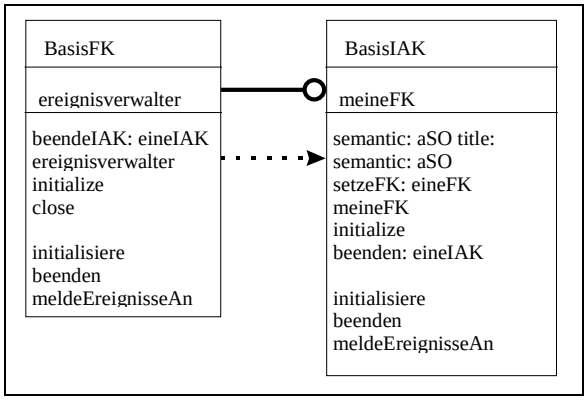


Abb. 18: Werkzeug Basisklassen

Ein Werkzeug besteht aus einer FK und einer oder mehreren IAKs (siehe Kapitel 5). Da diese immer ein bestimmtes Grundverhalten zeigen, gibt es Basisklassen BasisFK und BasisIAK, die dieses Gerüst zur Verfügung stellen.

Die BasisFK (Abb. 18) stellt Mechanismen zur Initialisierung, zur Bereitstellung von Ereignissen (zur Ereignisverwaltung siehe Kapitel 5.3) und zum Aufräumen beim Beenden bereit. Wird eine neue FK durch „Object - Create a New...“ erzeugt, so wird zunächst *initialisiere* und anschließend *meldeEreignisseAn* aufgerufen. Abgeleitete Klassen sollten diese Methoden für eigene Zwecke überladen. Von *meldeEreignisseAn* muß zuerst die gleichnamige Methode der Oberklasse aufgerufen werden. Der komplette Initialisierungsvorgang ist dem Interaktionsdiagramm in Abb. 19 zu entnehmen.

Der Beenden-Vorgang der FK wird bei Auswahl von „Edit - Throw Away“ eingeleitet. Die abgeleitete Klasse sollte ihre Aufräumarbeiten in der Methode *beenden* durchführen. Diese Aufräumarbeiten werden vor allem in der Abmeldung von der Materialverwaltung und der Bereinigung von Materialien bestehen.

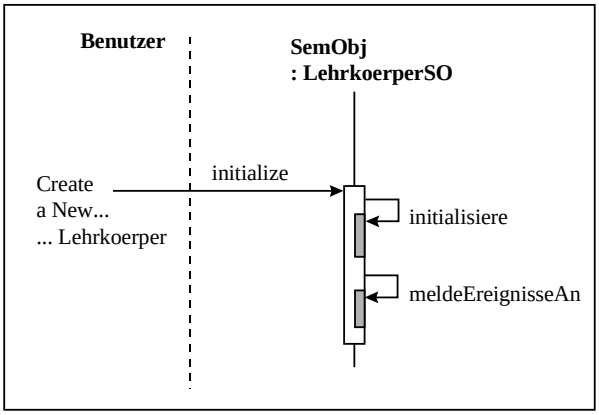


Abb. 19: Erzeugung einer LehrkoerperSO

Ähnlich der BasisFK stellt die BasisIAK (Abb. 18) Protokolle zur Erzeugung, zum Empfang von Ereignissen sowie zum Beenden zur Verfügung. Durch Doppelklick auf das Icon eines Semantikobjektes (einer FK oder eines Materials) wird intern bereits von DST die Bekanntmachung des Semantikobjektes an das Präsentationsobjekt bereitgestellt. Durch überladen der Methode *semantic:title:* erfährt die BasisIAK vom Setzen der zugehörigen BasisFK und ist in der Lage, die notwendigen, erst jetzt möglichen Initialisierungen vorzunehmen. Von der BasisIAK abgeleitete Klassen sollten diese Initialisierungsaufgaben in der Methode *initialisiere* ausführen und sich für Ereignisse in *meldeFuerEreignisseAn* anmelden. Beide Nachrichten werden nach erfolgter interner Initialisierung gesendet (Interaktionsdiagramm Abb. 20). Beim Anmelden für Ereignisse muß darauf geachtet werden, daß die gleichnamige Routine der BasisIAK aufgerufen wird.

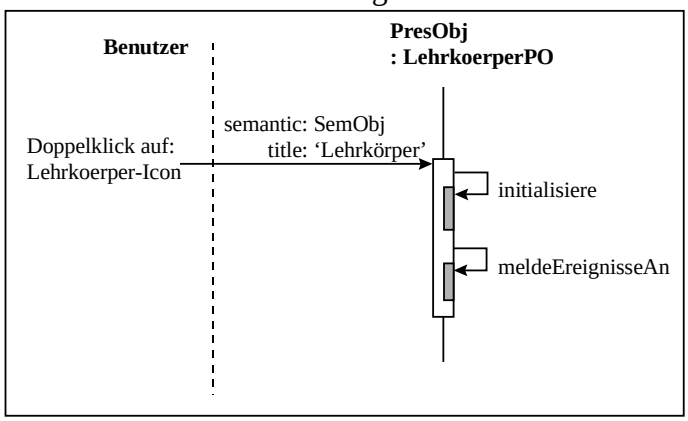


Abb. 20: Erzeugung einer LehrkoerperPO

Soll eine IAK beendet werden, so muß folgende Nachricht an ihre FK gesendet werden:

self meineFK beendeIAK: self

Dies sorgt dafür, daß das Ereignis #beenden an die zugehörige BasisIAK gesendet wird (Abb. 21). Diese sendet die Nachricht *beenden* an das Objekt der abgeleitete Klasse und schließt das Fenster. Die IAK sollte daher die Methode *beenden* überladen und notwendige Aufräumarbeiten darin erledigen. Dazu gehört auch das Zurücknehmen des Interesses für Ereignisse.

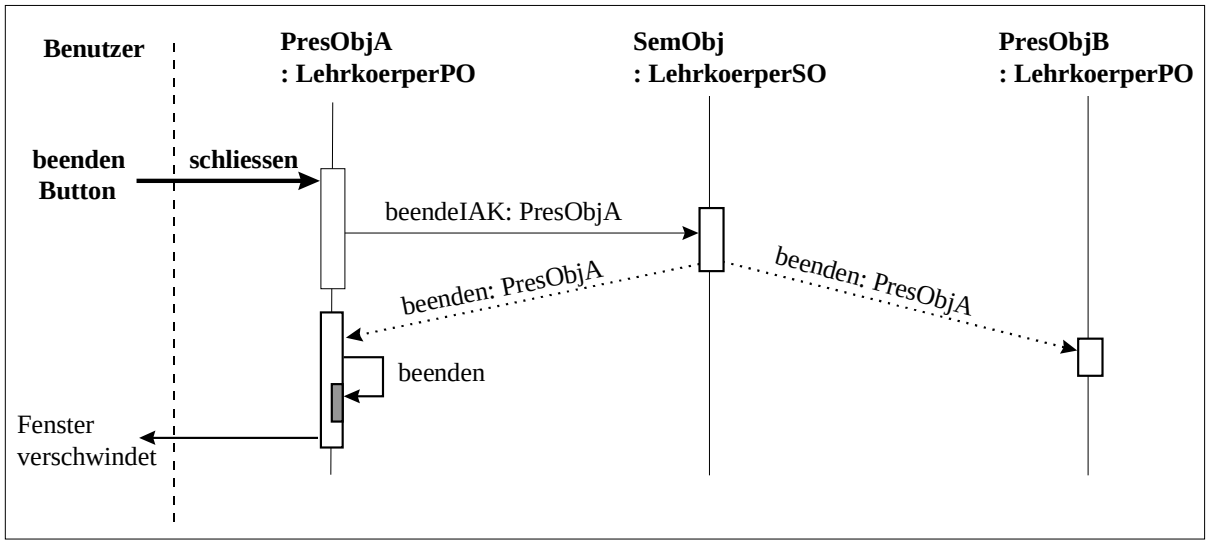


Abb. 21: Schließen einer LehrkoerperPO

5.2.2 Das Lehrkörpertool

Das **Lehrkörpertool** (Abb. 22) dient der Verwaltung von Lehrern eines Lehrkörpers. Dazu werden zugehörigen Lehrer in einer Listbox angezeigt. Der Erfassung und Änderung eines Lehrers dient ein Subtool (**Lehrertool**), welches sich nach Betätigung des Neu-Buttons bzw. durch Doppelklick auf einen Listeneintrag öffnet. Zum Löschen eines Lehrers aus dem Lehrkörper dient ein weiterer Button.

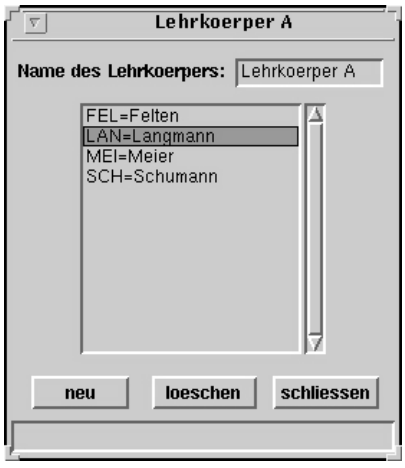


Abb. 22: Lehrkörpertool

Wenn das Lehrertool (Abb. 23) mittels Neu-Button im Lehrkörpertool geöffnet wird, so sind alle Eingabefelder leer. Diesen Zustand erreicht man auch durch den Reset-Button im Lehrertool. Durch Öffnen mittels Doppelklick werden die Eingabefelder mit den entsprechenden Werten des aktuellen Lehrers gefüllt. Das Lehrerkürzel ist ein eindeutiges Kürzel, das den Lehrer identifiziert. Die Stelle gibt an wieviel Prozent der regulären wöchentlichen Arbeitszeit der Lehrer unterrichtet. Die Einstellung Aufsichtsberechtigt gibt an, ob der Lehrer eine Pause verant-

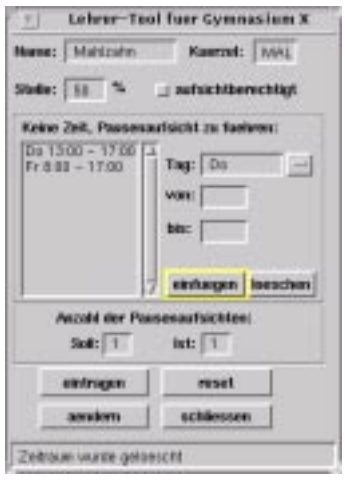


Abb. 23: Lehrertool

wortlich beaufsichtigen darf. Für jeden Lehrer können Zeiträume in die Listbox eingetragen werden während derer der Lehrer keine Zeit hat, Pausenaufsicht zu führen. Aufgrund der Dualität können diese auch wieder gelöscht werden.

In der unteren Hälfte des Fensters werden die Soll- und die Ist-Pausenaufsichtszahlen angezeigt. Der Sollwert wird automatisch aktualisert, wenn ein Lehrer dem Lehrkörper hinzugefügt oder entfernt wird oder wenn sich der Beschäftigungsgrad eines Lehrers ändert. Der Istwert ändert sich, wenn sich für den Lehrer eine Pausenaufsicht ändert.

Die beiden Tool sind auf mehrere Arten miteinander verknüpft: Wird im Lehrkörpertool der Schließen-Button gedrückt, beendet sich auch das Lehrertool. Das Lehrertool zeigt in der Titelzeile an, zu welchem Lehrkörpertool es gehört. Diese Anzeige wird bei Änderung des Lehrkörpernamens (rechte Maustaste auf das zugehörige Icon, dann „Edit Change Link Title“) aktualisiert. Wird ein Doppelklick auf einen Lehrereintrag gemacht, so aktualisiert sich auch das Lehrertool entsprechend dieser Auswahl.

Ein Lehrkörper kann beliebig häufig geöffnet werden, indem das Lehrkörpertool mehrfach gestartet wird. Alle Änderungen werden sofort an alle geöffneten Ansichten dieses Lehrkörpers propagiert. Dies gilt ebenso für verteilte Zugriffe auf verschiedenen Rechnern.

5.2.3 Der Pausenplaner

Mit dem Pausenplaner (Abb. 24) werden die Pausenaufsichten von Lehrern eines Lehrkörpers verwaltet. Der Lehrkörper wird dem Pausenplan per Drag & Drop vom Lehrkörpernamen des Lehrkörpertools in den des Pausenplaners zugeordnet. Durch Drag & Drop kann ein Lehrer aus dem Lehrkörpertool auf eine vorher markierte Pause gezogen werden, wodurch er dort eingetragen wird. Ist ein Lehrer markiert, kann die Pausenaufsicht mit Hilfe des Löschen-Buttons entfernt werden.

Abb. 24: Pausenplaner

Die in der Systemvision in Kapitel 2.2 vorgesehenen Probleme werden von diesem Werkzeug durch Einfärben hervorgehoben. Es werden zwei Arten der Markierung unterschieden: Ist eine Pause problembehaftet, so wird sie komplett in einer besonderen Farbe hinterlegt. Ist ein bestimmter Lehrer die Ursache eines Konfliktes, so wird sein Kürzel farbig hervorgehoben. Diese Arten der Markierung werden im entsprechenden Fall kombiniert. Die genaue Bedeutung der Farben sind den Tabellen zu entnehmen.

Farbe	Bedeutung
schwarz	Kein Konflikt.
rot	Lehrer nicht im Lehrkörper des Pausenplans enthalten.
blau	Lehrer hat zur angegeben Pause keine Zeit. Ist der Lehrer in der Vertretungsliste, so hat er im Laufe des Tages keine Zeit.
grün	Der Lehrer ist in einer Pause an zwei Orten eingetragen.

pink	Der Lehrer ist an einem Tag sowohl in einer Pausenaufsicht als auch in der Vertretungsliste eingetragen.

Abb. 25: Vordergrundfarben des Lehrers

Farbe	Bedeutung
grau	Pause ist ordnungsgemäß beaufsichtigt.
weiß	Die Pause wird nicht von genügend Lehrern beaufsichtigt.
gelb	Kein aufsichtsberechtigter Lehrer beaufsichtigt die Pause.

Abb. 26: Hintergrundfarben der Pause

Bei einem neuen Eintrag überprüft der Pausenplaner, ob das Lehrerkürzel einen im Lehrkörper enthaltenen Lehrer ergibt. Ist dies nicht der Fall so wird die Eingabe zurückgewiesen.

Auch dieses Werkzeug kann beliebig viele Präsentationen (Präsentationsobjekte) - auch auf unterschiedlichen Rechnern - besitzen, welche bei Änderungen am zugrundeliegenden Material alle automatisch aktualisiert werden.

5.3 Ereignisverwaltung

Für die Realisierung des Beobachtermechanismus mußte einerseits ein angemessenes Designpattern, andererseits eine standardisierte Form der Verteilung gewählt werden. Da DST den CORBA Event-Service anbietet, wird dieser verwendet um die Benachrichtigung zu realisieren. Der Event-Service wird von ParcPlace Event-Notification genannt und wird in Kapitel 5.3.1 beschrieben.

Da die Kombination dieses Event-Service mit einem einfachen Muster wie dem Observer-Pattern [GAMM95, S. 293ff.] schwer zu realisieren wäre, wird das um den ChangeManager erweiterte Muster [GAMM95, S. 299f.] verwendet. Die zentrale Klasse ist der EreignisVerwalter (Kapitel 5.3.2), der den gesamten Event-Mechanismus kapselt.

Durch die Benutzung des Event-Service sollte ein asynchroner Event-Mechanismus erreicht werden, der für die Verteilung Performance und Sicherheitsvorteile verspricht. Ein Nachteil besteht in der Problematik der erneuten Zustandsänderung der ereignisauslösenden Objekte, bevor die benachrichtigten Objekte alle Sondierungsabfragen durchgeführt haben. Dieses Problem ließe sich jedoch durch eine feine Granularität der Ereignisse in den Griff bekommen, tritt in DST jedoch gar nicht erst auf, da Distributed Smalltalk den Event-Service synchron implementiert hat.

5.3.1 DST Event-Notification

Die DST Event-Notification realisiert den von CORBA 2.0 vorgeschriebenen Event-Service. Die zugehörigen Klassen befinden sich in der Klassenkategorie **COS-Event**. Objekte, die an diesem Service partizipieren, müssen jeweils eine von zwei Rollen einnehmen: Der **Supplier** (Objekte der Klassen DSTPushSupplier, DST-PullSupplier) sendet Ereignisse, der **Consumer** (DSTPushConsumer, DSTPullConsumer) empfängt diese. Die Ereignisse werden vom Supplier einem **EventChannel** (einer Instanz der Klasse DSTEventChannel) übergeben, der die Ereignisse an die Consumer verteilt. Der EventChannel verhält sich demnach sowohl als Supplier wie auch als Consumer (Abb. 27). Er unterscheidet nicht zwischen verschiedenen Ereignissen: Sobald ein Supplier ein Ereignis sendet, werden alle Consumer benachrichtigt. Das Ereignis kommt bei den Consumern als Parameter an.

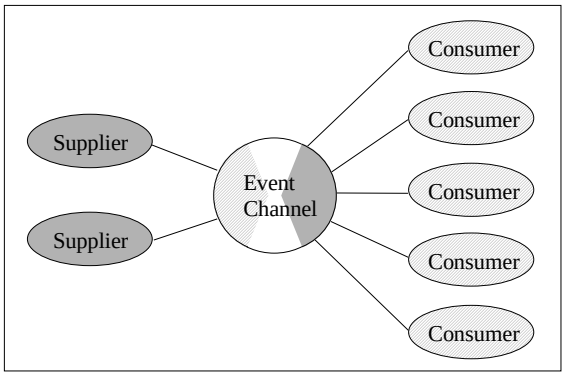


Abb. 27: Event-Channel

Es gibt zwei Arten auf die Ereignisse vom Supplier zum Consumer gelangen können: Das Push- und das Pull-Modell. Beim **Push-Modell** (Abb. 28 [ORFA96, S.120]) übernimmt der Supplier die Initiative und sendet das entsprechende Ereignis zum Event-Channel, der das Ereignis zu den Consumern weiterleitet (pushed). Damit ein Consumer Ereignisse empfangen kann, muß er sich vorher beim EventChannel anmelden, dabei gibt er an, welche Nachricht ihm geschickt werden soll; wenn der Consumer Ereignisse nicht mehr empfangen möchte, kann er sich beim EventChannel abmelden.

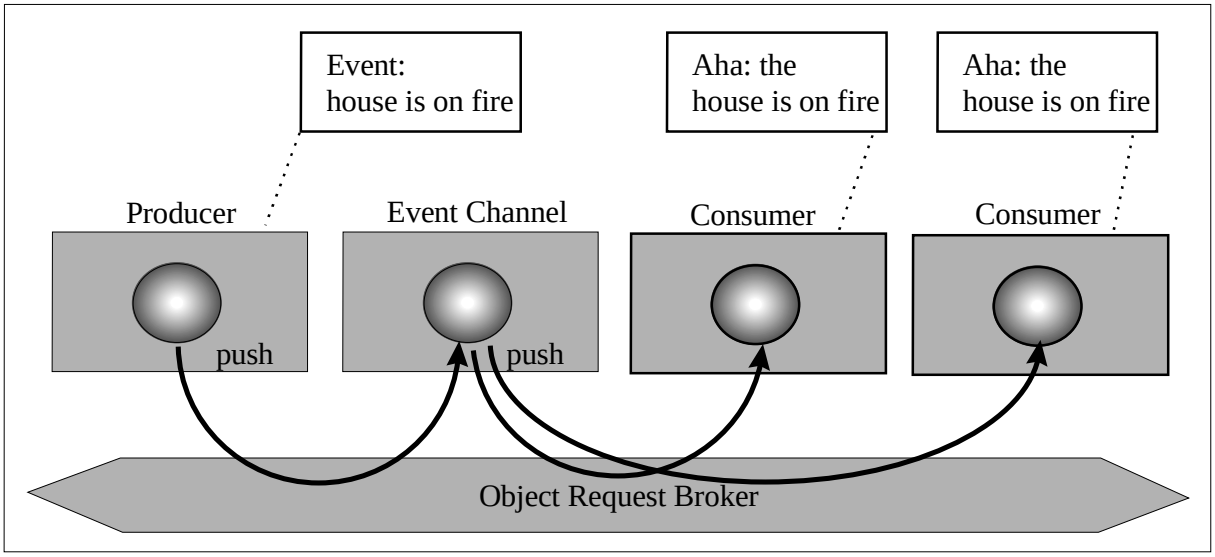


Abb. 28: Event-Service: Push-Modell

Beim **Pull-Model** (Abb. 29 [ORFA96, S. 120]) muß der Consumer die Initiative übernehmen: Er erfragt beim EventChannel, ob für ihn ein Ereignis vorliegt. Der EventChannel leitet diese Anfrage an den Supplier weiter.

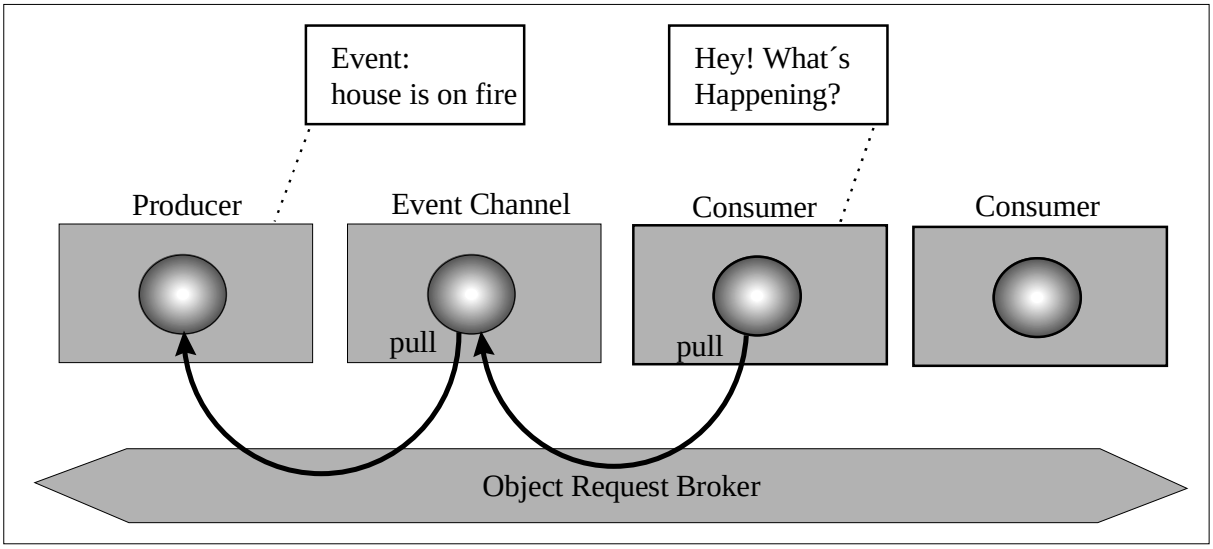


Abb. 29: Event-Service: Pull-Modell

Push- und Pull-Modell können für einen EventChannel kombiniert werden, so daß Pull-Supplier auch mit Push-Consumern und Push-Supplier mit Pull-Consumern kommunizieren können. Abb. 30 zeigt ein Beispiel für eine komplexere Situation.

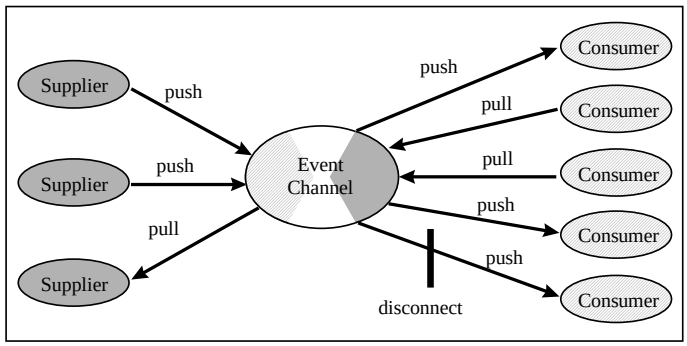


Abb. 30: Kombination aus Push- und Pull-Modell

Die Realisierung dieses Service erfolgt mit Hilfe von Proxy-Objekten (Abb. 31). Zwischen Consumer bzw. Supplier und dem EventChannel liegen jeweils die korrespondierenden Proxies, so daß ein Supplier immer direkt mit genau einem Consumer (DSTProxyConsumer) und ein Consumer immer mit einem Supplier (DSTProxySupplier) verbunden ist. Die Anwendungsobjekte (z.B. Presentations- und Semantikobjekt) werden als Hostobjekte von DSTSupplier bzw. DSTConsumer bezeichnet.

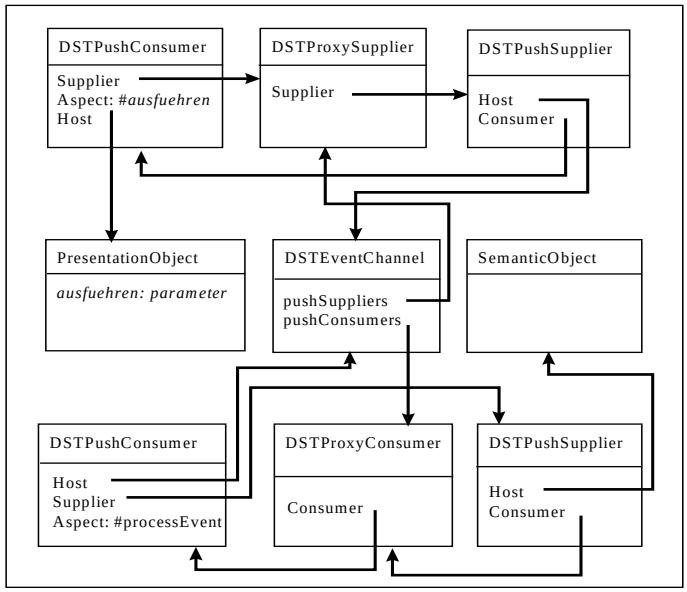


Abb. 31: Interner Aufbau des DST-EventService

Da die Klasse DSTEventChannel beim Abmelden von Consumer und Supplier fehlerbehaftet ist, haben wir die Klasse **EventChannelKorrigiert** von DSTEventChannel abgeleitet und die fehlerhaften Nachrichten überschrieben.

5.3.2 Die Klasse EreignisVerwalter

Der Ereignisverwalter (kurz für: eine Instanz der Klasse EreignisVerwalter) stellt dem Entwickler ein einfaches Protokoll zur Benutzung des Event-Service zur Verfügung. Er beschränkt sich dabei auf das in Kapitel 5.3.1 beschriebene Push-Modell, weil dieses dem Konzept des Beobachtermusters entspricht.

Eine neue Instanz eines Ereignisverwalters wird mit

EreignisVerwalter new

erzeugt.

Der Ereignisverwalter kennt Sender, Empfänger und (parametrisierte) Ereignisse, die durch Symbole repräsentiert werden. Diese Ereignisse unterscheiden sich von den aus Kapitel 5.3.1 bekannten dadurch, daß sie sowohl benannt als auch mit einem Parameter versehen sind. Dadurch kann der Ereignisverwalter Interessenten von verschiedenen Ereignissen differenzieren. Sender müssen sich für alle Ereignisse, die sie senden wollen, anmelden. Durch die erste Anmeldung eines Senders für ein Ereignis wird dieses Ereignis dem Ereignisverwalter bekannt gemacht. Ein Objekt meldet sich beim Ereignisverwalter mit der Nachricht

anmeldenAlsSender: einSender fuer: einEreignis

an.

Ein Ereignis kann mit oder ohne Parameter gesendet werden:

sendeEreignis: einEreignis sender: einSender
sendeEreignis: einEreignis sender: einSender mit: einParameter

Wird kein Parameter mitgesendet, dann bekommt der Empfänger den Sender als Parameter für das Ereignis mit. Der Sender bekommt die Kontrolle sofort zurück.

Wenn ein Sender ein Ereignis nicht mehr senden möchte, sollte er sich beim Ereignisverwalter für dieses Ereignis abmelden. Falls der Sender kein Ereignis mehr senden wird, so kann er sich für alle Ereignisse, für die er sich registriert hat, auf einmal abmelden. Dafür werden folgende Nachrichten angeboten:

senderAbmelden: einSender fuer: einEreignis
senderAbmelden: einSender

Empfänger müssen sich für jedes Ereignis, welches sie interessiert, beim Ereignisverwalter registrieren. Dabei müssen sie mitteilen, welche Nachricht ihnen bei Eintritt des Ereignisses geschickt werden soll, wobei zu beachten ist, daß die Nachricht bei der Übergabe als Symbol ohne Doppelpunkt angegeben werden muß; bei der Implementation muß die Nachricht jedoch einen Parameter akzeptieren.

anmeldenAlsEmpfaenger: einEmpfaenger fuer: einEreignis
benachrichtigen: eineMethode

Möchte ein Objekt nicht mehr bei Eintritt eines Ereignisses benachrichtigt werden, so muß es sich wieder vom Ereignisverwalter abmelden. Dies kann für einzelne oder pauschal für alle Ereignisse, für die das Objekt registriert ist, geschehen:

empfaengerAbmelden: einEmpfaenger fuer: einEreignis
empfaengerAbmelden: einEmpfaenger

Die Schnittstelle des Ereignisverwalters ist in IDL verfügbar. Entsprechend müssen alle Empfänger von Ereignissen ebenfalls ein IDL-Interface besitzen, in der vor allem die zu schickenden Nachrichten enthalten sein müssen.

Aus technischer Sicht funktioniert der Ereignisverwalter folgendermaßen: Für jedes neue Ereignis wird ein EventChannelKorrigiert erzeugt, bei dem die entsprechenden Sender und Empfänger eingetragen werden, so wie es für den DST Notification Service beschrieben ist [DSPR96, S. 17ff.]. Das Paar Ereignis und EventChannel wird in einem IdentityDictionary gespeichert. Zusätzlich werden im EreignisVerwalter Sender und Empfänger zu jedem Ereignis in jeweils einem IdentityDictionary gehalten. Für jedes Ereignis stehen in diesen Listen weitere IdentityDictionary, deren Schlüssel Sender bzw. Empfänger und deren Wert die dazugehörigen DSTPushSupplier bzw. DSTPushConsumer sind (Abb. 32). Beim Senden eines Ereignisses werden aus der Senderliste die möglichen Sender dieses Ereignisses geholt und von denen der zu dem Sender passende DSTPushSupplier gesucht. Über den kann der DST Event-Notification Mechanismus ausgeführt werden.

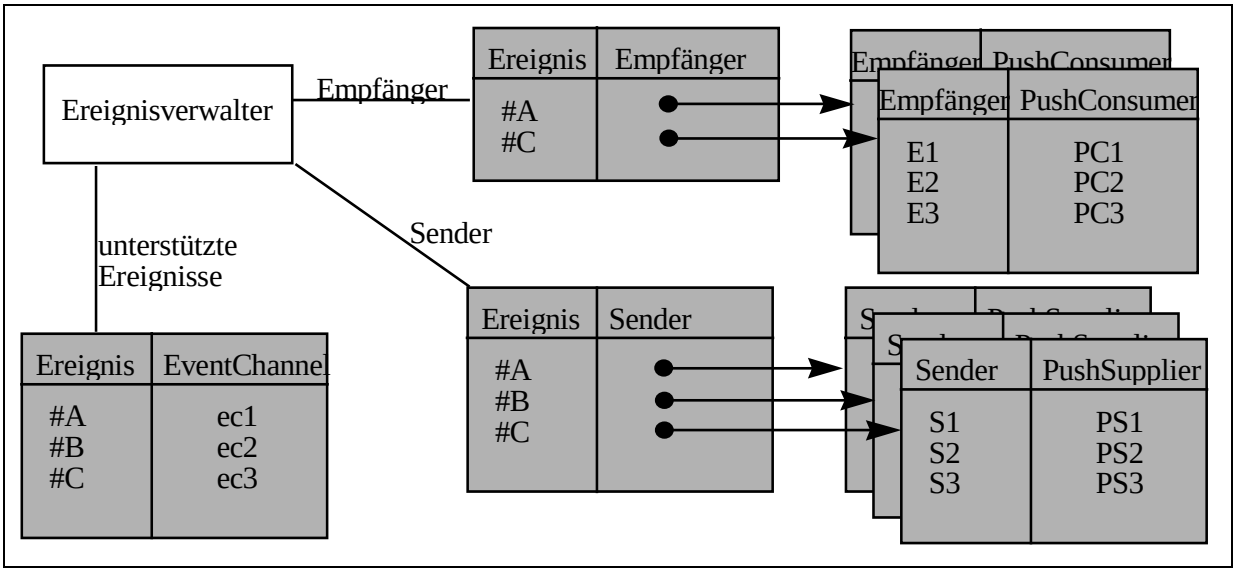


Abb. 32: Komponenten des Ereignisverwalters

5.4 Materialverwaltung

Das Problem, aus dem der Materialverwalter resultiert, liegt in der aus softwaretechnischen Gesichtspunkten geforderten paarweisen Unabhängigkeit von Werkzeugen: Einige müssen Materialien bearbeiten, die andere Werkzeuge erzeugt haben. Diese wiederum erfahren nichts von fremden Änderungen an „ihrem“ Material. Als Beispiel sei das Lehrertool angeführt, das der Erzeugung und Bearbeitung von Lehrern dient, die vom Pausenplan benötigt werden. Verändert der Benutzer im Pausenplan-Werkzeug die Pausenaufsichten eines Lehrers, so muß das Lehrertool davon erfahren können.

Das Protokoll um die Materialverwaltung sieht vor, daß Materialien, die von verschiedenen Werkzeugen benutzt werden können, bei der Materialverwaltung registriert werden. Materialien können durch die Vergabe eines eindeutigen Bezeichners identifiziert werden, so daß Werkzeuge über den Bezeichner das Material bekommen können.

Um Werkzeuge von Änderungen am Material zu unterrichten, benutzt die Materialverwaltung den Ereignisverwalter. Werkzeuge, die über Änderungen an einem Material informiert werden wollen, müssen sich als Interessent für dieses Material anmelden. Das Werkzeug kann genau festlegen, für welche Ereignisse es sich interessiert. Welche Ereignisse für ein Material zulässig sind, wird bei der Registrierung des Materials festgelegt.

Damit die Werkzeuge alle dieselbe Materialverwaltung ansprechen, wurde diese als Singleton realisiert [GAMM95, S. 127ff]. Dabei entsteht das Problem, daß trotz Benutzung von Klassenmethoden in jedem Image eine Instanz der Materialverwaltung existieren würde, welche unabhängig voneinander wären und vor allem nichts von den verwalteten Materialien wüßten. Um das zu verhindern, wird das Singleton-Pattern mit Hilfe des von DST angebotenen CORBA Naming Service (Kapitel 5.4.1) auf den gesamten Namensraum des ORBs ausgedehnt.

Die Klasse, die die Materialverwaltung realisiert, ist der MaterialVerwalter. Seine Funktionsweise und Benutzung ist in Kapitel 5.4.2 beschrieben.

5.4.1 DST Naming-Service

Der Naming Service dient dazu, Objekte im ORB zu lokalisieren. Jedes Objekt wird an einen eindeutigen Namen gebunden. Um Namenskonflikte zu vermeiden wurden Namensräume (Name Context) eingeführt. Diese können andere Namensräume oder einfache Namen (Simple Name oder Reference ID [ORFA96, S. 110]) enthalten. Namensräume und IDs bilden einen Namensbaum (Naming Graph [DSPR96, S. 10f.]). Ein kompletter Objektbezeichner besteht aus einer Sequenz von Name Contexts und einem abschließenden Simple Name (Wegbezeichnung von der Wurzel zum Blatt). Ein solcher Pfad wird Compound Name genannt. Im Beispiel (Abb. 33) wäre ein Compound Name z.B. /Griechenland/Samos oder /USA/FL/Miami.

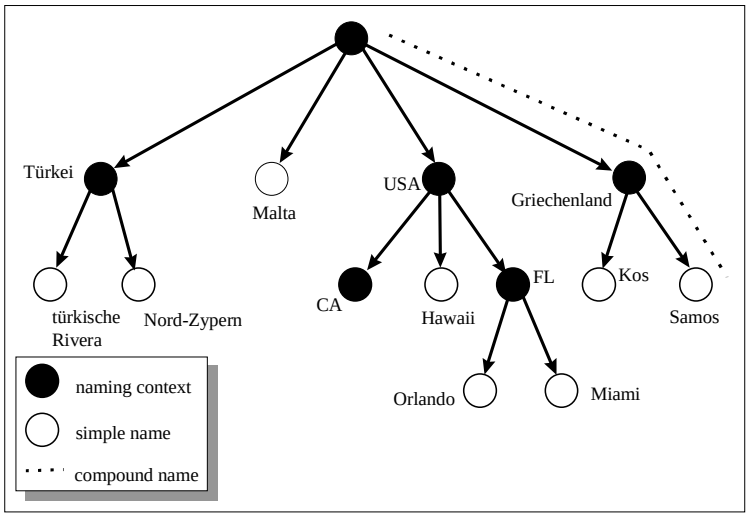


Abb. 33: Beispiel für Naming Service: Reiseziele

Um den Name Service ansprechen zu können, bietet DST die Möglichkeit, diesen CORBA Service und andere initiale DST-Objekte direkt beim ORB anzufordern (InterfaceRepository, FactoryFinder, UserSecurityDatabase). Dazu wird an die Klasse ORBObject die Nachricht *resolveInitialReferences: einSymbol* mit dem passenden Wert für einSymbol gesandt. Der Name Service kommt als Instanz der Klasse DSTNameContext zurück und befindet sich an der Wurzel des Namensgraphen. Der komplette Aufruf sieht folgendermaßen aus:

ORBObject resolveInitialReferences: #NameService

Die Klassen, deren Objekte in DST den Namensbaum aufspannen, befinden sich in der Klassenkategorie COS-Naming. Die Instanzen der Klasse DSTName repräsentieren einen vollständigen Namen, die der Klasse DSTNameContext jeweils einen Namensraum. Einen Simple Name erhält man entweder, indem an die Klasse DSTName die Nachricht *onString: einString* geschickt wird oder indem an einen String die Nachricht *asDSTName* gesendet wird. Ein Compound Name wird entsprechend erstellt, indem die Nachricht *onStrings:* verwendet wird. Der Parameter muß jedoch ein Array sein, welches die Komponenten als Strings enthält. Auch die Nachricht *asDSTName* ergibt einen Compound Name, wenn sie an ein solches Array geschickt wird.

Zwischen Namen und Objekten besteht eine eindeutige Zuordnung, die Binding genannt wird. Namen werden an Objekte gebunden, indem dem Name Context, in welches das Objekt eingefügt werden soll, die Nachricht: *contextBind: einDSTName to: einObjekt* gesendet wird. Soll in einen bekannten Namensraum N der Namensraum einDSTNameContext mit dem DSTNamen einDSTName eingefügt werden, so wird an N die Nachricht: *contextBindContext: einDSTName to: einDSTNameContext* gesandt. Entsprechend gibt es Nachrichten zum Auflösen und Ändern von Name-Objekt-Bindungen.

Um ein Objekt zu finden, dessen Name - auch als Compound Name - bekannt ist, wird *contextResolve: einDSTName* an den DSTNameContext gesendet, von dem der im einDSTName enthaltene Pfad entspringt.

5.4.2 Die Klasse MaterialVerwalter

Die Instanz der Klasse MaterialVerwalter (in Zukunft einfach nur „der Materialverwalter“) dient zur Versorgung der Werkzeuge mit Materialien und der Benachrichtigung der Werkzeuge bei Materialänderungen. Der Materialverwalter wurde als Singleton implementiert. Da die einzige Instanz unabhängig von den Anwendungen initialisiert und gelöscht werden kann, wird dringend empfohlen, den Materialverwalter nicht dauerhaft zu halten. Die Instanz erhält man durch:

MaterialVerwalter materialVerwalter.

Der Materialverwalter wird durch

MaterialVerwalter delete.

gelöscht.

Bevor Werkzeuge Materialien beim Materialverwalter anfordern oder von Materialänderungen erfahren können, müssen Materialien angemeldet worden sein. Dies wird in der Regel vom Materialerzeuger getan. Der Anmelder eines neuen Materials definiert die Ereignisse, die ein Material betreffen. Für dem Materialverwalter bekanntes Material können sich Werkzeuge daraufhin als Interessent für Änderungen registrieren lassen. Dies gilt auch für den Anmelder des Materials, wenn er benachrichtigt werden möchte.

Zur Anmeldung eines neuen Materials gibt es zwei Nachrichten:

materialAnmelden: material anmelder: eineSO
materialAnmelden: material anmelder: eineSO ereignisse: ereignisse

In beiden Fällen wird das Semantikobjekt als Benutzer des Materials eingetragen und für das Material ein eindeutiger Name vergeben, der an den Sender zurückgegeben wird. Mit der zweiten Nachricht kann zusätzlich eine Menge von Ereignissen als Array übergeben werden, mit denen die Granularität der Information über Änderungen am Material festgelegt werden. Ereignisse sind wie beim Ereignisverwalter Symbole. Unabhängig davon, welche Nachricht zur Anmeldung benutzt wurde, wird bei jeder Materialänderung das Ereignis #MaterialVeraendert gesendet, für das sich ein Werkzeug auch aktiv interessieren muß. Der Rückgabewert ist ein eindeutiger Name für das Material.

Wenn ein Werkzeug das Material zu einem gegebenen Namen haben möchte (s. Kapitel 5.5), so kann es den Materialverwalter danach fragen. Eine Nachricht für die umgekehrte Richtung existiert ebenfalls:

materialName: einMaterial

material: einMaterialName

Mit den beiden folgenden Nachrichten registriert sich ein Werkzeug bei dem Materialverwalter als Halter eines Materials; der Rückgabewert beider Methoden ist das Material:

material: einMaterialName interessent: eineSO

material: einMaterialName interessent: eineSO nachricht: nachrichten

In beiden Fällen wird der Interessent als Benutzer gespeichert. Mit der ersten Nachricht meldet sich eineSO als Halter des Materials an, möchte aber bei Änderung am Material nicht benachrichtigt werden. Mit der zweiten Nachricht kann zusätzlich entweder ein Symbol oder eine Menge von Ereignis/Nachricht-Paaren angegeben werden. Dies kann in der Form #((e1 n1) (e2 n2) ...) geschehen. Wird als Nachricht nur ein Symbol angegeben, so wird dieses Symbol als Name einer parametrisierten Nachricht angesehen und diese bei jeder Änderung am Material an das Werkzeug gesendet. Werden Ereignis/Nachricht-Paare angegeben, so wird bei Auftreten eines der angegebenen Ereignisse die entsprechende Nachricht an das Werkzeug gesandt.

Änderungen, die an einem Material vorgenommen worden sind, werden nicht automatisch an die interessierten Werkzeuge propagiert, da dies einen Automatismus innerhalb der Materialien erfordern würde: Materialien müßten den Materialverwalter kennen und ihm die Änderungen mitteilen. Materialien haben jedoch „unabhängig von der konkreten Systemplattform“ [KILB94, S. 29] zu sein und damit auch vom Beobachtungsmechanismus. Statt dessen muß jedes Werkzeug, nachdem es ein Material verändert hat, diese Änderung an den Materialverwalter melden. Dazu dienen folgende Nachrichten:

materialVeraendert: einMaterial sender: eineSO

materialVeraendert: einMaterial sender: eineSO ereignis: einEreignis

materialVeraendert: einMaterial sender: eineSO ereignis: einEreignis
parameter: einParameter

Der Parameter *eineSO* wird derzeit noch nicht benötigt. Er ist aber sinnvoll für Erweiterungen des Materialverwalters um beispielsweise einen Logging-Mechanismus. Das Ereignis kann entweder #MaterialVeraendert (dann entspricht die zweite Alternative der ersten) oder eines der vom Anmelder definierten Ereignisse sein. #MaterialVeraendert wird auch bei jedem anderen Ereignis an die Interessenten gesendet. Als Parameter kann prinzipiell ein beliebiges Objekt mitgesendet werden. Es wird aber empfohlen, daß der Entwickler, der die Ereignisse festlegt, eine Semantik für die übergebenen Parameter festlegt und dies dokumentiert. Wird kein expliziter Parameter angegeben, so wird das veränderte Material den Interessenten als Parameter übergeben.

Wird ein Material für ein Werkzeug uninteressant, sei es, daß das Werkzeug beendet oder ein anderes Material geladen wird, so kann mit folgender Methode das Interesse für das Material abgegeben werden.

MaterialZurueck: einMaterial interessent: eineSO

Sinnvollerweise sollte innerhalb des Werkzeuges jede Referenz auf das Werkzeug gelöst werden, da der Materialverwalter das Werkzeug aus der Liste der das Material Benutzenden austrägt.

Weiterhin werden folgende Sondierungsnachrichten angeboten:

ereignisseDesMaterials: einMaterial

benutzerVon: einMaterial

materialnamen

Alle Rückgaben erfolgen als OrderedCollection und sind neu erstellte Listen der vom Materialverwalter gehaltenen Objekte.

Aus technischer Sicht funktioniert die Materialverwaltung folgendermaßen: Jedes Material, welches der Materialverwalter kennt, dient als Schlüssel in drei IdentityDictionaries, die die Materialnamen, die Benutzer eines jeden Materials und den Ereignisverwalter für jedes Material halten. Weiterhin dient der Klassenname des Materials als Eintrag in ein IdentityDictionary, in dem die Anzahl der Objekte dieser Klasse gespeichert ist. Ein Materialname setzt sich zusammen aus dem Klassennamen des Materials, einem Doppelpunkt und des laufenden Klassenzählers des Materials (Abb. 34). Diese Einträge werden jeweils bei der Anmeldung eines neuen Materials, dem Registrieren von Interessenten bzw. bei der Zurückgabe eines Materials aktualisiert. Wie aus der Abbildung ersichtlich, ist für jedes Material genau ein Ereignisverwalter zuständig. Ein und dasselbe Material läßt sich nicht mehrmals anmelden.

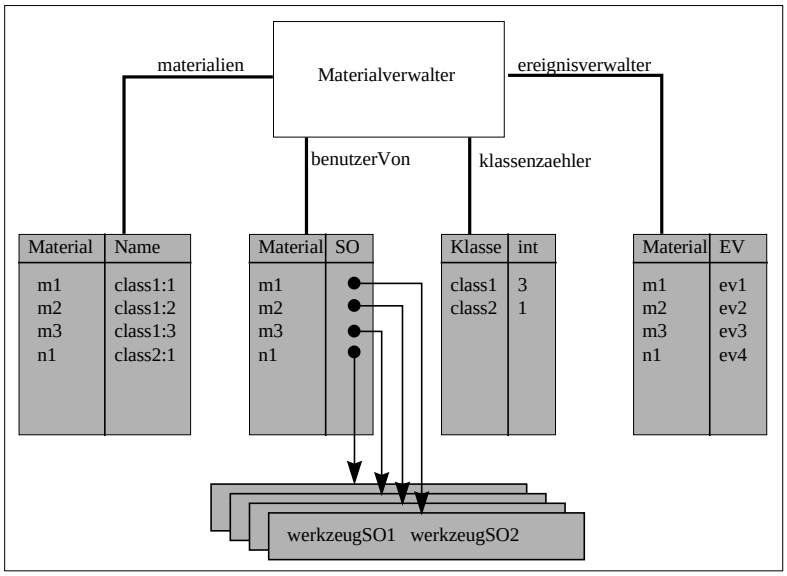


Abb. 34: Komponenten des Materialverwalters

Die Klasse MaterialVerwalter besitzt eine Klassenvariable Instanz, welche die einzige Instanz dieser Klasse hält. Mit *materialVerwalter* kann auf die Instanz zugegriffen werden. Enthält dieses Klassenattribut ein gültiges Objekt so bekommt man dieses. Wenn nicht wird geprüft, ob der ORB bereits den Materialverwalter kennt; dieser Fall kann in einem entfernten Image bei der Initialisierung einer neuen Anwendung eintreten. Ob der ORB den Materialverwalter kennt, wird über den DST Naming-Service (Kapitel 5.4.1) abgefragt, indem nach dem fest vorgegebenen CORBA-Namen „materialverwalter“ gesucht wird. Ist der Materialverwalter dem ORB bereits bekannt, so bekommt man die Instanz, ansonsten wird eine erzeugt und beim Naming Service an den Namen „materialverwalter“ gebunden.

Soll der Materialverwalter gelöscht und damit die Materialverwaltung initialisiert werden, so geschieht das über die Klassenmethode *delete*. Dies ist mit außerordentlicher Vorsicht durchzuführen, da alle Werkzeuge, die den Materialverwalter benutzen, nicht mehr über Materialänderungen informiert werden können und u.U. ungültige Materialnamen halten. Infolgedessen müssen alle Anwendungen geschlossen werden. Insbesondere bedeutet dies, daß es nicht reicht, alle Fenster zu schließen, sondern daß auch die entsprechenden Icons der Semantikobjekte entfernt werden müssen. Beim Löschen werden alle IdentityDictionaries und das Klassenmember Instanz initialisiert und die Bindung an den CORBA Namen „materialverwalter“ beim Name Service gelöst.

Zur Frage, welches Material angemeldet werden sollte, gilt die Empfehlung: Konstante Objekte sollten nicht angemeldet werden, da sie sich per definitionem nicht ändern können. Einfache Objekte, wie z.B. Strings oder Zahlen, müssen nicht angemeldet werden, da sie nur im Zusammenhang mit den sie einbettenden Objekten Sinn ergeben. Häufig ist es aber sinnvoll, ihnen ein eigenes Ereignis bei Veränderung innerhalb des übergeordneten Materials zu gönnen. Objekte, die in der Implementation verborgen sind, brauchen nicht angemeldet werden, da fremde Werkzeuge von solchen Materialien keine Kenntnis haben sollten.

5.5 Technisches Zusammenspiel der Werkzeuge

Im Gegensatz zur Anwendersicht, in der der Lehrkörper nach dem Drag & Drop scheinbar direkt mit dem Pausenplaner kommuniziert, kennt aus der technischen Sicht jedes Werkzeug nur den Materialverwalter und kein anderes Werkzeug.

Beim Start des Drags des Lehrkörpernamens holt sich das Lehrkörpertool den Identifier (z.B. ‘Lehrkoerper:4’) des Lehrkörpers vom Materialverwalter. Dieser Identifier wird mit dem Dragkontext zum Pausenplaner übertragen, der sich so das Material vom Materialverwalter besorgt und intern als Attribut speichert. Zusätzlich meldet sich der Pausenplaner für alle Ereignisse, die diesen Lehrkörper betreffen, beim Materialverwalter an. So kann der Pausenplaner gezielt auf Veränderungen am Lehrkörper reagieren.

Etwas anders funktioniert das Drag & Drop mit einem Lehrer: Mit dem Dragkontext wird das Lehrerkürzel - nicht der Identifier des Lehrers im Materialverwalter - übertragen. Der Grund dafür ist, daß der Lehrer dem Pausenplaner nicht bekannt sein soll, wenn er nicht im Lehrkörper des Pausenplaners enthalten ist.

Da (vereinfachend) davon ausgegangen wird, daß das Lehrkörpertool das einzige Werkzeug ist, mit dem der Lehrkörper verändert werden kann, meldet es beim Materialverwalter kein Interesse für den Lehrerkörper an. Das Lehrkörpertool meldet allerdings selber Änderungen am Lehrkörper, so daß neue Werkzeuge immer über Änderungen informiert werden, wenn sie Interesse angemeldet haben.

Anhand des Lehrers läßt sich die Granularität von Materialveränderungen diskutieren: Jede Änderung am Lehrer oder am Pausenplan wird durch ein #materialVeraendert Ereignis angezeigt, während Änderungen am Lehrkörper zusätzlich sehr fein detailliert sind. Das liegt an der Betrachtungsweise der modellierten Objekte: Der Pausenplan ist wie der Lehrer eine in sich geschlossene Einheit; der Lehrkörper ist eine Aggregation von Lehrern. Eine starke Aufsplittung in Ereignisse kann unübersichtlich und daher fehleranfällig werden. Andererseits kann eine grobe Strukturierung zu Performance-Einbußen führen, weil u. U. viele Sondierungsabfragen nötig sind bzw. immer das komplette Material neu dargestellt werden muß.

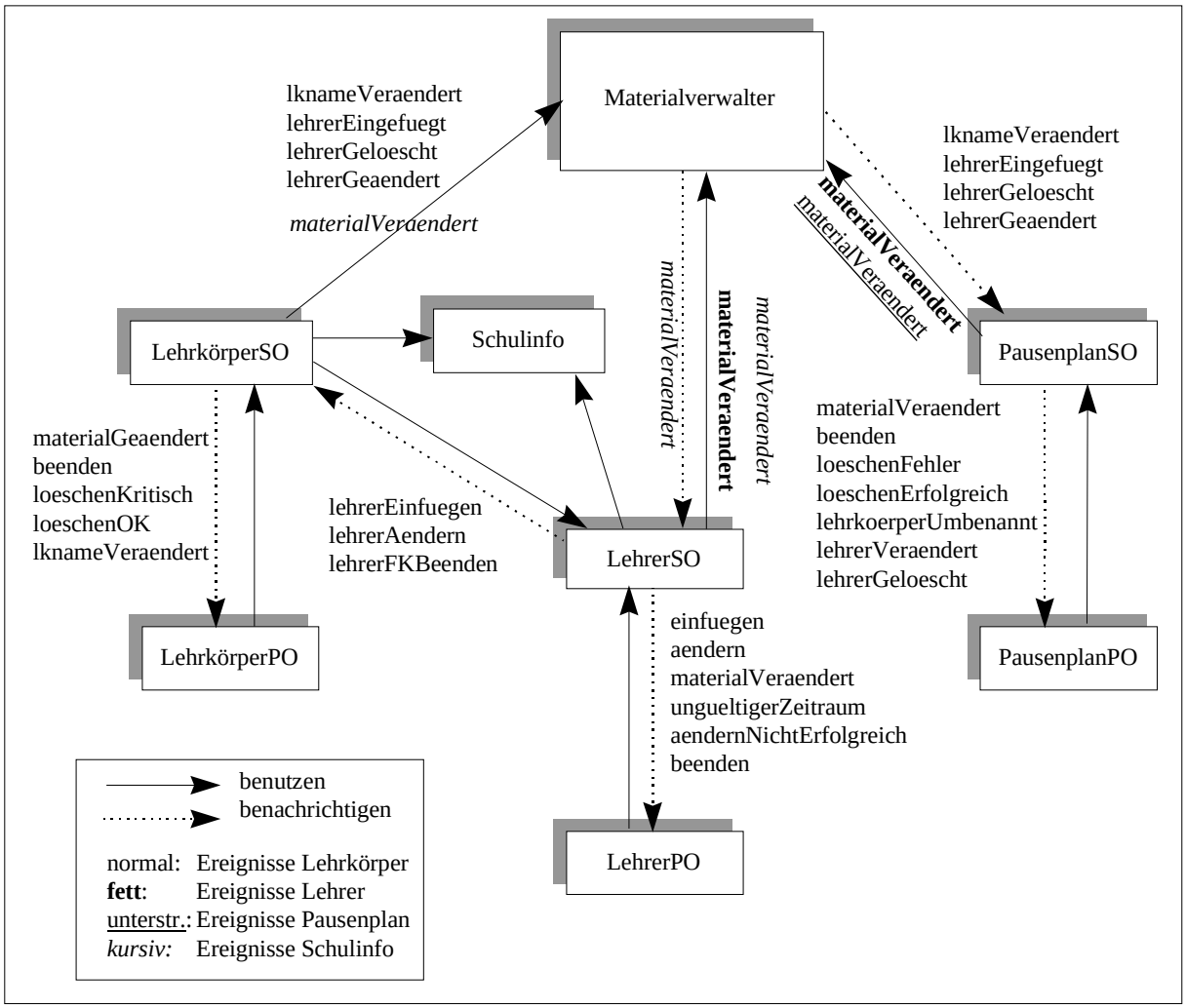


Abb. 35: Zusammenspiel der Werkzeuge

6 Bewertung von DST

Dieses Kapitel resultiert aus unserer Arbeit mit DST während der Erstellung dieser Studienarbeit. Die Bewertung ist daher zum Teil subjektiver Art und ohne Vergleich mit anderen verteilten Smalltalksystemen entstanden. Wir haben uns bemüht, zwischen den verteilten Erweiterungen und dem eigentlichen Smalltalksystem VisualWorks zu trennen. Da DST jedoch eine Erweiterung von VisualWorks ist, ist eine Differenzierung nicht immer sinnvoll. Wir haben dementsprechend auch Aspekte bewertet, die bereits in der nicht verteilten Smalltalk Version enthalten sind und machen dies im Text kenntlich.

- Smalltalk mit einer CORBA-Implementation zu versehen ist ein für Smalltalk-Entwickler wichtiger Schritt zur Erstellung verteilter Anwendungen. Dafür geht ParcPlace mit DST den Weg, das Smalltalk System VisualWorks zu erweitern. Das führt dazu, das Probleme, die bereits in VisualWorks bestanden, in DST weiter existieren. Dazu gehört z.B. der schlecht konfigurierbare Debugger und der Inspector, mit dem es nicht möglich ist, sich permanent aktuelle Werte anzeigen zu lassen. Der Vorteil besteht darin, daß selbst in den für die Verteilung zuständigen Objekten debugt werden, und daß man die Funktionsweise der Verteilung durch Studieren des Codes verstehen kann. Außerdem bekommt der Entwickler eine Umgebung in die Hand, die er bereits kennt.

- Ein schwerwiegender Nachteil von DST liegt in der Performance. Er resultiert daraus, daß der Code für die Verteilung in Smalltalk implementiert wurde und interpretativ mit viel Overhead abgearbeitet wird. Die lokalen Stellvertreterobjekte (Stubs) verstehen selbst keine der Nachrichten des Objektes, das sie vertreten. Sie fangen alle Methodenaufrufe als unbekannt ab, stellen fest, ob das eigentliche Objekt diese Methode besitzt, erstellen einen Stream, in den die Nachricht mit allen Parametern eingepackt wird und rufen den Serverstub mit diesem Stream auf. Dort wird er entpackt und die Nachricht weitergereicht. Wir vermuten, daß es für die Geschwindigkeit wesentlich besser gewesen wäre, diesen Vorgang nicht in interpretiertem Smalltalkcode zu implementieren. Diesen Code zu verfolgen macht auch sehr wenig Sinn. Lokale Aufrufe geschehen ohne großen Zeitverzug, daher ist es in Anwendungen angebracht, entfernte Objekte mit Hilfe des Lifecycle-Service an den Ort zu verschieben, an dem sie gebraucht werden. Dies widerspricht jedoch der örtlichen Transparenz, die CORBA zur Verfügung stellt. Daher haben wir von dieser Möglichkeit keinen Gebrauch gemacht.

- Den Local RPC Testing Modus können wir nur für sehr einfache, kurze Testzwecke empfehlen, da zum einen die Performance deutlich sinkt (tatsächlich ist sie schlechter als im echten verteilten Modus): Während bei echter Verteilung nur das mit dem für die Verteilung notwendigen Overhead aufgerufen wird, was auch entfernt ist, wird bei eingeschaltetem Local RPC Testing jeder Aufruf als potentiell entfernt bearbeitet. Zum anderen führt das Setzen von Breakpoints mittels *self halt* häufig zu Abstürzen. Manchmal kann der Entwickler die Kontrolle durch die Tastenkombination Meta-Alt-C zurück erlangen, wird aber auch dann häufig mit einer Menge an Dialogboxen konfrontiert, die er erst schließen muß. Und auch dabei zeigt das System ein sehr instabiles Verhalten.

- Der IDL-Generator und das Interface Repository stellen weitere Schwachstellen des Systems dar: Der IDL-Parser kommt nicht mit dem Code zurecht, den der IDL-Generator generiert, wenn als Datentyp für Rückgabewert oder Parameter *sequence*<...> angegeben wird. Der Entwickler muß selbst noch eine Typendefinition mittels *typedef* einführen. Der für Parameter und Rückgabewert als Default eingestellte Typ *SmalltalkObject* kann zu

Laufzeitproblemen führen. SmalltalkObject heißt das Interface für die Klasse *Object* in Smalltalk. Die Probleme können durch die Verwendung des IDL-Typen *any* anstelle von *SmalltalkObject* umgangen werden. Leider hat der Lockingmechanismus des Interface Repositories Fehler, die dazu führen, daß es gelegentlich nicht möglich ist, die IDL Definitionen zu öffnen. Die Fehlermeldung gibt bei einem Versuch an, daß das Interface von dem eigenen Rechner benutzt wird. Dieses Problem läßt sich nur durch Stoppen des ORBs und einer vollständigen Initialisierung beheben. Für den Entwickler ist es auch sehr hinderlich, daß sich IDL Module weder ein- noch ausfilen lassen. Dies kann für den Austausch von IDL-Modulen zwischen Entwicklern sinnvoll sein. Im Normalfall ist dies aber kein Nachteil, da die Interfaces bei eingeschaltetem ORB und Zugriff auf einen zentralen NamingService automatisch aktualisiert werden.

- Die Fehlermeldungen sind leider häufig zu allgemein und helfen bei der Fehlersuche nur unzureichend weiter. Ein Tip: Funktioniert eine Anwendung lokal, verteilt aber nicht, sollte vor allem ein Ereignis nicht beim Empfänger ankommen, so ist meistens eine im IDL-Interface fehlende oder unkorrekt spezifizierte Methode die Ursache.

- Viele Lösungen dieser Probleme hätten wir gerne in der Dokumentation gefunden. Diese ist jedoch außerordentlich spärlich ausgefallen. Die unterstützten CORBA-Services sind sehr schlecht beschrieben, häufig bleibt einem nichts anderes übrig als auszuprobieren und dabei zu debuggen oder sich die Klassen zusammenzusuchen und den Code zu analysieren. Die technischen Hintergründe der Verteilung werden gar nicht erklärt. Die im Workspace beschriebenen Einstellungen für die DST-Settings für den verteilten Zugriff sind in einem Punkt leider nicht ideal. Erklärt sind diese Settings ohnehin nirgendwo. Kurz: Die Dokumentation läßt einen häufig auf halbem Wege stehen.

- Der Komfort von DST für die Entwickler entspricht dem von VisualWorks. Die Tools, die für Verteilung zuständig sind, entsprechen dem Niveau des gesamten Systems, sind demzufolge übersichtlich und funktionell.

- Leider besteht das in VisualWorks übliche Problem der zum Teil sehr langen Wartezeiten, da das System nicht multithreadingfähig ist. In DST treten diese Probleme zusätzlich auf, wenn z.B. der ORB initialisiert wird oder das Interface Repository aktualisiert wird. In verteilten Anwendungen sollte die Gewichtung dieses Problems minimiert werden können, indem der Event-Service, der nach CORBA asynchron sein soll, Anwendung findet. Da ParcPlace jedoch den Event-Service synchron implementiert hat, entstehen prozessübergreifende Wartezeiten. Vor allem hat das zur Folge, daß ein gesamtes Smalltalksystem steht und nicht mehr korrekt beendet werden kann, wenn es aufgrund eines Events in eine Endlosschleife gerät. Weitere potentielle Empfänger erhalten das Ereignis nicht mehr und der Sender muß mittels CTRL-C abgebrochen werden.

- Die in DST enthaltenen Anwendungsbeispiele (vor allem Office, Building, etc.) sind sehr gut. Wir können uns vorstellen, daß sie bereits die Kommunikation bei der Entwicklung verteilter Anwendungen von mehreren Entwicklern stark vereinfachen und eine gute Grundlage für weitere Anwendungen sind.

- Die Implementation des Presentation/Semantic-Splits sieht vor, daß die Namen der Semantic- und Presentation-Klassen sich nur durch die Endung SO bzw. PO unterscheiden. Eine Umbenennung in z.B. FK/IAK ist dadurch unmöglich.

7 Ausblick

Bisher wurde beschrieben, welche Konzepte und Tools erstellt wurden, um die Problematik von verteilten Anwendungen in den Griff zu bekommen. Dieses Kapitel beschreibt Ideen, mit denen Ergänzungen und Verbesserungen durchgeführt werden können.

Die Beispielapplikation, die wir entwickelt haben, ließe sich um einige Komponenten erweitern. Alle Tools ließen sich bzgl. weiterer Benutzungsfreundlichkeit verbessern: Das Lehrertool könnte die Pausen des Lehrers anzeigen. Der Pausenplaner könnte den Lehrkörper darstellen. Zu jeder Pause könnte der vollständige Name des Lehrers gesondert angezeigt werden. Lehrer könnten auf dem Pausenplan per Drag & Drop verschoben oder kopiert werden. Weitere Nachfragen bzgl. kritischer Aktionen wären sinnvoll, wie Abfrage vor Löschen von Lehrern. Ein Kommunikationsmodell wäre zu entwickeln, welches bestimmt, wie Benutzer von Änderungen ihres gerade bearbeiteten Materials durch andere Benutzer erfahren könnten außer durch die plötzliche Änderung der Anzeige wie von Geisterhand. Exceptions sollten durch die Werkzeuge abgefangen werden.

Sicher ist es interessanter, die Erweiterungsmöglichkeiten des Ereignisverwalters und des Materialverwalters zu diskutieren. Im Ereignisverwalter wurde lediglich das Push-Modell verwirklicht (Kapitel 5.3). Hier wäre es denkbar, alternativ das Pull-Modell zu implementieren. Für bestimmte Anwendungszwecke wäre dies sicher besser geeignet als das Push-Modell. Eine Entwicklungshilfe wäre ein Tool, welches den Ereignisverwalter auf dem Bildschirm visualisiert, so daß ein Ereignislogging durchgeführt und angezeigt werden könnte.

Der Materialverwalter könnte ebenso wie der Ereignisverwalter um ein Loggingtool erweitert werden. Mit Hilfe dieses Tools könnten die Anwender sehen, wer wann welche Änderungen an den Materialien vorgenommen hat (ggf. auch mit Kommentar). Dieses Tool könnte selbst um eine Visualisierung des Materialverwalters erweitert werden, die anzeigt, welche Materialien zur Verfügung stehen. Auf dieser Basis wäre eine Integration in die Oberfläche von DST vorstellbar, so daß Verweise auf Materialien per Drag & Drop erhältlich sind. Der Materialverwalter könnte alle Materialien mit vom Benutzer vorgegebenen natürlichsprachlichen Namen versehen und die Materialien beim CORBA Naming Service anmelden. Dann wären auch Materialhierarchien denkbar. Weiterhin ist eine Replikation oder Fragmentierung des Materialverwalters aus Performance- und Verfügbarkeitsaspekten zu diskutieren.

Da DST die CORBA-Services Concurrency Control und Transaction anbietet, könnten damit die vorgestellten Konzepte um einen umfangreichen Transaktions- und Lockingmechanismus ergänzt werden.

Diskussionswürdig wäre es, anstelle der einfachen Aggregationsbeziehungen bzw. der gegenseitigen Kenntnisnahme von Materialien den Link-Service von DST zu verwenden. Dieser wird in DST bereits zur Verwaltung der Elemente in den Office-, Building- und anderen Aggregationsfenstern eingesetzt. Wir halten die Verwendung dieses Service in unseren Tools jedoch für zu aufwendig.

Für Anregungen und Kritik sowie Fragen sind wir unter folgenden E-Mail Adressen erreichbar:

Andreas Felten 2felten@informatik.uni-hamburg.de
Christian Langmann 2langman@informatik.uni-hamburg.de

Abbildungsverzeichnis

Literaturverzeichnis

[BÜCK95] Bücker, Matthias C.; Geidel, Joachim; Lachmann, Matthias F.: Programmieren in Smalltalk mit VisualWorks, 2. Erweiterte Auflage, Berlin, et. al. 1995

[DSIP96] VisualWorks Distributed Smalltalk IDL Programmer's Reference, ParcPlace-Digitalk, Inc., Sunnyvale, CA 1996

[DSPR96] VisualWorks Distributed Smalltalk Programmer's Reference, ParcPlace-Digitalk, Inc., Sunnyvale, CA 1996

[DSUG96] VisualWorks Distributed Smalltalk User's Guide, ParcPlace-Digitalk, Inc., Sunnyvale, CA 1996

[GAMM95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns, 5. Auflage, New York, et al. 1995

[GATZ96] Gatzham, Michael; Kyriakides, Valentino: Konstruktion interaktiver Software nach der Methode WAM in Smalltalk am Beispiel von VisualAge und VisualWorks, Studienarbeit, Fachbereich Informatik, Universität Hamburg, Hamburg 1996

[GELL96] Gellert, Olaf; König, Michael: Vorlesungskommentar des Fachbereichs Informatik Wintersemester 96/97, Universität Hamburg 1996

[KILB94] Kilberth, Klaus; Gryczan, Guido; Züllighoven, Heinz: Objektorientierte Anwendungsentwicklung, 2. Verbesserte Auflage, Braunschweig/Wiesbaden 1994

[MITT97] Mittendorfer, Josef: Smalltalk, 2. Auflage, Bonn 1997

[OMAG92] Object Management Architecture Guide: OMG Document 92.11.1 Revision 2.0, 1992

[ORFA96] Orfali, Robert; Harkey, Dan; Edwards, Jeri: The Essential Distributed Objects Survival Guide, New York, et.al. 1996

[ROOC96] Roock, Stefan; Wolf, Henning: Konzeption und Implementierung eines Reaktionsmusters für objektorientierte Softwaresysteme, Studienarbeit, Fachbereich Informatik, Universität Hamburg 1996

[WULF95] Wulf, Martina: Konzeption und Realisierung einer Umgebung zur Koordination rechnergestützter Tätigkeiten in kooperativen Arbeitsprozessen, Diplomarbeit, Universität Hamburg 1995