Prüfer: Prof. K. Rothermel

Betreuer: Dipl.-Inform. F. Sembach

begonnen am: 18. November 1993

beendet am: 17. May 1994

CR-Klassifikation: D3.4, C.2.4, D.1.5

Diplomarbeit Nr. 1109

Programmgenerator für Shared Workspace-Objekte

Markus Straßer

Fakultät Informatik
Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Zusammenfassung

Computer Supported Cooperative Work (CSCW) unterstützt Arbeitsgruppen auf vielfältige Weise beim Erreichen ihrer Ziele. Dieses erfolgt durch den Austausch und das gemeinsame Bearbeiten von Daten.

Die Entwicklung von CSCW-Anwendungen ist bis jetzt recht mühsam. Existierende Toolkits vereinfachen die Entwicklung, Kooperation zwischen verschiedenen Anwendungen ist jedoch nicht möglich. Der innerhalb des Projektes *TEATIME*[1] zu entwickelnde Shared Workspace (SWS) soll gerade diese Kooperation ermöglichen. Er ist konzeptionell ein Speicher für Objekte, auf den von mehreren Applikationen zugegriffen werden kann.

In dieser Diplomarbeit wird zuerst ein schon existierendes Modell des SWS erläutert und erweitert. Anhand dieses Modelles wird ein Objektmodell für im SWS enthaltene Objekte, welche mittels der Programmiersprache C++ realisiert werden sollen, entwickelt. Die hierbei dominanten Schwerpunkte sind die Fragen nach der Verwaltbarkeit beliebiger Objekte und nach der globalen Identifikation der Objekte im SWS.

Objekte im SWS werden als Replikate bei den Anwendungen gehalten. Dadurch ergibt sich die Notwendigkeit der Verteilung geänderter Daten der Objekte. Dazu werden Mechanismen zur Verteilung dieser Änderungen mittels Verteilung der geänderten Daten (Datenverteilung) und mittels Verteilung der ändernden Methoden (Methodenverteilung) entwickelt.

Die Entwicklung eines einfachen, verteilten Sperrprotokolls ist Thema eines weiteren Kapitels. Im vorletzten Kapitel werden Mechanismen zur Objektverwaltung im SWS vorgestellt.

Das letzte Kapitel befaßt sich mit dem zu entwickelnden Generator für SWS-Objekte. Ziel ist die Generierung von SWS-Objekten aus einer an C++ angelehnten Definitionssprache für SWS-Objekte. Nach der Definition der Synatx dieser Sprache wird die Arbeitsweise des Generators und die durch den Generator durchzuführenden Transformationen einer zu bearbeitenden SWS-Objektdefinition beleuchtet.

[1]TEATIME Enables A Team to Interactively Manage and Edit objects

Inhaltsverzeichnis

1 Einführung und Aufgabenstellung **1**

1.1 Einführung 1

1.2 Die Aufgabenstellung 3

2 Der Shared Workspace **4**

2.1 Das ideale Modell 4

2.1.1 Der SWS als Objekt-Speicher 4

2.1.2 Transparenz der Objektänderungen 5

2.1.3 Konsistenz im SWS 6

2.1.4 Objektpersistenz 7

2.1.5 Die Sitzung 7

2.2 Das reale Modell 7

2.2.1 Die Anwendungsschnittstelle 7

2.2.2 Konsistenz im SWS 8

2.2.3 „Lesende“ und „sehende“ Methoden 8

2.2.4 Datenhaltung im SWS 9

2.2.5 Replikation von Datenänderungen 12

2.2.6 Die Replikationscharakteristik 14

2.3 Die Architektur des Shared Workspace 15

2.3.1 Der Object Manager 15

2.3.2 Der Event Manager 17

2.3.3 Der Session Manager 17

2.3.4 Der Security Manager 18

2.3.5 Der Transaction Manager 18

2.4 Der Programmgenerator und das Basissystem 18

2.5 Vergleich mit anderen Ansätzen 18

3 Das Objektmodell 20

3.1 Objektorientierte Programmierung 20

3.1.1 Was ist ein Objekt 20

3.1.2 Klassen 21

3.1.3 Vererbung 22

3.2 Objekte im SWS 23

3.2.1 Die Daten des SWS-Objektes 23

3.2.2 Das Generische Objekt 24

3.2.3 Die Object-Identifier und der Aufbau des Objektes 26

3.2.4 Resultierende Einschränkungen 33

3.3 SWS-Objekte in der Applikation 34

4 Replikation 35

4.1 Allgemeine Aspekte der Verteilung der Änderungen von Objekten 35

4.2 Die Datenverteilung 36

4.2.1 Ermittlung der zu verteilenden Daten 36

4.2.2 Form der verteilten Daten 37

4.2.3 Verteilung durch den OM 38

4.2.4 Unlock während Datenverteilung 39

4.2.5 Eingehende Datenverteilungen 39

4.3 Die Methodenverteilung 39

4.3.1 Eindeutige Identifikation der Methode 40

4.3.2 Parameter der Methoden 40

4.3.3 Rückgabewert der SWS-Methoden 42

4.3.4 Verteilung der Methode 43

4.3.5 Probleme der Methodenverteilung 44

4.3.6 Zusätzliche Datenverteilung 44

5 Locking 48

5.1 Ein einfacher Sperrmechanismus 48

5.2 Verklemmungen 48

5.3 Schnittstelle zur Anwendung 50

5.3.1 Sperroperationen 50

5.3.2 Benötigte Sperren bei Methodenaufrufen 53

5.3.3 Automatische Anforderung von Locks 54
5.4 Der Lock-Algorithmus 54
5.4.1 Verteiltes Locking 54
5.4.2 Asynchrones Locking 55

6 Die Objektverwaltung 58
6.1 Die Aufgaben des Objektmanagers bei der Objektverwaltung 58
6.2 Kodierung von Objekten 59
6.3 Die Objektfabrik 59
6.4 Objektpersistenz 62
6.5 Benennung von Objekten 63
6.6 Neuverteilung von Objekten 64
6.6.1 Die Verteilung auf einen anderen Knoten 65
6.6.2 Anforderung eines Objektes durch die Anwendung 66
6.7 Neue SWS-Objekte 67
6.8 Löschen von Objekten 68
6.9 Beenden einer Applikation 68
6.10 Pointer-Elementarobjekte und die Objektverwaltung 69

7 Der Programmgenerator 70
7.1 Aufbau einer SWS-Klasse 70
7.1.1 Die Klassendefinition 70
7.1.2 Die Implementation der Methoden einer Klasse 76
7.2 Die Replikationscharakteristik 76
7.2.1 Die gewünschte RC eines Methodenaufrufes 76
7.2.2 Die tatsächliche RC eines Methodenaufrufs 78
7.3 Die Arbeitsweise des Generators 79
7.3.1 Die Bearbeitung des Header-Files 79
7.3.2 Die Bearbeitung des Implementationsfiles 82
7.4 Aufbau der Navigationsstruktur 85
7.5 Vom Generator erzeugte Methoden und Funktionen 86
7.5.1 Methoden für die Verwaltung 87
7.5.2 Frei zugängliche Methoden 87
7.6 Benutzung der SWS-Objekte 88

8 Ausblick **89**

Anhänge **90**

A Verwendung der Navigationsstruktur **90**

B Beispiel zur Methodenverteilung **92**

C Die Lockalgorithmen **95**

C.1 Der Lockalgorithmus beim Initiator 95

C.2 Der Lockalgorithmus auf einem Remote-Knoten 97

C.3 Ankommende Prelock-Anforderung 97

C.4 Der Unlockalgorithmus 98

D Algorithmen zur Objektverwaltung **99**

D.1 Algorithmus zur Verteilung eines Objekts 99

D.1.1 Der Algorithmus 99

D.1.2 Setzen des Distribution-Locks 101

D.1.3 Ankommender Objekterstellungsauftrag 102

D.2 Anforderung eines existierenden Objektes 103

E Grammatik der SWS-Klassen **105**

E.1 Die Klassendefinition 105

E.2 Die Implementation 108

F Zusätzliche Members der SWS-Klassen **109**

G Die Erweiterungen der Methoden **111**

G.1 „Sehende“ Methoden 111

G.2 Nicht verteilbare Methoden 112

G.3 Verteilbare Methoden 112

Abbildungsverzeichnis **115**

Tabellenverzeichnis **117**

Literaturverzeichnis **118**

Kapitel 1

Einführung und Aufgabenstellung

1.1 Einführung

Im Projekt *TEATIME*[1] wird ein kooperativer Multimedia-Editor entwickelt. Diese Art Anwendung gehört der Klasse der CSCW[2]-Applikationen an. Im Gegensatz zu gewöhnlichen Ein-Benutzer-Anwendungen, bei denen ein Anwender ein Problem bearbeitet und Zusammenarbeit höchstens sequentiell stattfinden kann, wird bei CSCW-Anwendungen ein Problem von mehreren Personen gleichzeitig kooperativ im Team bearbeitet (Sitzung). Der durch die enge Zusammenarbeit (und den sich dadurch ergebenden intensiven Gedankenaustausch) erhoffte Effekt ist die Erzielung qualitativ besserer Ergebnisse in kürzerer Zeit.

Mögliche Anwendungsgebiete der CSCW sind alle Arten von Editoren (Texteditoren, Grafikeditoren, CAD-Anwendungen, ...), Programme zur Unterstützung der Ideenfindung (Brainstorming) u.a. Einige Beispiele werden in [Wir93, EGR91, GS87, KLL+91] beschrieben.

Die Anwendungen realisieren im Prinzip einen gemeinsamem Arbeitsbereich (Shared Workspace, SWS), auf welchen jedes Mitglied des Teams Zugriff hat. Die einzelnen Realisierungen sind sehr unterschiedlich. Eine Klassifizierung der hierbei möglichen Architekturen findet man in [SM93]. Allen gemeinsam ist, daß der SWS in jeder Applikation komplett neu implementiert wird, worauf ein nicht geringer Teil der Entwicklungsarbeit entfällt. Neuere Entwicklungen [CMB+90, RG92, MLF92, PHR90] bieten deshalb Toolkits zur Entwicklung von CSCW-Programmen, die dem Applikationsentwickler diese Arbeit in gewissem Umfang abnehmen.

Die kooperative Bearbeitung von Problemen ist jedoch bei den bisherigen Anwendungen auf die jeweilige Anwendung eingegrenzt — ein CSCW-Texteditor ermöglicht gemeinsame Arbeitssitzungen bisher eben nur mit seinesgleichen, eine Kooperation mehrerer verschiedener Applikationen ist nicht vorgesehen. Auch die Toolkits bieten bisher keine

[1] TEATIME Enables A Team to Interactively Manage and Edit objects

[2] Computer Supported Cooperative Work

wesentlich darüber hinausgehende Hilfe. Die Aufhebung dieser Beschränkung durch eine Verallgemeinerung des SWS-Begriffes wäre aber durchaus wünschenswert, da die Palette der möglichen Anwendungen sehr breit ist. Ein Beispiel ist ein Team, welches ein DTP[3]-Dokument gemeinsam bearbeitet, das außer Text auch noch Grafiken enthält. Um eine der verwendeten Grafiken zu ändern, gibt es prinzipiell zwei Möglichkeiten. Entweder besitzt der DTP-Editor die notwendigen Funktionen, um die Grafik zu bearbeiten, oder einer der Sitzungsteilnehmer muß die Grafik mit einem externen Programm (eventuell auch ein CSCW-Programm) bearbeiten. Beide Lösungen sind nicht optimal. Die erste nicht, da es unnötig erscheint, in das DTP-Programm einen erstklassigen Grafikeditor einzubauen obwohl es auf dem Markt genug andere Grafikeditoren gibt. Die zweite Lösung ist insofern ungünstig, da das Ändern der Grafik außerhalb der Arbeitsgruppe geschieht und dadurch zum Beispiel das Ergebnis erst nach der kompletten Änderung für alle anderen sichtbar wird. Die günstigste Lösung wäre der gemeinsame Zugriff von DTP-Programm und Grafikeditor auf die Grafik.

Ein verallgemeinerter SWS muß also in der Lage sein, die problemrelevanten Daten verschiedener Applikationen aufzunehmen und diese dann allen Teilnehmern am SWS zur Verfügung zu stellen. Weiterhin muß er die Persistenz und die Konsistenz der Daten gewährleisten. Die bis hierher aufgeführten Forderungen werden auch von herkömmlichen Datenbanken erfüllt. Datenbanken sind von ihrem Aufbau her jedoch nicht für kooperative Arbeit ausgelegt, sie gehen von konkurrierenden Zugriffen auf die Daten aus. Darüberhinaus müssen Änderungen, die ein Anwender an SWS-Daten gemacht hat, überall dort bekannt gegeben werden, wo dies von Interesse ist. Wird zum Beispiel ein Text-Absatz geändert, so sollte diese Änderung beinahe gleichzeitig allen anderen Applikationen, die den dazugehörigen Text bearbeiten, bekannt gemacht werden, damit deren Anwender keine „alten“ Daten zu sehen bekommen.

Im Rahmen des Projekts *TEATIME* soll ein solcher applikationsunabhängiger SWS entwickelt werden, auf den dann unterschiedliche Anwendungen zur kooperativen Manipulation multimedialer Daten aufsetzen können. Ein Konzept wurde in vorausgehenden Arbeiten schon entwickelt und vorgestellt [Kaba93, SR93, Sem93]. Das Ziel dieser Arbeit und einer damit verbundenen Studienarbeit ist die Realisierung des SWS in einer Art und Weise, die es dem Applikationsprogrammierer erlaubt, sich auf die Programmierung der Applikation zu konzentrieren. Die Verwaltung der gemeinsamen Daten sollen ein Basissystem (Laufzeitsystem) und ein Programmgenerator übernehmen.

In dieser Arbeit wird als erstes das Konzept des zu realisierenden SWS kurz vorgestellt und mit anderen Ansätzen verglichen. Aus diesem Konzept werden dann der genaue Aufbau und die Funktionalität des SWS und die Anforderungen des Programmgenerators an das in einer anderen Arbeit zu erstellende Basissystem entwickelt.

[3]DeskTopPublishing

1.2 Die Aufgabenstellung

Die Aufgabenstellung der Diplomarbeit

Programmgenerator für Shared Workspace-Objekte

lautet :

Im Rahmen des Projektes *TEATIME* wird eine Software-Architektur für einen objektorientierten gemeinsamen Arbeitsbereich (Shared Workspace, SWS) entwickelt. Der SWS dient als konzeptioneller Speicher für Objekte, die von kooperativen Anwendungen manipuliert werden. Die Objektstruktur wird von einem Applikationsprogrammierer festgelegt, für den sich die Entwicklung einer kooperativen Anwendung nicht wesentlich von der einer Einbenutzeranwendung unterscheiden soll. Die Verwaltung von Objektreplikaten auf verschiedenen am SWS beteiligten Rechnern übernimmt ein SWS-Basissystem, das in einer anderen Arbeit entwickelt wird. Der Programmierer definiert SWS-Objekte in der Form einer C++ Klassenbeschreibung, die um Konstrukte zur Steuerung der Replikationsverwaltung erweitert ist.

In dieser Arbeit soll ein System zur Umsetzung von SWS-Objektdefinitionen in C++-Objekte entwickelt werden. Zur Steuerung der Replikationsverwaltung soll die Angabe einer sogenannten Replikationscharakteristik (RC) dienen. Ein Teil der RC gibt die Art der Replikationsinformation an, d.h. ob zum Update von Replikaten die neuen Daten oder die verändernden Methodenaufrufe verteilt werden. Ein zweiter Teil beeinflußt die Granularität der Replikation. Es ist zu untersuchen, wie Direktiven zur Steuerung der Replikationsverwaltung als möglichst organische Erweiterung der Sprache C++ realisiert werden können. Vom SWS-Basissystem wird eine Menge von Basisobjekten angeboten, aus denen die SWS-Objekte zusammengebaut werden. SWS-Objekte werden als Spezialisierung eines ebenfalls vom Basissystem gelieferten generischen Objekts definiert. Die Anforderungen des Programmgenerators an das Basismodell sind in Zusammenarbeit mit dem Bearbeiter der 'Basismodell-Studienarbeit' zu definieren.

Der Softwareanteil der Arbeit ist auf IBM RS/6000 Workstations unter AIX in C++ zu implementieren. Die Richtlinien der Abteilung Verteilte Systeme für den Softwareentwurf sind zu befolgen[4].

[4]Die Richtlinien schreiben die Verwendung von definierten Prefixes für Bezeichner vor. Diese Prefixes werden in dieser Ausarbeitung aus Gründen der Lesbarkeit weggelassen.

Kapitel 2

Der Shared Workspace

In diesem Kapitel wird das in [Kaba93][Sem93] entwickelte Konzept des Shared Workspace (SWS) vorgestellt. Der Schwerpunkt hierbei sind die grundlegenden Fragen der Datenhaltung im SWS, darüber hinausgehende Aspekte wie Sessionmanagement und Transaktionsverwaltung werden nur angeschnitten, sie sind Thema weiterer Arbeiten.

2.1 Das ideale Modell

2.1.1 Der SWS als Objekt-Speicher

Der SWS ist konzeptionell ein Speicher, in dem Objekte abgelegt werden können. Ein Objekt ist hier ein Objekt im Sinne der objektorientierten Programmierung. Es besteht aus Daten und Methoden, durch die die Daten gelesen und manipuliert werden können. Die Objekte stehen prinzipiell verschiedenen Anwendungen (auch auf verschiedenen Rechnern) zur Verfügung. Der SWS wird durch SWS-Agenten realisiert, von denen es für jede Applikation genau einen gibt. Der SWS-Agent repräsentiert für die Applikation den gesamten SWS, die Gesamtheit der SWS-Agenten realisiert den SWS.

Die Verfügbarkeit *eines* Objektes für *mehrere* Applikationen bedingt eine Trennung des Objekts in darstellungsspezifische Aspekte und den eigentlichen Objektzustand. Die darstellungsspezifischen Aspekte werden in ein sogenanntes View-Objekt abgebildet, die Zustandsbeschreibung wird in dem eigentlichen SWS-Objekt modelliert. Abbildung 2.1 zeigt die sich daraus ergebende Grobarchitektur.

Die Notwendigkeit der Trennung zwischen SWS-Objekt und View-Objekt wird anhand des schon in [Kaba93] aufgeführten Beispiels des Würfels deutlich. Ein Würfel kann u.a. durch seine acht Ecken beschrieben werden. Die Darstellung, in der ein Anwender diesen Würfel zu sehen bekommt, kann von Applikation zu Applikation verschieden sein. So wird beim Grafik-Editor eventuell ein Drahtmodell angezeigt während das CAD-Programm denselben Würfel als Auf- und Grundriß darstellt (s. Abbildung 2.2). Das Objekt im SWS enthält

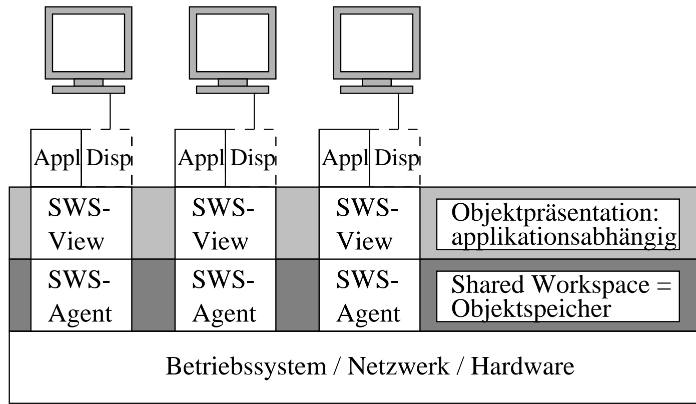


Abbildung 2.1: Grobarchitektur des SWS

also im Falle des Würfels nur die ihn beschreibenden Daten, für die Darstellung sind die unterschiedlichen View-Objekte zuständig.

2.1.2 Transparenz der Objektänderungen

Änderungen an Objekten, die für alle am SWS teilnehmenden Applikationen „sichtbar" sind, sollen (eventuell abhängig vom View-Objekt) einen direkten Einfluß auf ihre Darstellung haben. Dies ist kein Problem, wenn das Objekt durch das anzeigende View-Objekt geändert wurde. Das View-Objekt weiß, was geändert wurde und wie es darauf zu reagieren hat. Problematischer ist die Änderung des SWS-Objektes für alle anderen mit dem geänderten SWS-Objekt verbundenen View-Objekte. Diese müssen die Änderung mitge-

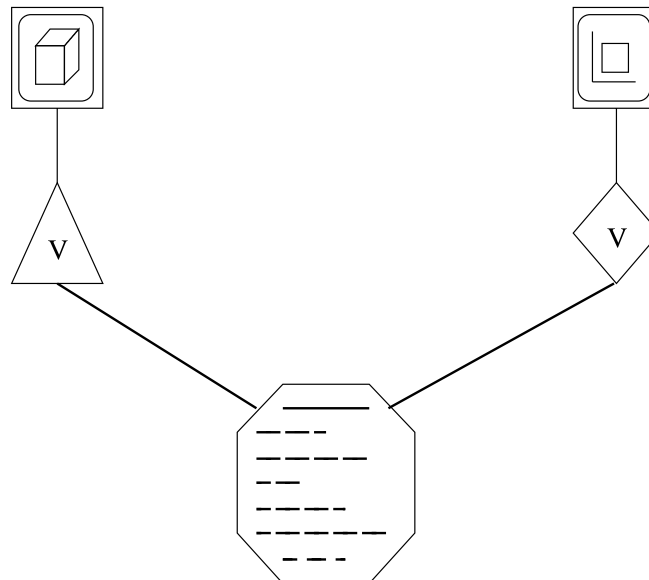


Abbildung 2.2: Trennung in Daten- und Viewobjekte

teilt bekommen. Es muß deshalb eine Meldeschnittstelle existieren, über die das SWS-Objekt den View-Objekten der unterschiedlichen Applikationen mitteilt, daß es geändert wurde. Die Information die über diese Schnittstelle geht kann sehr unterschiedliche Qualitäten haben. Man kann dies am Beispiel des Würfels verdeutlichen. Angenommen die Daten des Würfels beschreiben dessen acht Ecken in einem kartesischen Koordinatensystem. Eine Methode des Objektes Würfel führt eine Verschiebung des Würfels im Raum durch. Die Meldung, die nach der Ausführung der Methode an das View-Objekt versandt wird, kann im Prinzip folgende drei Formen annehmen:

- „Das Objekt hat sich geändert.“ Diese wohl allgemeinste Änderungsmeldung unterstützt das View-Objekt in keinster Weise. Es hat nur die Möglichkeit, das komplette Objekt neu darzustellen.
- Acht mal „Die Ecke … hat sich geändert“. (Dies entspricht der allgemeinen Meldung, daß sich ein bestimmtes Unterobjekt geändert hat). Diese Meldungen sind für das View-Objekt auch nicht sonderlich sinnvoll. Erst nachdem alle acht Meldungen angekommen, also der Würfel komplett verschoben wurde, ist das Objekt wieder ein Würfel, alle anderen Zwischenzustände ergeben die tollsten Formen, nur keinen Würfel.
- „Der Würfel wurde … verschoben“. Diese Meldung ist die sinnvollste der bisher aufgeführten, das View-Objekt weiss genau, wie es darauf zu reagieren hat.

Der Vorteil der Meldung der ausgeführten datenändernden Methoden an das View-Objekt zeigt sich vor allem bei komplexeren Objekten, da hier anhand der gemeldeten Methode entschieden werden kann, ob eventuell nur Teile des Objektes neu dargestellt werden müssen.

Bei genauer Betrachtung der beiden letzten Meldungen fällt auf, daß sie sich eigentlich nicht sonderlich unterscheiden. Auch die Meldung, daß sich eine Ecke geändert hat, ist eine Meldung einer datenändernden Methode, nur daß sich diese Meldung nicht auf eine Methode des Objektes „Würfel“ bezieht sondern auf in ihm enthaltene Objekte „Punkt“. Die Methoden der Ecken werden von der Methode „verschieben“ des Würfels aufgerufen, sie stehen in der Aufrufhierarchie weiter unten. Offensichtlich werden die Meldungen an die View-Objekte desto unpräziser, je weiter unten sie in der Aufrufhierarche entstehen. Es ist daher wünschenswert, daß die Änderungsmeldungen der datenändernden Methoden auf möglichst hoher Stufe („möglichst bald“) entstehen.

2.1.3 Konsistenz im SWS

Das Problem der Mitteilung der Änderungen an die View-Objekte ist nur die eine Seite der Datenänderungen an Objekten. Der wesentlich schwierigere Punkt sind mögliche gleichzeitige Zugriffe von verschiedenen Anwendungen auf das gleiche Objekt. Um die

Konsistenz der Daten zu wahren, müssen hier, ähnlich wie bei Datenbanken, Sperrmechanismen bzw. Transaktionskonzepte entwickelt werden. Dies ist Thema weiterer Arbeiten, ausführlichere Information hierzu findet man in [Kaba93].

2.1.4 Objektpersistenz

Als Objektspeicher ist der SWS nicht nur für die Verwaltung der Objekte zur Laufzeit zuständig. Die Lebensdauer von SWS-Objekten ist prinzipiell nur durch explizites Löschen des Objektes durch eine Anwendung begrenzt. Der SWS muß daher für die persistente Sicherung der Objekte auf Sekundärspeicher sorgen. Darüberhinaus muß dem Anwender die Benennung eines Objektes möglich sein, damit auf das Objekt in späteren Sitzungen unter diesem Namen wieder zugegriffen werden kann. Der Mechanismus der Objektspeicherung und -benennung wird in Kapitel 6 vorgestellt.

2.1.5 Die Sitzung

Unter einer Sitzung versteht man die kooperative Bearbeitung eines Problems durch mehrere Anwender unter Verwendung eventuell verschiedener Applikationen. Eine Sitzung ist abstrakt gesehen nichts anderes als eine Menge von Anwendungen, die an der Sitzung teilnehmen und einer Menge von Objekten, welche in der Sitzung bearbeitet werden. Eine Sitzungsverwaltung hat zum Beispiel die Aufgabe, daß neu zur Sitzung hinzukommende Teilnehmer nicht explizit angeben müssen, welche Objekte sie bearbeiten wollen, sondern daß sie diese automatisch zur Verfügung gestellt bekommen. Der Begriff der Sitzung ist dabei sehr allgemein (ein Anwender in mehreren Sessions, ein Objekt in mehreren Sessions), auch er ist Objekt weiterer Arbeiten. Eine etwas genauere Betrachtung findet man wiederum in [Kaba93][Sem93].

2.2 Das reale Modell

2.2.1 Die Anwendungsschnittstelle

Ein Ziel des SWS ist die für den Anwendungsentwickler möglichst „transparente" Behandlung von SWS-Objekten. Die Ideallösung ist die Definition der SWS-Objekte in der Implementierungssprache der Anwendung. Die SWS-Aspekte können dann von einem speziellen Programmgenerator behandelt werden. Die in dieser Arbeit vorgegebene Sprache ist C++.

Elementarobjekte

Im SWS können keine beliebigen Objekte abgelegt werden, sie müssen darauf vorbereitet sein. Diese Vorbereitung wird durch den zu entwickelnden Programmgenerator vorgenommen. Der Programmgenerator akzeptiert als Eingabe Klassenspezifikationen, die

aus leicht modifiziertem C++ bestehen und generiert daraus vom Compiler verarbeitbares C++. Eine der Modifikationen ist die Beschränkung der Daten-Member eines Objektes. Ein SWS-Objekt darf sich nur aus anderen SWS-Objekten und den sogenannten **Elementarobjekten** des Basissystems zusammensetzen. Diese Elementarobjekte umfassen spezielle Implementierungen der Basistypen von C++, spezielle Array-Objekte (von Basistypen) und einen Pointertyp, der auf SWS-Objekte zeigt (analog zum normalen Pointer in C++). Der Pointertyp wird auch SWS-Pointer genannt. Der Aufbau der SWS-Objekte wird in Kapitel 3, die genaue Syntax der SWS-Objektdefinition in Kapitel 7.1 vorgestellt.

2.2.2 Konsistenz im SWS

Die Entwicklung konsistenzerhaltender Konzepte (z.B. Transaktionskonzept) ist erst für spätere Arbeiten vorgesehen. Eine Implementierung eines SWS ohne solche Mechanismen ist jedoch nicht möglich oder zumindest nutzlos. Aus diesem Grunde wird in dieser ersten Realisierung ein einfaches Sperrprotokoll mit Read- und Write-Locks eingebaut. Diese Locks gelten immer auf ganze Objekte. Möchte eine Applikation an einem Objekt etwas ändern, so muß sie zuerst einen Write-Lock auf dieses Objekt anfordern, möchte sie Daten lesen, so benötigt sie dafür einen Read-Lock. Dieser Sperrmechanismus ist jedoch im höchsten Grade isolierend und behindert die kooperative Arbeit, er ist daher nur als Übergangslösung zu verstehen.

2.2.3 „Lesende“ und „sehende“ Methoden

Wie in 2.1.2 beschrieben, ist eines der Ziele des SWS, daß Änderungen überall möglichst schnell sichtbar werden. Dies verträgt sich überhaupt nicht mit der in 2.2.2 eingeführten Sperrpolitik. Wenn eine Applikation ein Objekt ändert, benötigt sie einen Write-Lock. Die Semantik des Write-Lock schließt nun den Besitz eines Read-Locks auf eben dieses Objekt durch eine andere Applikation aus, d.h. die geänderten Objekte können durch die View-Objekte nicht gelesen werden. Es wird daher bei Zugriffen auf Objekten zwischen „lesenden“ und „sehenden“ Methoden unterschieden. Für einen lesenden Zugriff benötigt man mindestens einen Read-Lock, für einen sehenden Zugriff benötigt man keinen Lock. Damit können die View-Objekte für reine Darstellungszwecke die Objektdaten „sehen“, für alle anderen Zwecke (Berechnungen, Entscheidungen anhand der Objektdaten) werden die Daten weiterhin „gelesen“. Ändernde Methoden sind prinzipiell lesende Methoden. Damit die Read-Locks ihre Bedeutung nicht verlieren, ist der Aufruf sehender Methoden von lesenden SWS-Methoden[1] aus verboten und umgekehrt.

Elementarobjekte bilden hier eine Ausnahme. Auch sie benötigen für schreibende Methoden einen Read-Lock. Bei den anderen Methoden wird jedoch nicht zwischen „sehenden“ und „lesenden“ Methoden unterschieden. Das Verhalten dieser Methoden ist davon abhängig, durch was sie aufgerufen wurden. Bei Aufruf aus „lesenden“ Methoden benötigt

[1] Eine Methode eines SWS-Objektes wird als SWS-Methode bezeichnet

das Elementarobjekt einen Read-Lock, bei Aufrufen aus „sehenden“ Methoden oder von außerhalb des SWS wird kein Lock auf das Elementarobjekt benötigt.

2.2.4 Datenhaltung im SWS

Das ideale Modell beschränkt sich auf die idealisierte Sicht des SWS, welche den SWS als eine verteilte Black-Box betrachtet, in der die SWS-Objekte enthalten sind. Die tatsächliche Verwaltung der Objekte innerhalb dieser Black-Box wird dabei außer acht gelassen. Zwei mögliche Strategien werden im folgenden vorgestellt.

Ein Objekt — ein Ort

Bei diesem Ansatz werden die Daten eines Objektes genau an einem Ort gehalten. Bezogen auf die in Abbildung 2.1 dargestellte Grobstruktur des SWS wird ein Objekt von genau einem SWS-Agenten verwaltet. Die anderen SWS-Agenten enthalten nur einen Verweis auf dieses Objekt. Dieser Verweis kann auf verschiedene Arten realisiert werden. Eine Möglichkeit sind sogenannte Proxy-Objekte [ND92], welche ein Objekt, das bei einem der SWS-Agenten gehalten wird, bei den anderen SWS-Agenten vertreten und die Nachrichten, die an das Objekt geschickt werden, an das originale Objekt weiterleiten. Diese Kommunikation auf Nachrichtenebene entspricht dem Paradigma der objektorientierten Programmierung und ist daher für rein objektorientierte Sprachen eine geradlinige Erweiterung. Für Hybridsprachen wie C++ ist er dagegen nicht so geeignet, da bei diesen der Kontrollfluß nicht rein auf dem Nachrichtenkonzept basiert.

Der Ansatz hat den Vorteil, daß Konsistenz vergleichsweise einfach zu wahren ist, da physikalisch nur ein Objekt vorhanden ist. Änderungen auf den Objekten müssen jeweils nur an dem Ort gemacht werden, wo das Objekt real vorhanden ist, Zugriffskontrolle kann lokal abgehandelt werden.

Der große Nachteil dieser Realisierung der Objekthaltung liegt aber im Lesezugriff auf Objekte. Für jeden dieser Zugriffe muß im allgemeinen ein Zugriff über eine relativ langsame Kommunikationsverbindung (Netzwerk, Inter-Prozess-Kommunikation) erfolgen, was die Lesezugriffe erheblich bremst. Da lesende Zugriffe im SWS durch die View-Objekte sehr häufig sein dürften, wirkt sich dieser Punkt um so negativer aus.

Schreibzugriffe sind natürlich auch nicht schneller als die Lesezugriffe. Durch die geforderte Aktualisierung der Anwender-Displays durch die View-Objekte bei Änderungen an den Objekten müssen vom eigentlichen Objekt zusätzlich die Benachrichtigungen versandt werden, daß sich das Objekt geändert hat, was dann eine Reihe von (ebenso langsamen) Lesezugriffen auf das Objekt zur Folge hat.

Eine möglich Optimierung ist die Migration eines Objektes zu dem SWS-Agenten jenes Anwendungsknotens, auf dem gerade viel auf das Objekt zugegriffen wird. Ob dies in Hinsicht auf den dazu notwendigen Mehraufwand und die Display-Aktualisierungen bei Objektänderung entscheidende Verbesserungen bringt, darf bezweifelt werden.

Ein Objekt — „alle“ Orte

Dieser Ansatz wird auch Datenreplikation [Rot] genannt. Hier wird jedes Objekt bei jedem SWS-Agenten, dessen Anwendung dieses Objekt benötigt, mit den kompletten Daten verwaltet, d.h. ein Objekt ist theoretisch beliebig oft vorhanden (Replikation der Objekte)

Die Wahrung der Konsistenz der Daten ist hier wesentlich schwieriger. Es müssen verteilte Mechanismen entworfen werden, welche die gleichzeitige Änderung eines Objektes durch mehrere Applikationen vermeidet bzw. in konsistenter Weise synchronisiert. Diese Mechanismen sind relativ komplex und benötigen einigen Kommunikationsoverhead.

Die Lesezugriffe hingegen sind bei diesem Ansatz wesentlich effizienter, da (zumindest bei „sehenden“ Methoden) nur lokale Zugriffe notwendig sind.

Die Änderung der Daten eines Objektes sind etwas komplexer. Eine Änderung muß nicht nur an einer Stelle, sondern überall, wo das Objekt existiert, durchgeführt werden („lies eine, schreibe alle“). Die dafür verwendbaren Mechanismen werden weiter hinten besprochen. Die Meldungen einer Objektänderung an die View-Objekte werden in diesem Falle natürlich von den jeweiligen lokalen Objekten gemacht. Diese Meldung und auch das darauf folgende „sehen“ sind lokal und deshalb wesentlich effizienter.

Zusätzlich zu den beiden hier vorgestellten Ansätzen sind beliebige Zwischenstufen denkbar. Ein Objekt könnte mehrfach, aber nicht überall wo es benötigt wird, gehalten werden. Diese würde aber weder die Nachteile der beiden vorgestellten Ansätze eliminieren noch die Vorteile in sich vereinen. Zudem müßten komplexe Verwaltungsmechanismen entworfen werden, die zusätzlichen Aufwand in die Verwaltung einbringen würden.

Vergleich der beiden Ansätze

Um die beiden vorgestellten Ansätze miteiander vergleichen zu können, werden zuerst die reinen Schreib- und Lesevorgänge betrachtet. Die als vorläufig geplante Synchronisation mittels einem einfachen Lockmechanismus wird dabei außer acht gelassen.

Bei Lesezugriffen ist die zweite Lösung wesentlich effizienter. Objekte können lokal gelesen werden, ein Zugriff über langsame Kommunikationskanäle ist nicht notwendig.

Bei Datenänderungen ist der Vergleich etwas schwieriger. Bei der ersten Methode muß im allgemeinen eine Botschaft über die langsamen Kommunikationskanäle zu dem SWS-Agenten geschickt werden, welcher dieses Objekt beherbergt. Die Antwort kommt auf dem gleichen Wege zurück. Der SWS-Agent, bei dem ein Objekt geändert wurde, muß alle mit diesem Objekt verbundenen View-Objekte über die Änderung benachrichtigen. Dies geschieht ebenfalls über die langsamen Kommunikationskanäle, genauso die darauf folgenden Lesezugriffe der View-Objekte auf das Objekt. Bei der zweiten Methode wird nur die Änderung als solches an alle SWS-Agenten verteilt, die das geänderte Objekt lokal besitzen. Also schneidet auch hier der zweite Ansatz deutlich besser ab.

Der Lockmechanismus ist beim zweiten Ansatz allerdings wesentlich komplizierter. Beim ersten Ansatz kann an einer Stelle entschieden werden, ob ein Anwendungsknoten einen Lock auf ein Objekt erhalten kann oder nicht, beim zweiten Ansatz muß dies in Abstimmung mit allen SWS-Agenten geschehen, die dieses Objekt lokal haben. Betrachtet man das übliche Vorgehen bei dieser Art Synchronisationsmechanismus, dann fällt dies allerdings nicht sonderlich ins Gewicht. Im Normalfall wird ein Lock auf die zu bearbeitenden Daten angefordert, dann werden die Daten bearbeitet und erst am Schluß wird der Lock wieder freigegeben. Hat ein Knoten erst einmal einen Lock, dann kann die Anwendung ungehindert auf den Daten arbeiten.

Fazit

Der Vergleich der vorgestellten Ansätze zur Datenhaltung im SWS fällt, zumindest bei theoretischer Betrachtung, zu Gunsten des zweiten Ansatzes aus. Interessant wäre ein praktischer Vergleich durch zwei verschiedene Implementierungen. Dies ist jedoch im Rahmen dieser Arbeit nicht möglich.

Die weiteren Ausführungen werden sich deshalb auf den Ansatz beziehen, in dem ein Objekt auf alle SWS-Agenten verteilt wird, deren lokale Applikation das Objekt benötigt.

Welche Objekte werden repliziert ?

Welche Objekte benötigt eine Applikation? Ganz einfach — alle diejenigen, auf die sie lesend oder schreibend zugreifen möchte. Die Daten im SWS muß man sich prinzipiell als ein Netzwerk von SWS-Objekten vorstellen, die durch die Pointer-Elementarobjekte des Basissystems miteinander verbunden sind. Bearbeitet eine Applikation schon im SWS vorhandene Objekte, so müssen diese Objekte als lokale Replikate vom SWS-Agent bereitgehalten werden. Die Objekte die von einer Applikation bearbeitet werden, „hängen“ im Normalfall an einem oder an einigen wenigen Objekten. Bei einem CAD-Programm könnte dies zum Beispiel ein Objekt sein, welches die verschiedenen Zeichenebenen einer CAD-Zeichnung verwaltet. Die Zeichenebenen sind weitere Objekte, auf die im Ebenenverwaltungsobjekt Pointer existieren. Die Zeichenebenen haben dann weitere Pointer auf den Inhalt der Zeichenebenen usw. Es stellt sich nun die Frage, ob bei Bearbeitung einer CAD-Zeichnung all diese über Pointer erreichbaren Objekte sofort lokal repliziert werden müssen. Bei sehr grossen Hypertext-Systemen stellt sich diese Frage gar nicht mehr, da dies aus Speichergründen schlicht unmöglich ist.

In dieser Arbeit werden lokale Kopien eines Objektes erst dann beim SWS-Agenten angelegt, wenn die Objekte benötigt werden. Der Anwender kann Objekte beim SWS-Agenten explizit anfordern. Objekte können aber auch implizit angefordert werden, z.B. durch Zugriff auf ein Pointer-Elementarobjekt, das auf ein lokal noch nicht existierendes SWS-Objekt zeigt. Der Pointer fordert dieses Objekt dann automatisch beim SWS-Agenten an (siehe auch Kapitel 6).

Vorerst nicht vorgesehen ist das Entfernen von der Applikation im Moment nicht mehr benötigter Objekte. Dies kann aber z.B. bei Hypertextsystemen notwendig werden, da Speicher nicht unbegrenzt zur Verfügung steht. Man hat hier ein ähnliches Problem wie bei einem Cache: welches Objekt wirft man weg, wenn der Speicher eng wird — meistens das, was als nächstes benötigt wird. Eine nähere Betrachtung dieser Problematik bleibt weiteren Arbeiten vorbehalten.

2.2.5 Replikation von Datenänderungen

Im letzten Abschnitt wird, mehr nebenbei, die Notwendigkeit erwähnt, daß bei Durchführung einer Änderung diese auf allen Objektreplikaten durchgeführt werden muß. In diesem Abschnitt wird darauf näher eingegangen.

Methodenverteilung und Datenverteilung

Objektänderungen können auf verschiedene Weisen an die Replikate des geänderten Objekts verteilt werden. Eine einfache, aber sehr zeitaufwendige Lösung wäre, die Verteilung komplett in die Elementarobjekte der SWS-Objekte zu verlegen. In diesem Falle würde jede Änderung sofort auf alle anderen Kopien dieses Elementarobjektes (also überall dorthin, wo das zugehörige SWS-Objekt lokal im SWS-Agent gehalten wird) verteilt. Bei Methoden mit sehr vielen Schreibzugriffen auf ein und dasselbe Elementarobjekt wird dabei extrem viel Zeit nur für den Update der anderen Kopien des geänderten Objektes verbraucht. Es ist daher unumgänglich, eine bessere Lösung zu finden.

Um Änderungen an Objekten zu verteilen, gibt es grundsätzlich zwei verschiedene Ansätze: entweder verteilt man die geänderten Daten (Objekte) oder man verteilt die Methode, die diese Daten geändert hat. In Bezug auf das in 2.1.2 aufgeführte Beispiel der Verschiebung eines Würfels bedeutet dies, daß entweder die acht Eckkoordinaten des Würfels oder die Methode „Verschiebe“ an alle anderen Kopien des Objektes Würfel verteilt werden. Angesichts dieses Beispiels liegt der Schluß nahe, die Replikation der Änderungen ausschließlich über Methodenverteilung zu machen.

Leider eignen sich nicht alle Methoden für die Methodenverteilung. Geeignete Methoden sind all diejenigen, die mit wenig Aufwand relativ viel Daten ändern bzw. wo bei Datenverteilung sehr viel verteilt werden müßte. Wird zum Beispiel bei einem String (Character Array) ein Buchstabe angehängt, so ist hier eine Verteilung der Methode sehr effizient, eine Verteilung des (eventuell sehr langen) Strings viel zu aufwendig.

Weniger geeignet sind Methoden, bei denen mit sehr viel Rechenaufwand relativ wenig am Objekt geändert wird. Diese würden bei Methodenverteilung alle Rechner belasten, auf denen das zu ändernde Objekt gehalten wird (im Extremfall mehrere Ausführungen der gleichen Methode auf einem Rechner, da jeder Teilnehmer an der Sitzung einen eigenen SWS-Agenten hat).

Die ideale Lösung ist eine Kombination beider Ansätze, bei der automatisch entweder Methoden- oder Datenverteilung verwendet wird — je nachdem, welches von beiden effizienter ist.

Ein ebenfalls schwierig zu entscheidender Punkt ist der Zeitpunkt, wann die Verteilung stattfinden soll (analog zu 2.1.2). Ist eine Methode erst einmal verteilt worden, so müssen Methoden, die aus ihr heraus aufgerufen werden, natürlich nicht mehr verteilt werden. Die Verteilung kann zum frühestmöglichen Zeitpunkt geschehen — dann, wenn eine Methode eines SWS-Objektes (von einem View-Objekt) aufgerufen wird. Statt dessen können aber auch erst die Methoden, die von dieser „äußersten“ Methode aufgerufen werden, verteilt werden (bzw. Methoden, die in der Aufrufhierarchie noch weiter unten stehen). Dies macht auf den ersten Blick nicht sehr viel Sinn — dann muß ja statt einmal eventuell viele Male verteilt werden. Es ist aber i.a. so, daß eine Methode weitere Methoden aufruft. Diese können sich jetzt wiederum mehr für Methoden- oder mehr für Datenverteilung eignen. Hier ist die Entscheidung für eine der beiden Verteilungsstrategien zum Aufrufzeitpunkt der äußeren Methode nicht sehr sinnvoll. Besser ist, die untergeordnet aufgerufenen Methoden zu verteilen, jede so, wie es jeweils am sinnvollsten ist

Ideal wäre auch hier die automatische Entscheidung, wann verteilt werden soll.

Der in dieser Arbeit zu implementierende Ansatz wird durch [Sem93] und die Aufgabenstellung in 1.2 definiert. Datenänderungen werden entweder durch Methodenverteilung oder durch Datenverteilung repliziert. Welche der beiden Verteilungen und zu welchem Zeitpunkt diese durchgeführt werden, wird durch eine Replikationscharakteristik entschieden, die durch den Applikationsentwickler angegeben wird.

Auswirkung auf die Meldeschnittstelle

Änderungen an SWS-Objekten müssen der Applikation über eine Meldeschnittstelle bekannt gemacht werden, damit auf diese Veränderung reagiert werden kann (i.a. neue Darstellung des Objektes). Bei der hier verwendeten Art der Verteilung dieser Änderungen kann man sich zwei Möglichkeiten der Durchführung dieser Meldungen vorstellen:

- Die „oberste“ SWS-Methode[2](d.h. diese SWS-Methode wurde weder direkt noch indirekt von einer SWS-Methode aufgerufen) verteilt nach ihrer Ausführung die Änderungsmeldung an alle Applikationen, die das Objekt verwenden
- Die verteilten Datenänderungen und Methodenaufrufe generieren jeweils lokal eine Änderungsmeldung.

Wie schon in 2.1.2 beschrieben, erhält man dadurch zwei sehr unterschiedliche Qualitäten der Meldungen. Die Verteilung der Meldungen durch die „oberste“ SWS-Methode garantiert einen maximalen Informationsfluß. Bei der Generierung der Meldungen durch die

[2]SWS-Methode = eine auf ein SWS-Objekt aufgerufene Methode

verteilten Methodenaufrufe können diese Meldungen, je nachdem wie „spät“ diese Methode verteilt wird, wesentlich weniger inhaltliche Aussagekraft besitzen. Bei Datenverteilung ist die Meldung nur die Information, daß ein Objekt geändert wurde. Der große Vorteil der Generierung der Meldungen durch die verteilten Daten/Methoden ist eine wesentliche Einsparung an Kommunikation zwischen den Anwendungen. Während beim ersten Ansatz zusätzliche Kommunikation notwendig ist, ist dies hier nicht erforderlich. Da die Kommunikation sehr teuer[3] ist, wird in dem hier zu realisierenden SWS, trotz der besseren Verwertbarkeit der Meldungen der ersten Möglichkeit, die Generierung der Meldungen durch die verteilten Methoden/Daten gemacht.

2.2.6 Die Replikationscharakteristik

Die optimale Entscheidung, welche der beiden Replikationsstrategien zu welchem Zeitpunkt angewandt werden soll, erfordert eine sehr genaue Kenntnis des Programms und der darin implementierten Algorithmen. Eine vollautomatische statische Entscheidung nur anhand des Quellcodes ist, falls überhaupt möglich, extrem schwierig. Bei einer dynamischen Entscheidung müssen die zu verteilenden Methoden erst ausgeführt werden und dabei untersucht werden, welche der beiden Replikationsstrategien die günstigere ist (Abwägung Rechenaufwand der Methode gegen Aufwand der Datenverteilung). Dieser Ansatz scheint deshalb auch nicht machbar.

Der Anwendungsentwickler hat das notwendige Wissen, um diese Entscheidungen zu treffen. Die Syntax der Definitionssprache für die SWS-Objekte wird deshalb um die sogenannte Replikationscharakteristik(RC) erweitert.

Verteilbare Methoden

Die RC setzt sich bei verteilbaren Methoden (i.a. datenändernde Methoden) aus der *Replikationsinformation(RI)* und der *Replikationsgranularität(RG)* zusammen. Die RI beschreibt die Art der Verteilung (Datenverteilung oder Methodenverteilung) durch die Schlüsselwörter *Data* und *Method*, die RG den „Zeitpunkt“ der Verteilung durch die Schlüsselworte *early*(früh, jetzt) und *late*(spät(er)) Als Kurzschreibweise für eine bestimmte Replikationscharakteristik wird die Form RC(RI,RG) verwendet

Die RC kann im Programm an verschiedenen Stellen angegeben werden:

- Bei der class-Definition für alle Methoden aller Instanzen der Klasse
- Bei der Methodendefinition für diese Methode in allen Instanzen der Klasse
- Bei einem Methodenaufruf für genau diesen Methodenaufruf

[3]teuer in Hinsicht auf den Zeitbedarf

Die Priorität nimmt von oben nach unten zu. Die Angabe der RC ist optional, wird sie nicht angegeben, werden default-Werte angenommen.

Aus diesen Angaben wird für jeden Methodenaufruf eine gewünschte Replikationscharakteristik bestimmt. Die tatsächlich verwendete RC wird zur Laufzeit aus der Aufrufhistorie und der gewünschten RC berechnet (siehe Kapitel 7.2).

Nicht verteilbare Methoden

Ganz außer acht gelassen wurden bis jetzt SWS-Methoden, welche nur Daten eines SWS-Objektes lesen. Diese Methoden müssen nicht verteilt werden. Hier ist keine Unterscheidung in RI und RG nötig. Die RC für diese Methode besteht bei der Methodendefinition nur aus dem Schlüsselwort *NotDistributed*. Eine Angabe einer RC bei einem Aufruf dieser Methode ist nicht möglich.

Die Angabe dieser RC ist auch für datenändernde Methoden gestattet und entspricht dann logisch einer beim Aufruf der Methode angegebenen RC (default/late). Der Vorteil ist eine schnellere Abarbeitung der Methode, da der für die Verteilung notwendige Overhead entfällt.

2.3 Die Architektur des Shared Workspace

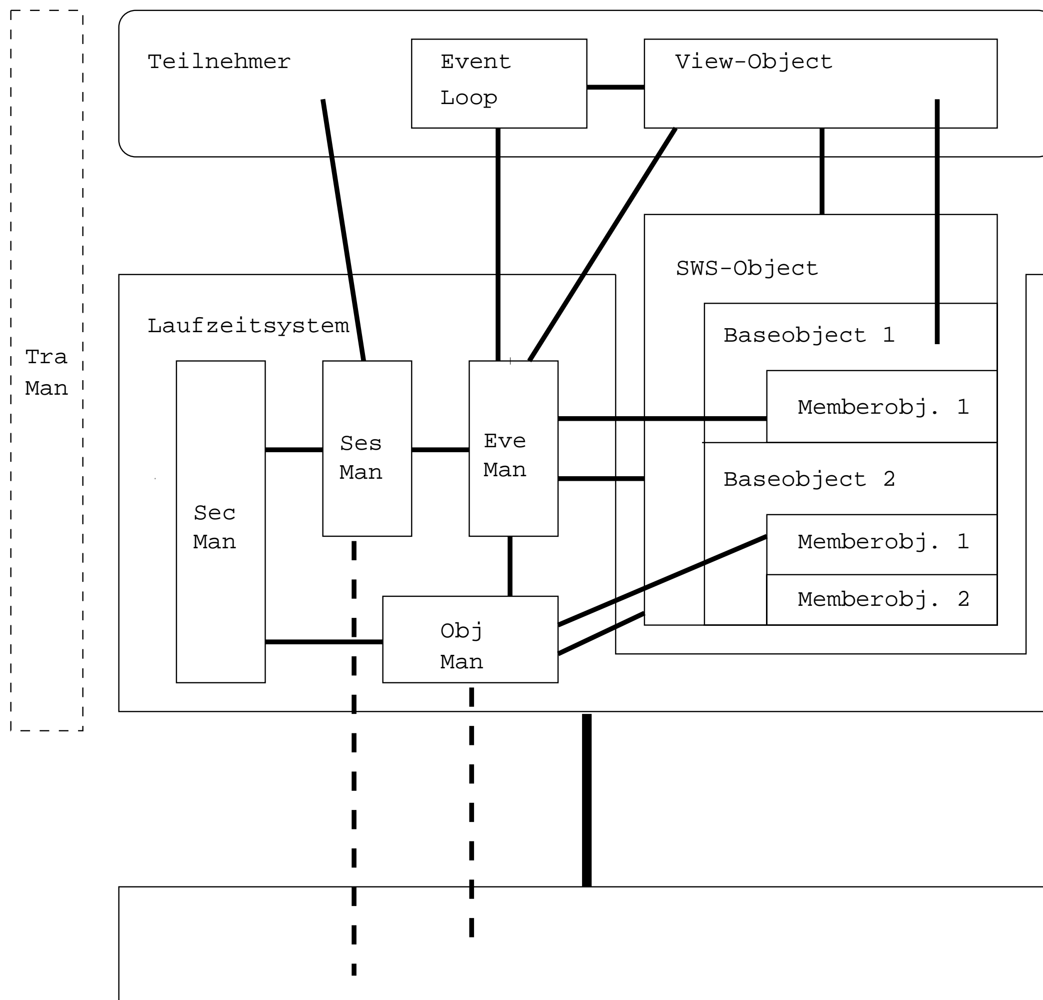
Abbildung 2.3 zeigt die in [Sem93] beschriebene mögliche Software-Architektur. Der SWS-Agent besteht danach aus vier Managern (Object Manager, Event Manager, Session Manager und Security Manager), der angedeutete Transaktionsmanager gehört nicht direkt zum SWS-Agent. Der SWS-Agent setzt auf DCE auf. Die Kommunikation mit anderen SWS-Agenten findet über RPC's (im folgenden oft als „Nachrichten" bezeichnet) statt.

2.3.1 Der Object Manager

Der Object Manager(OM) ist die zentrale Einheit des SWS. Er ist für die Verwaltung der Objekte zuständig. Bei ihm fordert die Anwendung schon im SWS vorhandene Objekte an, neue Objekte werden von ihm in den SWS eingefügt. Ein großer Teil der Kommunikation und Synchronsiation im System wird über den OM abgehandelt (dazu gehört in diesem Falle auch die Lock-Verwaltung). Er kommuniziert dazu mit den OM's der anderen SWS-Agenten

Die zwischen den OM's ausgetauschten Nachrichten kann man grob in zwei Sorten einteilen:

- Nachrichten, die sofort behandelt werden müssen.
- Nachrichten, die nicht sofort behandelt werden müssen.



Sec Man - Security Manager
Ses Man - Session Manager
Eve Man - Event Manager
Obj Man - Object Manager
Tra Man - Transaction Manager

Abbildung 2.3: Architektur des SWS

Zu den sofort zu behandelnden Nachrichten gehören all jene, welche mit der Neuverteilung von Objekten von und zu anderen SWS-Agenten zusammenhängen und einige der Nachrichten des Lockmanagements. Alle anderen gehören zu denen, die nicht sofort bearbeitet werden müssen.

Diese Unterteilung dient der Steigerung der Gesamtleistung im System. Sie ermöglicht die asynchrone Abarbeitung von verteilten Methoden. Ohne die Unterscheidung ergäbe sich folgendes Problem: Knoten[4] I ändert Objekt 1. Durch die RC wird entschieden, daß der

[4]Als Knoten wird eine am SWS teilnehmende Instanz einer Applikation bezeichnet.

erste Teil der Änderung als Methodenverteilung, der zweite Teil als Datenverteilung an die Objektreplikate versandt wird. Knoten II besitzt ein Replikat von Objekt 1, ist also Ziel der Methodenverteilung, mit der Knoten I die Änderung an Objekt 1 verteilt. Wartet Knoten I, bis die verteilte Methode auf Knoten II durchgeführt wurde, kann für den Anwender auf Knoten I eine merkliche Verzögerung auftreten (Knoten II ist eventuell gerade sehr beschäftigt bzw. gerade gar nicht am Rechnen (Multitasking!!) und kann deshalb die verteilte Methode nicht sofort ausführen). Wartet Knoten I nicht, kann es vorkommen, daß die Datenverteilung auf Knoten II ankommt, noch bevor die verteilte Methode fertig ist. Diese Datenverteilung muß nun auf jeden Fall warten, da sie sonst mit der Methodenverteilung in Konflikt gerät. Die Lösung ist die Einführung von Warteschlangen im Objektmanager, in die die nicht sofort auszuführenden Nachrichten eingereiht werden. Theoretisch würde pro OM eine solche Warteschlange ausreichen. Es werden aber bei Methodenverteilungen synchronisierende Nachrichten benötigt (siehe Kapitel 5 und 6), die die Einführung einer Warteschlange pro am SWS teilnehmendem Knoten in jedem OM als sinnvoll erscheinen lassen. Diese Warteschlangen werden als *absenderspezifische Warteschlangen* bezeichnet. Ihr Inhalt wird streng sequentiell abgearbeitet, die synchronisierenden Nachrichten bei Methodenverteilungen werden von der laufenden Methode entnommen. Die Synchronisation zwischen den unterschiedlichen Warteschlangen ist durch den Lockmechanismus automatisch gegeben. Um die Asynchronität noch weiter zu steigern, können die nicht sofort zu behandelnden Nachrichten auch beim Sender in eine Warteschlange gestellt werden und parallel zur laufenden Anwendung verteilt werden.

Wie schon erwähnt, wird die Kommunikation zwischen den OM's mit RPC's durchgeführt. Die oben erwähnten Nachrichten werden durch Remote Procedures realisiert. Die sofort zu behandelnden „Nachrichten“ können durch die Remote Procedure abgehandelt werden. Die nicht sofort zu behandelnden „Nachrichten“ sind etwas komplizierter, ein RPC läßt sich nun mal nicht in eine Warteschlange stellen. Hier trägt die Remote-Procedure den eigentlichen Auftrag des RPC in die entsprechende Warteschlange ein. Der Aufrufer des RPC's wartet in beiden Fällen, bis der RPC beendet ist.

Die Aufgabengebiete des Objektmanagers werden in den Kapiteln 4, 5 und 6 beschrieben.

2.3.2 Der Event Manager

Der Event-Manager stellt die Meldeschnittstelle zwischen SWS und Anwendung (View-Objekten) dar. Über ihn laufen unter anderem die von den Methoden- und Datenverteilungen generierten Änderungsmeldungen.

2.3.3 Der Session Manager

Der Session Manager wird für das Session Management zuständig sein. Er wird in weiteren Arbeiten entwickelt werden.

2.3.4 Der Security Manager

Der Security Manager ist für die Sicherheit im SWS verantwortlich. Dazu gehören Aufgaben wie Autorisierung der Teilnehmer und Authentifizierung. Ebenfalls möglich wäre die Realisierung einer Art Transaktionsmanagement. Eine Realisierung bleibt weiteren Arbeiten vorbehalten.

2.3.5 Der Transaction Manager

Der Transaktionsmanager ist für die Synchronisation der Datenänderungen im SWS verantwortlich. Eine Realisierung bleibt weiteren Arbeiten vorbehalten.

2.4 Der Programmgenerator und das Basissystem

Die Aspekte der Behandlung der Verteilung und der Lock-Verwaltung werden, für den Applikationsprogrammierer so transparent wie möglich, durch den Programmgenerator und das Basissystem (Elementarobjekte und die diversen Manager) abgehandelt. Der Generator transformiert dazu die Klassenspezifikationen (Header und Implementierung) der SWS-Klassen.

Die Schnittstelle zwischen Programmgenerator und Basissystem kann sich auf sehr unterschiedlichen Niveaus befinden. Man kann das Basissystem sehr einfach gestalten, der durch den Programmgenerator zu erzeugende Code wird dann sehr komplex. Je intelligenter dagegen das Basissystem ist, desto einfacher wird der vom Generator zu erzeugende Code.

Bei der Entwicklung der Funktionalität des SWS und des Basissystems in den folgenden Kapiteln wird ein möglichst intelligentes Basissystem angestrebt. Das zur Verwaltung der Objekte notwendige Wissen soll für den Generator nicht relevant sein. Es ist zum Beispiel die Aufgabe des Generators festzustellen, wann Datenänderungen verteilt werden sollen und auf welche Art diese durchgeführt werden soll. Festzustellen, wohin diese verteilt werden müssen, ist Aufgabe des Basissystems.

Dieser Ansatz hat den Vorteil, daß sehr viele Aufgaben zentral im Basissystem abgehandelt werden können. Die Modifikationen des Generators an den Methoden der SWS-Objekte wird dadurch auf ein Minimum reduziert, die Lesbarkeit des generierten Codes erheblich gesteigert.

2.5 Vergleich mit anderen Ansätzen

In diesem Abschnitt soll der in diesem Kapitel vorgestellte SWS anderen Ansätzen zur Unterstützung der Entwicklung von CSCW-Applikationen gegenübergestellt werden.

Art der Unterstützung

Die verschiedenen Ansätze unterstützen die Entwicklung von CSCW-Applikationen auf sehr unterschiedliche Art und Weise. Auf der einen Seite existieren Toolkits, die nur einen Teilaspekt der Entwicklung abdecken. Ein Beispiel dafür ist GroupKit [RG92]. GroupKit ist ein Toolkit zum Erstellen von verteilten visuellen Umgebungen, wo die Aktionen eines Benutzers sofort bei den anderen Benutzern sichtbar werden. Eine andere Art Toolkit versucht, die CSCW-Anwendung aus einzelnen „Bausteinen“ zu modellieren [MLF92]. Eine Entwicklung der Anwendung im herkömmlichen Sinne ist dazu nicht notwendig. Der Ansatz soll dem Anwender gestatten, seine Applikationen selbst zusammenzustellen.

Der generellste Ansatz ist die Erweiterung einer normalen Programmiersprache bzw. der Entwurf einer neuen Sprache, um die Erstellung der CSCW-Anwendung zu vereinfachen. Rendezvous [PHR90] bietet eine Erweiterung von Common Lisp. Eine Applikation besteht aus einem zentralen Prozess, mit dem die Anwender über virtuelle Terminals kommunizieren. Eine Erweiterung der Sprache Smalltalk [ND92] realisiert einen Shared Workspace, indem sie das Smalltalk-Image ändert, so daß verteilbare Objekte definiert werden können und auf diese transparent zugegriffen werden kann.

Die wenigsten der hier erwähnten Erweiterungen (abgesehen von der Smalltalk-Erweiterung) sind für die Entwicklung von CSCW-Programmen geeignet, die auch mit anderen CSCW-Anwendungen kooperieren können. Der in diesem Kapitel vorgestellte SWS ist gerade zur Entwicklung solcher Anwendungen entworfen worden.

Verteilung der Daten der CSCW-Anwendung

Die Ansätze für die Datenhaltung der CSCW-Applikationen sind sehr unterschiedlich. Eine Möglichkeit ist die zentrale Haltung der Daten an einer Stelle, wie dies von manchen Systemen realisiert wird (z.B. [PHR90]). Die meisten Systeme (z.B. [ND92] [GSW92], ...) jedoch halten die Daten verteilt. Bei den SWS-Ansätzen existiert jedes Objekt i.a. einmal und wird transparent angesprochen, d.h. es ist für die Applikation kein Unterschied, ob das Objekt lokal vorhanden ist oder nicht — abgesehen von der Zugriffszeit. Diese ist bei nicht-lokalen Zugriffen sehr hoch. Der hier entwickelte SWS umgeht dieses Problem teilweise, indem er die Objekte überall lokal hält, aber bei jeder Objektänderung alle Objektreplikate aktualisiert.

Kapitel 3

Das Objektmodell

Was ist ein Objekt — und welchen Anforderungen muß ein SWS-Objekt genügen? Diesen Fragen soll in diesem Kapitel auf den Grund gegangen werden.

3.1 Objektorientierte Programmierung

3.1.1 Was ist ein Objekt

Ein Objekt ist eine Einheit, welche einen Status und eine Menge von Operationen hat, um diesen Status zu lesen und zu ändern. Eine solche Einheit bildet i.a. einen Gegenstand oder Vorgang der realen Welt ab. Der Status eines Objektes beschreibt in einer beliebigen, im Normalfall nach außen nicht sichtbaren Form den modellierten Gegenstand bzw. Vorgang und setzt sich aus anderen Objekten (in diesem Zusammenhang auch Attribute genannt) zusammen. Der Zugriff auf diese Attribute (lesend oder ändernd) erfolgt über die Operationen, welche auch Methoden genannt werden. Die Methoden bilden also die Schnittstelle des Objektes zur Umwelt, die interne Realisierung ist nach außen hin nicht sichtbar (Datenkapsel). Die Kommunikation zwischen den Objekten findet konzeptionell über Nachrichten statt.

Unser alter Bekannter, der Würfel, muß auch hier wieder als Beispiel dienen. Abbildung 3.1 zeigt eine mögliche, sicher nicht vollständige Schnittstelle eines Objektes Namens *Würfel*. Diese Definition sagt nichts über die interne Realisierung des Objektes aus. Der Würfel und seine Lage könnte beispielsweise durch seine acht Ecken oder durch eine Ecke und drei Vektoren beschrieben sein. Die Schnittstelle bietet die Möglichkeit, den durch das Objekt beschriebenen Würfel völlig neu zu definieren, oder nur Teilaspekte des Würfels (Lage im Raum, Oberfläche, ...) zu ändern. Dem Würfel wird eine solche Änderung über eine Nachricht mitgeteilt. Eine gebräuchliche Schreibweise (unter anderem in C++) ist die Form `Objekt.Nachricht(Parameter)`. Unser Würfel kann also unter Verwendung von `Würfel.verschiebe(Vektor)` verschoben werden. Der Parameter `Vektor` ist ebenfalls ein Objekt, das einen Vektor beschreibt.

```
Objekt-Definition Würfel
   Methode neuer_Würfel(Position, Kantenlänge, Lage_im_Raum, Oberfläche)
   Methode drehe(Drehung)
   Methode verschiebe(Vektor)
   Methode neue_Oberfläche(Oberfläche)
Ende (Objekt-Definition Würfel)
```

Abbildung 3.1: Schnittstelle des Objekts *Würfel* in Pseudo-Notation

Bei C++ wurde aus Performanzgründen leider auf die komplette Kapselung des Zustandes verzichtet. Die Zustandsdaten (*data members* genannt) und die Methoden (*function members* genannt) werden gleich behandelt. Es existieren Schlüsselworte (sogenannte *protections*), die einen Member (data oder function) entweder als *private* oder als *public* deklarieren. Ein *private member* kann nur von Methoden des Objekts verwendet werden, auf *public members* kann von der Allgemeinheit zugegriffen werden.

3.1.2 Klassen

Die Definition einzelner Objekte ist recht mühsam, da i.a. mehrere gleiche Objekte benötigt werden. Ein Grafikprogramm benötigt für jeden Würfel ein eigenes Objekt *Würfel* — theoretisch beliebig viele. Es kann nicht für jeden Würfel eine eigene Definition existieren. Die Objektorientierte Programmierung hat deshalb Klassen eingeführt. Grob gesagt ist eine Klasse eine Schablone für ein Objekt. In der Klassendefinition wird angegeben, wie die Schnittstelle eines Objektes, das dieser Klasse angehört, aussieht. Die Definition einer Klasse könnte ähnlich wie die in Abbildung 3.1 gezeigte Objektdefinition aussehen. Ein Objekt ist dann die Instanz einer Klasse. Auf diese Weise können viele Würfel-Objekte hergestellt werden. Für jedes Objekt muß bei der Definition angegeben werden, von welcher Klasse es ist.

Für C++ gilt bei den Klassen das schon bei den Objekten gesagte. Die *private*-Eigenschaft bezieht sich aber jetzt auf die gesammte Klasse, d.h. jedes Objekt einer Klasse kann auf die privaten Daten und Methoden der anderen Objekte, die derselben Klasse angehören, zugreifen. C++ ist (trotz der objektorientierten Ansätze) eine typisierte Sprache. Eine Klasse ist in C++ ein Typ, die Objekte sind folglich Variablen dieses Typs. C++ bietet eine besondere Art von Methoden an, die sogenannten *Konstruktoren.* Dies sind Methoden, welche bei der „Konstruktion“, das heißt bei der Deklaration, des Objektes ausgeführt werden und zur Initialisierung dienen (die Methode `neuer_Würfel` ist ein Beispiel für solch eine Methode). Es gibt noch weitere spezielle Methoden der Objekte in C++, für nähere Information wird eine generelle Einführung in C++ empfohlen ([Lip91] u. andere).

3.1.3 Vererbung

Die Vererbung ist ein weiteres strukturierendes Mittel der OOP[1]. Sie stellt gewisse Verwandschafts-Beziehungen, die zwischen unterschiedlichen Klassen herrschen, her.

Bei Vererbung erbt eine Klasse von einer anderen Klasse. Die erbende Klasse enthält, zusätzlich zu den in ihr definierten Methoden (und Attributen), i.a. alle Attribute und Methoden der geerbten Klasse.

Für diese Form der Vererbung, der *Einfachvererbung*, kann man wiederum den Würfel als Beispiel heranziehen. Ein Würfel ist eigentlich ein Spezialfall des Quaders. Die Attribute, die den Würfel beschreiben sind denen des Quaders sehr ähnlich. Der einzige Unterschied ist, daß die Kanten des Würfels alle gleich lang sind. Auch die Methoden des Drehens und des Verschiebens sind im Endeffekt dieselben. Unterschiedlich ist lediglich jene Methode, mit der das Objekt initialisiert wird. Die Klasse Würfel kann also die Klasse Quader erben, muß aber die Initialisierungfunktion mit einer Funktion überladen, die einen Würfel und nicht den allgemeinen Quader initialisiert. Bei diesem Beispiel bleiben die Attribute eigentlich die gleichen, nur die Interpretation durch die Methoden ändert sich.

Der sicherlich häufigere Fall ist der, daß in der spezialisierten Klasse auch noch einige Attribute hinzukommen. [Lip91] führt als ein Beispiel einen Zoo an. Eine der dort möglichen Klassen ist die Klasse *Zootier*. Ein *Zootier* hat einen bestimmten Käfig im Zoo, bestimmte Fütterungszeiten, eventuell ein Geburtsdatum, die Verbreitung usw. Die Klasse *Zoo-Bär* ist eine Spezialisierung des Zootiers, sie erbt diese Klasse. Zusätzlich werden in dieser Klasse noch spezielle Eigenschaften der Bären definiert. Eine Klasse, die von der Klasse *Zoo-Bär* erbt, könnte zum Beispiel die Klasse *Zoo-Panda* sein.

Die kompliziertere Form der Vererbung ist die *Mehrfachvererbung*. Bei der Mehrfachvererbung kann eine Klasse von mehreren Klassen erben. Ein Bär im Zoo ist ein Zootier, er ist aber auch ein Bär. *Zoo-Bär* könnte also sowohl von *Zootier* als auch von der allgemeinen Klasse *Bär* erben. Dabei werden sich sicherlich einige der Attribute überschneiden. Die Verbreitung ist zum Beispiel sowohl bei einem Zootier als auch beim allgemeinen Bären vorhanden. Ähnliche Konflikte kann es auch bei Methoden geben.

Anhand des Zoo-Beispiels wird auch der Vorteil der Mehrfachvererbung deutlich (s. Abbildung 3.2). Angenommen, die Klassen *Zootier* und *Bär* existieren bereits. Die Definition der Klasse *Zoo-Bär* ist dann sehr einfach und schnell durch das Erben dieser beiden Klassen möglich. Bei Einfachvererbung müßten die Bär-spezifischen Eigenschaften alle in der Klasse *Zoo-Bär* definiert werden, die eventuelle Überschneidung von Attributen und Methoden würde aber entfallen.

In C++ ist die Mehrfachvererbung erlaubt. Dies bringt einerseits wesentliche Vorteile im Klassenentwurf mit sich, macht die Klassenhierarchie aber etwas undurchsichtiger. In Kombination mit einigen Eigenheiten von C++, auf die hier nicht näher eingegangen werden soll (virtuelle Funktionen, überladene Funktionen, virtuelle Vererbung, Freund-

[1] Objekt-Orientierte Programmierung

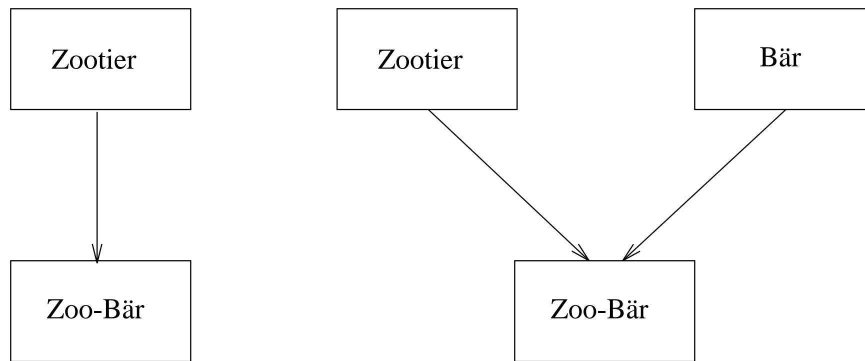


Abbildung 3.2: Mögliche Vererbungshierarchien der Klasse *Zoo-Bär*

Beziehungen), ist die Mehrfachvererbung ein nicht ganz ungefährliches Instrument, welches nur mit Vorsicht zur Anwendung kommen sollte.

Die *protections* in C++ werden für die Vererbung noch um das Schlüsselwort *protected* erweitert. Damit gekennzeichnete Members sind nur für die Klasse selbst und alle von ihr (direkt oder indirekt) erbenden Klassen zugreifbar. Die *protections* können aber nicht nur für Members eingesetzt werden, auch bei der Vererbung selbst können sie eingesetzt werden. So kann eine Klasse eine andere Klasse entweder *public, protected* oder *private* erben. Public geerbte Klassen entsprechen der obigen Beschreibung der Vererbung. Private Basisklassen[2] sind nur von der direkt erbenden Klasse sichtbar. Erbt eine Klasse eine andere Klasse, welche eine private Basisklasse besitzt, so kann sie nicht auf die Daten und Methoden dieser privaten Basisklasse zugreifen.

3.2 Objekte im SWS

Die SWS-Objekte sollen in einer der Sprache C++ angelehnten Definitionssprache definiert werden und vom Programmgenerator in reines C++ umgewandelt werden. Die folgenden Abschnitte beschreiben detailliert die Realisierung der SWS-Objekte in C++. Die Terminologie lehnt sich deshalb an C++ an.

3.2.1 Die Daten des SWS-Objektes

In Abschnitt 2.2.1 wurde der Aufbau eines vom Applikationsentwickler definierten SWS-Objektes schon einmal erwähnt. Eine SWS-Klasse setzt sich wie jede normale Klasse in C++ aus *data members, function members* und den geerbten Basisklassen zusammen. Bei den SWS-Objekten wird hier die Einschränkung gemacht, daß sie als *data members* nur andere SWS-Objekte und die Elementarobjekte des Basissystems besitzen dürfen. Auch die Vererbung ist auf SWS-Klassen eingeschränkt, d.h. eine SWS-Klasse darf nur andere

[2]in C++ ist eine Basisklasse eine Klasse, welche (direkt oder indirekt) geerbt wurde.

SWS-Klassen als Basisklassen besitzen und eine SWS-Klasse darf nur von anderen SWS-Klassen geerbt werden.

Der Grund für die Einschränkung der *data members* auf die Elementarobjekte des Basissystems und andere SWS-Objekte ist folgender: Der Ansatz des SWS sieht vor, die Datenänderungen an Replikate der SWS-Objekte zu „beliebigem“ Zeitpunkt zu übertragen (siehe Abschnitt über Replikationscharakteristik). Aus diesem Grund muß jedes Objekt im SWS in der Lage sein, die Änderungen, die an dem Objekt vorgenommen werden, zu verteilen. Dies kann ein beliebiges Objekt nicht. Angenommen, ein SWS-Objekt besitzt einen beliebigen Member, der kein SWS-Objekt ist. Auf dieses SWS-Objekt wird eine Methode aufgerufen. Die Replikationscharakteristik ergibt für diesen Aufruf, daß erst später eine Methodenverteilung stattfinden soll. Ändert diese Methode nun direkt diesen einen Member, so muß diese Änderung übertragen werden. Da aber von der Seite dieses Members dies nicht vorgesehen ist, gestaltet sich die Verteilung dieser Änderung recht schwierig, vor allem, da die Struktur dieses Members sehr komplex sein kann. Eine Bit-weise Kopie kommt aus Portabilitätsgründen nicht in Frage (der SWS soll auch in inhomogenen verteilten Systemen nutzbar sein), eine allgemeine Kodierung des Objektinhaltes zur Übertragung ist sehr komplex. Aus diesem Grund werden als Grundbausteine der SWS-Objekte die Elementarobjekte des Basissystems verwendet, die die Verteilung ihrer Datenänderungen unterstützen.

Unter anderem aus diesem Grund wird auch die Vererbung bei SWS-Klassen eingeschränkt. Dürfte jede beliebige Klasse von einer SWS-Klasse geerbt werden, bestünde dasselbe Problem wie bei den *data members*.

3.2.2 Das Generische Objekt

Würde die Vererbungshierarchie einer SWS-Klasse ungeändert in C++ umgesetzt, hätte man ein großes Problem. C++ ist eine typisierte Sprache. Es ist also in C++ nur schwer machbar, eine Sammlung beliebiger Objekte zu verwalten, so wie dies im SWS geschehen soll. Eine Variable eines Typs kann nur genau diesen Typ enthalten. Etwas flexibler sind da schon Pointer. Ein Pointer ist im Normalfall definiert als ein Pointer auf einen bestimmten Typ. Er kann nun auf Objekte dieses Typs bzw. auf Objekte, die diesen Typ[3] als Basisklasse besitzen, zeigen. Erbt zum Beispiel die Klasse B von der Klasse A, kann ein Zeiger vom Typ *Zeiger auf A* sowohl auf Objekte vom Typ A als auch auf Objekte vom Typ B zeigen. Der flexibelste Zeiger ist ein *void-Pointer*, der auf Variablen (Objekte) beliebiger Typen zeigen kann.

Um „beliebige“ Objekte zu verwalten gibt es drei Möglichkeiten:

- Verwaltung der Objekte als anonyme Objekte. Auf jedes Objekt existiert im Objektmanager ein *void-Pointer*.
- Verwaltung der Objekte getrennt nach Klassen.

[3] In C++ ist eine Klasse ein Typ !

- Erweiterung der Objekthierarchie um eine gemeinsame Basisklasse.

Die Verwaltung der Objekte als eine Menge von *void-Pointern* scheidet von vornherein aus. Der Objektmanager muß die von anderen Knoten verteilten Datenänderungen (egal ob Daten- oder Methodenverteilung) auf irgendeine Art an das betroffene Objekt weitergeben. Dies kann nur geschehen, wenn der Typ des Objektes bekannt ist.

Die Verwaltung getrennt nach Klassen scheint auf den ersten Blick eine gangbare Lösung. Der Objektmanager muß alle SWS-Klassen kennen, die in einer Applikation verwaltet werden. Dies kann durch den Programmgenerator sichergestellt werden, er muß dazu jedoch die gesamte Applikation parsen. Die Verwendung von Bibliotheken macht aber Probleme, da intern eventuell SWS-Objekte verwendet werden, die nach außen nicht sichtbar sind. Ausserdem wird der Objektmanager sehr umständlich, da jedes Objekt anders behandelt wird. Ganz unmöglich wird diese Lösung allerdings erst dadurch, daß durch den Objektmanager eventuell SWS-Objekte verwaltet werden müssen, deren Klassen zur Übersetzungszeit der Applikation noch nicht existierten (siehe Kapitel 6).

Die Lösung ist also die Erweiterung der SWS-Klassen durch den Programmgenerator um eine gemeinsame Basisklasse. Diese Klasse ist das *Generische Objekt (GO).* Es enthält gewisse virtuelle Funktionen, die zur Kommunikation zwischen OM und dem Objekt notwendig sind (Funktionen zur Daten-/Methodenverteilung, „Cast“-Funktionen,...) und Members, welche allen SWS-Objekten gemeinsam sind. Auf diese Weise kann der OM eine Menge von GO's verwalten, mit denen er über eine definierte Schnittstelle kommuniziert. Die Handhabung der sehr unterschiedlichen Objekte wird dadurch vereinheitlicht. Die Elementarobjekte sind vom Prinzip her auch SWS-Objekte. Sie müssen genau wie diese vom Objektmanager verwaltet werden. Es bietet sich deshalb an, daß auch sie das Generische Objekt als Basisklasse haben. Einen näheren Einblick in den Aufbau der Basisobjekte bietet Abschnitt 4.3.2

Der Haken dieser Lösung liegt in der mehrfachen Vererbung. Ein Objekt einer SWS-Klasse, die mehrere SWS-Klassen erbt, enthält mehrere generische Objekte. Welche werden vom OM verwaltet — alle ? Die Lösung liegt in der wohl seltsamsten Eigenheit von C++, der *virtuellen Vererbung.* Bei der virtuellen Vererbung enthält eine Klasse eine (in verschiedenen Basisklassen) mehrmals virtuell geerbte Basisklasse nur einmal. Ein Beispiel zeigt Abbildung 3.3. In dem Beispiel sind insgesamt drei SWS-Klassen abgebildet. Die Klassen A und B, welche keine weiteren Klassen erben und die Klasse C, welche die Klassen A und B erbt. Alle drei Klassen erben das Generische Objekt. Auf der linken Seite der Abbildung ist die Vererbungshierarchie der Klasse C bei normaler Vererbung dargestellt. Das gestrichelt dargestellte Generische Objekt ist dabei nicht unbedingt notwendig. Die rechte Seite der Abbildung zeigt dieselbe Hierarchie bei virtueller Vererbung der Klasse GO, wobei die gestrichelt eingezeichnete Vererbung dem gestrichelt eingezeichneten GO der linken Seite entspricht. Bei der virtuellen Vererbung des Generischen Objektes ist das (virtuelle) Erben des GO's durch die Klasse C wegen der Konstruktoren der Klasse GO unbedingt notwendig. Die durch den Programmgenerator erzeugte Klassendefinitionen der Ableitungshierarchie der Klasse C zeigt Abbildung 3.4.

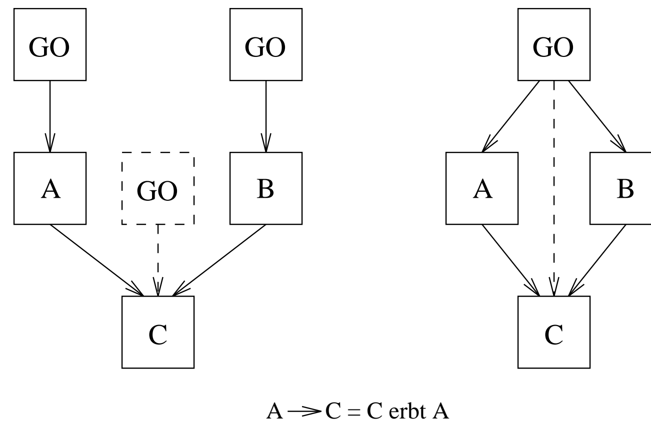


Abbildung 3.3: Normale Vererbung des GO und virtuelle Vererbung des GO

3.2.3 Die Object-Identifier und der Aufbau des Objektes

Die Objekt-Id

C++ kann ein Objekt im Normalfall anhand seiner Adresse eindeutig identifizieren. Im SWS ist dies nicht möglich. Die Objekte sind hier nur auf jedem Knoten durch ihre Adresse lokal eindeutig identifizierbar, eine SWS-weite Identifizierung ist damit nicht möglich. Dies ist im SWS aber unumgänglich. Objekte müssen auf andere Knoten verteilt werden, Datenänderungen müssen an die anderen Replikate eines geänderten Objekts übertragen werden usw.

Es muß deshalb eine eindeutige Objektidentifizierung eingeführt werden, die sogenannte OID(Objekt-Id). Eine OID ist eindeutig, d.h. sie darf nur ein einzigesmal an ein Objekt vergeben werden. Dies ist in einem verteilten System i.a. nicht trivial. Im Falle des SWS ist die Lösung einfach, da der SWS auf DCE aufsetzt. DCE bietet unter anderem die Generierung sogenannter UUID's (Universal Unique ID) an. Diese haben eine Länge von 128 Bit (16 Byte) und werden durch das DCE weltweit eindeutig erzeugt. Objekte sind durch eine solche UUID also eindeutig identifizierbar.

Das generische Objekt könnte also eine UUID enthalten, und wäre dadurch eindeutig iden-

```
class A : virtual public SGO_GenericObject { ...};

class B : virtual public SGO_GenericObject { ...};

class C : virtual public SGO_GenericObject, protA A, protB B { ...};
```

`protA` und `protB` sind die in der SWS-Klassendefinition spezifizierten *protections* von A und B

Abbildung 3.4: Generatorerzeugte Klassendefinitionen der Klasse C und ihrer Basisklassen

tifizierbar. Dies wäre aber ein sehr aufwendiges Verfahren. Ein Objekt hat im Normalfall Member-Objekte, welche i.a. Member-Objekte haben, usw. Jedes dieser Memberobjekte hat ein GO als Basisklasse. Hätte jedes dieser GO's eine UUID, dann würde ein Objekt eventuell sehr viele UUID's enthalten. Da eine UUID sehr groß ist, würden die Objekte daher sehr groß. Man stelle sich ein Objekt vor, welches nur einen Member hat, der nur einen Integer enthält. Dieses Objekt würde bereits zwei UUID's beinhalten (36 Byte für einen Integer (4Byte)).

Das Top-Level-Objekt

Bei genauerer Betrachtung stellt man fest, daß Objekte, welche Members eines anderen Objektes sind, nie allein vorkommen, sie gehören zu dem Objekt. Dies ist eine Eigenschaft des einzelnen Objekts, nicht der Klasse. Man kann also bei einzelnen Objekten noch unterscheiden, ob sie zu einem anderen Objekt gehören oder nicht. Die Objekte, welche zu keinem weiteren Objekt mehr gehören werden hier **Top-Level-Objekte** genannt. Die Eigenschaft Top-Level-Objekt ist objektspezifisch, hängt also nicht von der Klasse ab. Ein Objekt kann sowohl Top-Level-Objekt sein (z.B. per *new* erzeugt), es kann aber auch Member in einem anderen Objekt sein. Man kann diese Eigenschaft also nicht durch die Vererbung erschlagen, obwohl dies auf den ersten Blick als sehr verlockend erscheint (ein Top-Level-Objekt ist eigentlich eine Spezialisierung eines Objektes).

Die Top-Level-Eigenschaft bietet die Möglichkeit, die Object-Identifier in ihrem Platzbedarf etwas zu zu reduzieren. Prinzipiell reicht es aus, das Top-Level-Objekt eindeutig zu identifizieren. Alle Objekte, die in dem Top-Level-Objekt enthalten sind, können durch eine eindeutige Numerierung innerhalb des Top-Level-Objektes identifiziert werden. Bild 3.5 zeigt die zwei möglichen Strategien zur eindeutigen Numerierung. Die erste ist eine Art hierarchische Numerierung. Jeder Member eines Objekts erhält eine im Objekt eindeutige Numerierung. Die eindeutige Numerierung innerhalb des Top-Level-Objektes ist dann die Zusammensetzung der Nummern der umschließenden Objekte. Die zweite Strategie ist die

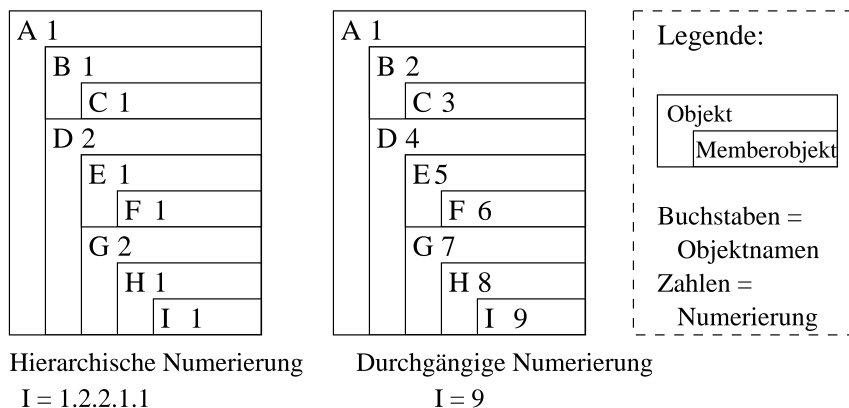


Abbildung 3.5: Mögliche Strategien zur eindeutigen Numerierung der Member innerhalb eines Objektes

eindeutige Durchnumerierung aller Objekte innerhalb des Top-Level-Objektes. Die erste Strategie hat einige Nachteile. Um die Objekt-Id eines bestimmten Objektes zu bestimmen, muß jedes Objekt einen Pointer auf sein umgebendes Objekt besitzen. Diesem Pointer müßte nachgegangen werden, um die eindeutige Nummer des Objektes zu erhalten. Das Ergebnis dieses Ansatzes sind unterschiedlich lange Nummern, die insgesamt auch nur sehr spärlich besetzt sind (ein Objekt hat selten mehrere Dutzend Members). Der zweite Ansatz ist wesentlich handlicher. Um die eindeutige Nummer innerhalb des Top-Level-Objektes zu bestimmen, ist nur ein Zugriff notwendig. Für die Bestimmung der Objekt-Id ist aber dennoch ein Zeiger auf das Top-Level-Objekt notwendig. Die eindeutige Nummer innerhalb eines Top-Level-Objektes wird **Memberobjekt-Id (MOID)** genannt. Die Realisierung als Integer (unsigned) sollte ausreichen, da ein SWS-Objekt keine Arrays im herkömmlichen Sinne enthält (Die vom Basissystem angebotenen Arrays sind ein einzelne Objekte, welche keine SWS-Objekte enthalten.). Die Objekt-Id ergibt sich als Zusammensetzung der MOID und der eindeutigen Nummer des zugehörigen Top-Level-Objektes.

Die Top-Level-Information

Die UUID, mit der die Top-Level-Objekte eindeutig identifiziert werden, heißt die **Top-Level-Objekt-Id (TLOID)**. Diese muß irgendwo gespeichert werden. Wird sie im GO gespeichert, dann ist die Unterscheidung zwischen Top-Level-Objekten und anderen Objekten umsonst. Es wird daher eine weitere Klasse, die **Top-Level-Information (TLI)** eingeführt. Jede Instanz dieser Klasse gehört genau zu einem Top-Level-Objekt. Sie speichert die TLOID und weitere Informationen, die für das Objekt als ganzes notwendig sind (wo existiert das Objekt noch, wer hat welchen Lock auf dieses Objekt, ...). Man könnte die TLI auch als die *Verwaltungsinformation* des Top-Level-Objekts bezeichnen. Logisch gehört die TLI damit eher zum Objektmanager, sie muß aber auch für alle im Top-Level-Objekt enthaltenen Objekte zugreifbar sein (z.B. zur Bestimmung ihrer Objekt-Id anhand der MOID und der TLOID). Es wird deshalb in den Generischen Objekten der Member eines TLO's ein Pointer auf die zugehörige TLI gehalten. Ein Pointer auf umgebende Objekte wird nicht benötigt, da die Unterscheidung innerhalb eines Objektes, welches jetzt das Top-Level-Objekt ist und welches die in diesem TLO enthaltenen Objekte sind, bei dieser Struktur nicht nötig ist.

Einer der wichtigsten Member der Top-Level-Information wurde bis jetzt noch gar nicht erwähnt. Die Top-Level-Information enthält eine Tabelle mit Pointern auf die GO's der Objekte, die im zugehörigen TLO enthalten sind (einschließlich einem Pointer auf das oberste Objekt), die sogenannte **Membertabelle**. Die Indizes dieser Tabelle sind die MOID's der Objekte, auf die die Pointer zeigen (speziell ist der erste Eintrag mit Index Null der Pointer auf das Top-Level-Objekt). Den schematischen Aufbau eines Objektes samt TLI zeigt Abbildung 3.6.

Dieser Aufbau erlaubt eine relativ einfache Verwaltung der Objekte im Objektmanager. Der OM muß nur die TLI's in einer Verwaltungsstruktur speichern. Die Objekte findet er einfach durch suchen der TLI (die TLOID ist in der OID enthalten) und einen

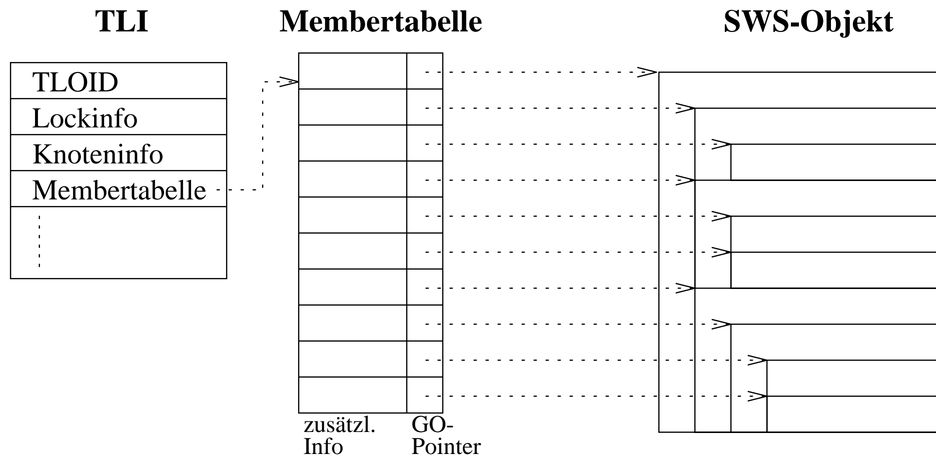


Abbildung 3.6: Aufbau eines SWS-Objektes

Zugriff in die Membertabelle. Dieser Aufbau verringert den Suchaufwand innerhalb des Objektmanagers. Wenn alle Objekte (bzw. deren GO's) verwaltet werden müßten, wäre die entsprechende Struktur viel größer. Ein weiterer Vorteil ist die direkte Zugänglichkeit jedes einzelnen Objektes von der TLI aus. In der Membertabelle können zusätzliche Informationen angespeichert werden (z.B. ob der Member ein Elementarobjekt oder ein Pointer-Elementarobjekt ist, ...), die weitere Operationen wesentlich vereinfachen. Einen möglichen Aufbau der Verwaltung der Objekte im Objektmanager zeigt Abbildung 3.7.

Der Aufbau des Objektes wird automatisch bei der Initialisierung eines Objektes durch die Konstruktoren erstellt (einschliesslich Erstellung der TLI). Das Eintragen der Verwaltungsinformationen (TLOID, ...) und der Eintrag in die Verwaltungsstruktur des OM wird durch diesen erst anschließend erledigt..

Man kann jetzt einwenden, daß durch die vielen Pointer nichts eingespart wurde gegenüber

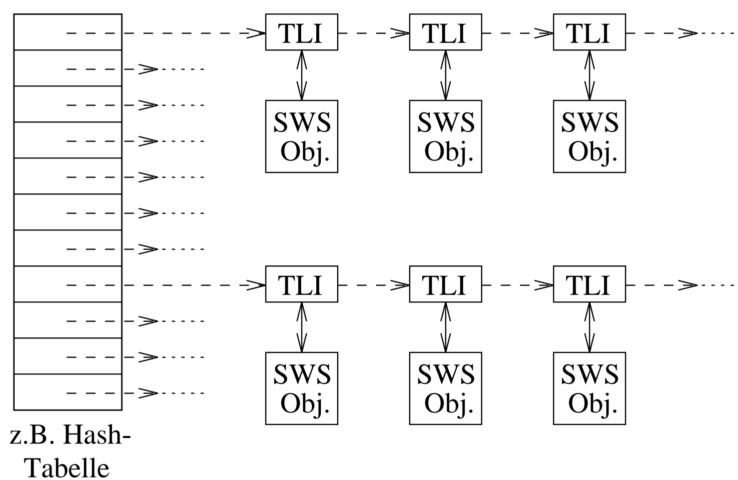


Abbildung 3.7: Verwaltung der Objekte im Objektmanager

der Version mit der UUID im Generischen Objekt. Man muß sich aber vor Augen halten, daß sich die Pointer der Membertabelle in der Verwaltungsstruktur des OM wiedergefunden hätten. Die Pointer von den GO's auf die TLI vereinfachen einige Zugriffe wesentlich und wären auf jeden Fall notwendig geworden.

Base Class Identifier

Ein bislang nicht beachtetes Problem entsteht durch die Flexibilität von C++ bezüglich der Pointer. Ein Pointer kann auf eine beliebige Basisklasse eines Objekts zeigen. Diese Möglichkeit sollen auch die Pointer-Elementarobjekte des Basissystems bieten. Abbildung 3.8 zeigt links anhand der Klasse D zwei Probleme. Ein Pointer auf ein Objekt C kann in C++ auch auf ein Objekt vom Typ D zeigen, da C eine Basisklasse von D ist. Ein Pointer ist in C++ im Normalfall als eine Zahl realisiert, welche eine Speicheradresse angibt. Vergleicht man einen C-Pointer (also ein Pointer, der auf ein Objekt der Klasse C zeigt) und einen D-Pointer, die beide auf dasselbe Objekt der Klasse D zeigen, so sind die beiden Zahlenwerte unterschiedlich. Bei Zuweisungen innerhalb einer Applikation ist dies kein Problem, der Compiler macht diese Umrechnungen automatisch. Problematisch wird dies aber im SWS. Wird ein Pointer-Elementarobjekt geändert (z.B. durch Zuweisung), so muß diese Änderung verteilt werden. Eine Übertragung der Adresse als solche ist unsinnig, die Übertragung der Objekt-Id des Objektes, welches dem Pointer zugewiesen wurde, reicht aber nicht aus.

Wird einem Pointer auf ein Objekt der Klasse C ein Objekt der Klasse D zugewiesen und diese Zuweisung an die anderen Replikate des Pointers verteilt, so können die Replikate anhand der Objekt-Id nur einen Pointer auf das zum Objekt gehörige Generische Objekt ermitteln und daraus (mittels einer virtuellen Methode) einen Pointer auf ein Objekt vom Typ D. Leider weiß der Pointer auf ein Objekt der Klasse C nicht, was für einen Pointer er mittels dieser virtuellen Methode bekommt, da bei der Definition von C nicht alle

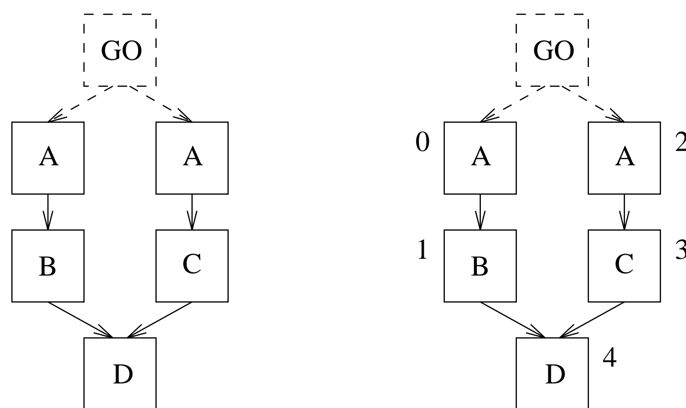


Abbildung 3.8: Mehrdeutigkeit bei Basisklassen und deren mögliche Auflösung

potentiellen erbenden Klassen bekannt sein können.

Es muß also eine Möglichkeit geschaffen werden, daß auch die Basisklassen innerhalb eines Objektes eindeutig identifizierbar sind. Ein Ansatz wäre, das Generische Objekt nicht virtuell, sondern „normal“ zu vererben, wie es in Abbildung 3.3 auf der linken Seite dargestellt ist. Dies würde für die reine Pointer-Zuweisung ausreichen. Es wäre dadurch ein Pointer auf das Generische Objekt ermittelbar, der in einen Pointer auf das gewünschte Objekt (in diesem Fall ein Objekt der Klasse C) umgewandelt werden könnte. Abgesehen davon, daß dies sehr viel Platz benötigt, bringt dies auch andere Probleme. Um Daten und Methoden verteilen zu können, muß das Generische Objekt eine Schnittstelle bieten, über die der OM die Verteilung an das entsprechende Objekt weitergeben kann. Diese Schnittstelle muß auf virtuellen Funktionen basieren, da das Generische Objekt nicht alle Objekte kennen kann, von denen es geerbt wird. Beim Aufruf der virtuellen Funktion würde aber prinzipiell die entsprechende Funktion der am weitesten abgeleiteten Klasse[4] aufgerufen (in dem konkreten Beispiel die Implementation der virtuellen Funktion in der Klasse D). Diese müßte nun die entsprechende Basisklasse wissen, auf die die Methode zu verteilen ist — was bei entsprechenden Protections in der Vererbungshierarchie gar nicht möglich ist.

Eine Möglichkeit wäre die Identifizierung der Basisklasse durch eine eindeutige Klassen-Id, die sowieso benötigt wird, um das korrekte Erstellen von Objekten zu gewährleisten. Abbildung 3.8 zeigt aber, daß dies durch die Tatsache, daß Basisklassen in der Vererbungshierarchie mehrfach vorkommen können, nicht als Lösung in Frage kommt. Die Lösung ist eine eindeutige Durchnumerierung der Basisklassen eines Objektes, wie sie in Abbildung 3.8 auf der rechten Seite dargestellt wird. Diese Numerierung wird dynamisch bei der Initialisierung des Objektes durch die Aufrufreihenfolge der Konstruktoren definiert. Da diese in C++ genormt ist (siehe [ES92]), ist die Numerierung bei Objekten derselben Klasse überall gleich. Da die Numerierung die Basisklassen eines Objektes identifiziert, werden die Nummern **Base Class Identifier** genannt. Sie sind Member von jeder Klasse. Eine Basisklasse kann nun durch eine virtuelle Funktion gefunden werden, indem der „Vererbungsbaum“ nach dieser Base Class Id durchsucht wird.

Hiermit hat man nun die Möglichkeit, einen Elementar-Pointer auf ein SWS-Objekt eindeutig global darzustellen. Er setzt sich zusammen aus der Objekt-Id und der Base Class Id. Zur Kodierung eines Elementarpointers müssen dann nur diese beiden ID's kodiert werden.

Die „Navigationsstruktur“

Die Suche (egal zu welchem Zweck) nach einem Basisobjekt innerhalb eines Objektes geschieht mittels virtueller Methoden, die auf dem zugehöriges Generischen Objekt aufgerufen werden. Diese Methoden werden in jeder Klasse durch den Programmgenerator

[4] d.h. nicht die Funktion einer der Basisklassen des Objekts

	0	1	2	3	4
0	0	0	0	0	0
1	1	0	0	0	0
2	0	0	0	0	0
3	0	0	1	0	0
4	1	1	2	2	0

Tabelle 3.1: Navigationsstruktur der Klasse D

definiert. Beim Aufruf wird (da virtuell) die entsprechende Methode der Klasse des Objekts aufgerufen. Es gibt jetzt zwei Möglichkeiten. Die erste Möglichkeit ist eine Tiefensuche innerhalb der Vererbungshierarchie. Dazu wird einfach nacheinander in den direkten Basisklassen gesucht, welche wiederum in ihren direkten Basisklassen nachschauen. Um die Basisklasse Nummer 2 des Objektes D in Abbildung 3.8 zu finden, würde zuerst die Basisklasse B mit dem Suchen beauftragt, welche dann A beauftragen würde. A hat keine weiteren Basisklassen, würde also melden, dass die Basisklasse nicht gefunden wurde, genauso B. D würde darufhin C mit der Suche beauftragen, welche dann A beauftragen würde. Damit wäre die gesuchte Basisklasse gefunden und die eigentlich durchzuführende Methode könnte durchgeführt werden.

Diese Art der Suche ist nicht zielgerichtet. Eine zielgerichtete Suche wäre aus Effizienzgründen aber wünschenswert. Im gerade ausgeführten Beispiel wäre der direkte Weg der Suche sofort über die Basisklasse C. Das Problem ist nur, daß die Numerierung der Basisklassen dynamisch ist. Ist D selbst Basisklasse eines anderen Objektes, dann haben seine Basisklassen andere Nummern als im Beispiel. Das Wissen, bei welcher Nummer über welche der Basisklassen gesucht werden muß, ist also abhängig von der Vererbungshierarchie. Man kann dieses Wissen aber in Form einer Tabelle für jedes Objekt generieren. In dieser Tabelle ist für jede Basisklasse des Objektes ein Eintrag, in dem angegeben wird, welche Basisklassen von dieser Basisklase erreichbar sind, und wie diese erreichbar sind. Tabelle 3.1 zeigt dies anhand des Objektes D. Die Zeilenindizes sind die Base Class Id's der Basisklassen, von denen aus gesucht wird, die Spaltenindizes die Nummer der gesuchten Basisklassen. Die 1 in (4/0) sagt zum Beispiel aus, daß von Basisklase 4 (in dem Fall von D) aus die Basisklasse 0 (in dem Fall A) über die erste direkte Basisklasse (hier: B) erreicht werden kann. Die direkten Basisklassen einer Klasse werden dazu von 1 bis n (n = Anzahl der direkten Basisklassen) in der Reihenfolge ihrer Definition im Klassenkopf durchnumeriert. B ist also die erste, C die zweite direkte Basisklasse von D. Die obige Suche der Basisklasse 2 wäre hierdurch schneller von statten gegangen. Die Funktion in D hätte in der Tabelle in Zeile 4 (eigene Base Class Id), Spalte 2 nachgeschaut. Die dortige 2 verrät ihr, daß die gesuchte Basisklasse über ihre direkte Basisklasse C zu finden ist (zu beachten ist, daß die Numerierung der direkten Basisklassen bei 1 beginnt, die Base Class Id's bei 0).

Diese Tabelle kann für jede Klasse bei der Bearbeitung der Klassendefinition durch den Programmgenerator erzeugt werden und in einer statischen Struktur abgelegt werden. Diese Struktur wird (in Ermangelung eines besseren Namens) **Navigationsstruktur** genannt (mittels ihr wird durch die Basisklassen navigiert). Bei der Initialisierung eines SWS-Objektes werden jedesmal die Pointer auf die entsprechenden Navigationsstrukturen in den Generischen Objekten der Members abgelegt.

Ein Beispiel für eine Methode, welche den *this-Pointer* einer Basisklasse mit gegebener Base Class Id zurückgibt ist in Anhang A aufgeführt

3.2.4 Resultierende Einschränkungen

Aus dem Objektaufbau resultieren mehrere Einschränkungen. Die erste Einschränkung ergibt sich aus der Tatsache, daß der Konstruktor zum Aufbau dieser Objektstruktur benötigt wird und somit dem Anwender nicht zur Verfügung steht. Dieser muß eine separate Initialisierungsmethode schreiben, falls dies notwendig sein sollte. Analog entfällt der Destruktor für den Anwender. Diese Einschränkung ist sowieso sinnvoll, da zum Zeitpunkt der Konstruktion das Objekt noch kein vollwertiges SWS-Objekt ist. Die Ausführung normaler SWS-Methoden auf dieses Objekt würde weitere Komplikationen mit sich bringen. Eine bisher nicht angesprochene Einschränkung ist das Verbot der Benutzung des Copy-Konstruktors, eine nähere Erläuterung dazu folgt in Abschnitt 4.3.2.

Zwei weitere Einschränkungen betreffen die möglichen Formen der Vererbung. Prinzipiell sind die gleichen Vererbungshierarchien möglich wie in C++. Es gibt hierbei zwei Ausnahmen:

- Der Cast von einer Klasse auf ihre direkten Basisklassen muß eindeutig sein.
- Es muß beachtet werden, daß die Konstruktoren der SWS-Objekte keine Default-Konstruktoren sind. Benutzt der Anwender virtuelle Vererbung, so muß er darauf achten, daß die virtuellen Basisklassen „nach oben“[5] sichtbar bleiben, da deren Konstruktoren in den *initializer lists* der Konstruktoren aller Klassen, die diese virtuelle Basisklasse direkt oder indirekt erben, aufgerufen werden müssen.

Die beiden Fälle werden in Abbildung 3.9 veranschaulicht. In der Abbildung fehlt das GO, das von allen Klassen *virtual public* geerbt wird. Die linke Abbildung veranschaulicht Ausnahme eins. Hier ist das rechte A von C aus nicht erreichbar, was aber für gewisse Algorithmen (zum Beispiel Suche eines Basisklasse) erforderlich ist. Die rechte Abbildung veranschaulicht die zweite Ausnahme.

[5] für die erbenden Klassen

3.3 SWS-Objekte in der Applikation

In C++ gibt es zwei Möglichkeiten, Objekte zu erstellen. Sie werden entweder mittels Angabe der Klasse und einem Bezeichner deklariert (classname o; erstellt ein Objekt o der Klasse classname) oder sie werden durch den *new*-Operator erstellt. Für SWS-Objekte ist die erste Möglichkeit nur innerhalb von SWS-Klassendefinitionen sinnvoll, da bei dieser Art Deklaration auf jeden Fall ein neues Objekt erstellt wird, das bei Verlassen des Bezugsrahmens wieder verschwindet. Dies ist nicht sonderlich sinnvoll. SWS-Objekte werden deshalb ausschließlich durch die vom Programmgenerator für jede SWS-Klasse erzeugte statische Memberfunktion **newObject()** erstellt. Diese Funktion gibt einen Zeiger auf das durch sie neu erstellte Objekt zurück.

Diese Lösung spiegelt auch das Verhältnis einer Applikation zu den durch sie bearbeiteten SWS-Objekten wieder. Die Applikation besitzt nicht die SWS-Objekte sondern nur einen Verweis auf ein Objekt im SWS.

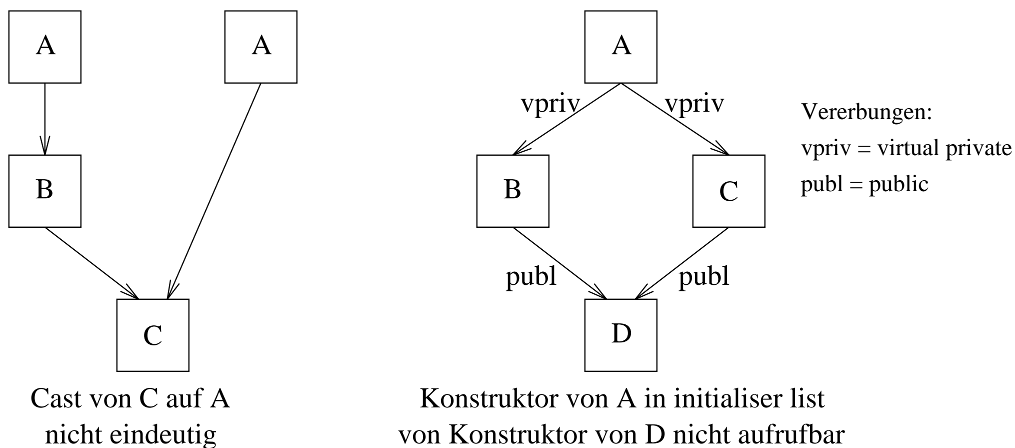


Abbildung 3.9: Zwei Fälle von Vererbung, die nicht möglich sind

Kapitel 4

Replikation

Dieses Kapitel beschreibt die Verteilung der Änderungen an einem Objekt an die im SWS existierenden Replikate des Objektes. Der erste Teil geht auf allgemeine Aspekte der Realisierung dieser Verteilung ein. Der zweite Teil beschäftigt sich mit der Datenverteilung während der dritte Teil die Verteilung der Datenänderungen mittels Verteilung der ändernden Methode beschreibt.

4.1 Allgemeine Aspekte der Verteilung der Änderungen von Objekten

Änderungen von SWS-Objekten müssen an alle Replikate des geänderten Objektes bekanntgegeben werden. Im SWS soll dies wahlweise mittels Verteilung der geänderten Daten oder mittels Verteilung der ändernden Methode geschehen. Welche dieser beiden Verteilungsmethoden zu welchem Zeitpunkt angewendet wird, wird durch die Auswertung einer durch den Programmierer angegebenen Replikationscharakteristik RC (siehe Abschnitt 2.2.6) entschieden. Es wird dabei bei jedem Aufruf einer *verteilbaren* Methode ausgewertet, ob jetzt zu diesem Zeitpunkt verteilt werden soll, und wenn ja, ob dies durch Methodenverteilung oder Datenverteilung geschehen soll. Die Entscheidung für eine Verteilung betrifft die ganze Methode, bei deren Aufruf die Entscheidung zur Verteilung erfolgte, d.h. sämtliche mit dieser Methode geänderten Daten (auch in weiteren Methodenaufrufen innerhalb der Methode) werden verteilt. Zumindest bei Methodenverteilung ist diese Tatsache einleuchtend.

Diese Entscheidung, und die damit verbundene Abhandlung der Verteilung kann an zwei verschiedenen Orten geschehen:

- Am Ort des Aufrufes der Methode
- Am Anfang bzw. in der aufgerufenen Methode

Die Entscheidung am Ort des Aufrufes hat dabei zwei gravierende Nachteile.

- Der gesamte Source-Code einer Applikation, die am SWS teilnehmen soll, muß untersucht werden und jeder Aufruf einer ändernden SWS-Methode muß um den Code zur Verwaltung der Verteilung erweitert werden. Dies bedingt einerseits die Implementation eines vollständigen Parsers für C++. Andererseits werden die generierten Programme dadurch unnötig lang, da jeder Aufruf einer ändernden SWS-Methode um Code ergänzt werden muß. Bei der Behandlung der Verteilung am Anfang der Methode ist dies nur in jeder ändernden SWS-Methode nötig, die Aufrufe bedürfen keiner weiteren Behandlung.

- Virtuelle Funktionen machen die Behandlung der Verteilung am Ort des Aufrufes beinahe unmöglich. Bei Aufruf von Funktionen auf einen Zeiger oder eine Referenz auf ein Objekt wird bei virtuellen Funktionen abhängig vom referenzierten Objekt entschieden, welche „Version“ der Funktion ausgeführt wird. Bei Methodenverteilung müßte dieser Mechanismus der virtuellen Funktionen nachgebildet werden, damit die richtige Methode verteilt werden könnte. Ein Zugriff auf den entsprechenden Mechanismus des Compilers ist nicht portabel und auch relativ aufwendig.

Bei Behandlung der Verteilung in der Methode hat man diese beiden Probleme nicht mehr. Der Code für die Verteilung ist nur einmal in jeder ändernden SWS-Methode enthalten, das Durchsuchen des gesamten Source-Codes nach den Methodenaufrufen entfällt. Zudem müssen die virtuellen Methoden nicht gesondert behandelt werden, da die Entscheidung für die Methode zum Zeitpunkt der Verteilung schon gefallen ist (und der verteilenden Methode daher auch bekannt ist).

Der Programmgenerator handelt aus den eben angeführten Gründen die Verteilung einer Methode in dieser Methode ab. Die Realisierung wird in den nächsten beiden Abschnitten behandelt.

4.2 Die Datenverteilung

4.2.1 Ermittlung der zu verteilenden Daten

Unter Datenverteilung versteht man die Verteilung der geänderten Daten eines Objektes an dessen Replikate durch die Übertragung der geänderten Daten (im Gegensatz zur Verteilung der Änderung durch Verteilung der ändernden Methode). Man kann sich hier wieder zwei Ansätze vorstellen:

- Jede Objektänderung während der Ausführung der Methode (einschliesslich der Ausführung der von der Methode direkt oder indirekt aufgerufenen Methoden) wird sofort verteilt. Dies könnten die Elementarobjekte selbstständig machen, wenn sie anhand der Replikationscharakteristik feststellen, daß Datenverteilung angesagt ist.

- Die Änderungen werden protokolliert und am Ende der Methode gesammelt verteilt.

Auf den ersten Blick scheint der erste Ansatz die idealere Lösung zu sein. Er ist einfacher zu realisieren und benötigt keine aufwendige Protokollierung der Änderungen. Der entscheidende Nachteil ist der, daß jede Kommunikation sehr „teuer“ ist. In der Zeit, die zur Kommunikation benötigt wird, kann sehr viel protokolliert werden. Wird zudem ein Elementarobjekt mehrere Male geändert, so wird jede Änderung einzeln übertragen, anstatt nur das endgültige Ergebnis der Methode zu verteilen.

Beim zweiten Ansatz können einzelne Elementarobjekte beliebig oft geändert werden, es wird nur das endgültige Ergebnis der Methode verteilt. Welche Informationen werden für diesen Ansatz benötigt? Theoretisch reicht die Menge der geänderten Elementarobjekte aus. Diese könnten dann am Ende der zu verteilenden Methode ihren aktuellen Inhalt an die anderen Replikate verteilen. Dies würde für jedes einzelne Elementarobjekt mehrere Kommunikationen erfordern. Man kann aber i.a. davon ausgehen, daß, falls überhaupt mehrere Elementarobjekte geändert werden, diese zu einem, oder, wenn innerhalb der zu verteilenden Methode auch Methoden auf anderen Top-Level-Objekten aufgerufen werden (z.B. über Pointer), einigen wenigen Top-Level-Objekten gehören. Es bietet sich an, die in einem Top-Level-Objekt geänderten Elementarobjekte bei der Übertragung zusammenzufassen, wodurch wesentlich weniger Kommunikation benötigt wird. Dadurch erweitert sich prinzipiell das notwendige Wissen nicht. Durch die Realisation der Objekte im SWS läßt sich dieses Wissen in einer recht eleganten Form repräsentieren. Man erweitert die *Membertabelle* um ein Flag, das anzeigt, ob ein Objekt geändert wurde. Dieses Flag hat nur bei Objekten eine Bedeutung, die Elementarobjekte sind. Wird ein Elementarobjekt geändert, während Datenverteilung angesagt ist (ergibt sich aus der RC), so veranlaßt es, daß das zu ihm gehörige Flag in der Membertabelle gesetzt wird. Es ist nun nur noch eine Menge jener Top-Level-Objekte (genauer: eine Menge von TLI's) zu erstellen, in denen Elementarobjekte geändert wurden. Dies kann durch die Methode erledigt werden, die die Flags in der Membertabelle bei Elementarobjektänderungen setzt. Diese Menge der geänderten TLO's wird vom Objektmanager selbst verwaltet. Am Schluß sind nur alle in dieser Menge enthaltenen Top-Level-Objekte der Reihe nach abzuarbeiten.

4.2.2 Form der verteilten Daten

An dieser Stelle muß man sich wohl Gedanken darüber machen, wie man die Daten denn nun verteilt. Um die Daten verteilen zu können, benötigt man eine eindeutige, auch in heterogenen Umgebungen benutzbare Kodierung der Daten. Dazu könnten zum Beispiel entsprechende Funktionen des DCE verwendet werden. Die Kodierung ihrer Inhalte ist Aufgabe der Elementarobjekte. Die Planung ist dort jedoch zum momentanen Zeitpunkt noch nicht so weit fortgeschritten. Es wird deshalb davon ausgegangen, daß die Elementarobjekte eine Methode zur Kodierung in einen Puffer besitzen. Diese Methode bekommt als Parameter ein Objekt Puffer, in dem der Inhalt des Elementarobjektes additiv abgelegt wird (d.h. die Kodiermethode hängt die Kodierung des zu kodierenden Elementarobjektes an den bisherigen Inhalt des Buffers an). Abbildung 4.1 zeigt die Kodierung eines Integer-Wertes in einen Buffer, der durch ein Byte-Array (DCE: Octett String) realisiert ist. Zur

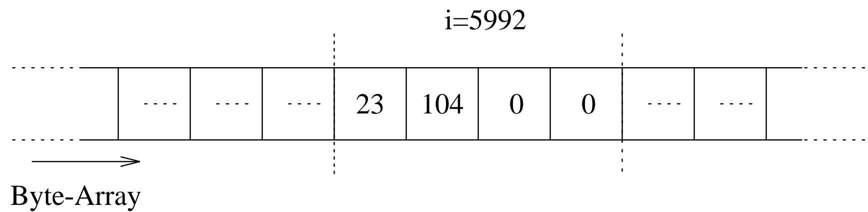


Abbildung 4.1: Kodierung eines Integers (4 Byte) in MSB-Order

Dekodierung existiert eine dazu „inverse“ Methode. Zudem besitzen die Elementarobjekte eine Methode, die den Platz in Bytes als Rückgabewert liefert, den sie zur Kodierung ihres Dateninhaltes benötigen.

Die Kodierung der gesamten Änderungen innerhalb eines TLO's ist dann relativ einfach. Es existiert hierzu eine Methode der TLI, welche einfach das Memberarray nach geänderten Elementarobjekten absucht (anhand der Flags). Es wird nun zuerst die Größe des benötigten Buffers ermittelt, indem die entsprechenden Methoden der Elementarobjekte aufgerufen werden. Die Größe ergibt sich aus der Summe der von den Elementarobjekten gemeldeten Anzahl von Bytes plus der Anzahl Bytes, die zur Kodierung der Memberobjekt-Id's der geänderten Elementarobjekte benötigt wird. Zur Kodierung werden die Daten der geänderten Elementarobjekte abwechselnd mit den Kodierungen ihrer MOID's im Feld abgelegt. Für die Kodierung der MOID's sorgt die Methode der TLI, nicht die der Elementarobjekte. Nach der Kodierung werden die Flags in der Membertabelle zurückgesetzt. Die Dekodierung ist ähnlich einfach. Die entsprechende Methode des TLI dekodiert dazu einfach abwechselnd die MOID's und ruft dann die entsprechende Dekodierfunktion des so ermittelten Elementarobjektes auf.

Die hier beschriebene Kodierung besitzt noch einen Schönheitsfehler. Ein Elementararry ist ein einzelnes Elementarobjekt, das eventuell sehr viele Daten enthält. Bei dem gerade beschriebenen Algorithmus müßten sämtlich Daten dieses Arrays kodiert werden, wenn ein einzelnes Element geändert wurde. Elementararrays „merken“ sich deshalb bei Datenverteilung die Elemente, die geändert wurden. Bei der Kodierung werden dann nur diese Elemente (mit entsprechenden Zusatzinformationen) kodiert.

4.2.3 Verteilung durch den OM

Die Änderungen müssen jetzt nur noch verteilt werden. Dies ist Aufgabe des Objektmanagers. Er weiß, auf welchen Knoten die Änderungen verteilt werden müssen. Da er auch die Menge der geänderten TLO's kennt, ist die Verteilung komplett durch ihn durchführbar — er muß nur noch wissen, zu welchem Zeitpunkt. Dies wird ihm von der zu verteilenden Methode am Ende der Ausführung der Methode durch die Funktion

```
OM::dataDistribution();
```

mitgeteilt. Er muß jetzt die Menge der geänderten Top-Level-Objekte abarbeiten. Jedes Objekt wird kodiert und diese Kodierung an alle anderen Replikate des Objektes ver-

schickt. Diese Nachrichten können bei Verwendung einer Sende-Warteschlange auch in diese gestellt und verzögert abgeschickt werden.

4.2.4 Unlock während Datenverteilung

Ein Objekt kann nur geändert werden, während der ändernde Knoten einen Write-Lock auf dieses Objekt besitzt. Dieser Write-Lock kann nach einer Änderung freigegeben werden, obwohl die ändernde Methode noch läuft. Geschieht dies, während eine Methode datenverteilt wird, müssen die Änderungen dieses Objektes frühzeitig verteilt werden. Dafür existiert im Objektmanager die Methode

```
OM::dataDistribution(TLI *tli);
```

Diese Methode wird bei Bedarf von der Unlock-Methode aufgerufen (siehe Kapitel 5). In dieser Methode werden die Änderungen des zur übergebenen TLI gehörigen TLO's kodiert und an die Replikate des Objektes versandt. Das TLO (eigentlich die TLI) wird dabei aus der Menge der geänderten TLO's entfernt. Auch die Verteilung dieser Nachrichten kann in die Sende-Warteschlange gestellt werden.

4.2.5 Eingehende Datenverteilungen

Bei Eingang einer Datenverteilung wird diese vom Objektmanager erst einmal in die absenderspezifische Warteschlange gestellt (die Warteschlange, die zu dem Knoten gehört, von dem die Datenverteilung kommt). Diese Datenverteilung enthält neben der Kodierung der Daten auch noch eine Kodierung der Top-Level-Objekt-Id des geänderten TLO's. Sobald die Datenverteilung an der Reihe ist, wird anhand der TLOID die TLI des TLO's ermittelt, bei der dann die Methode zur Dekodierung aufgerufen wird. Die Methode bekommt die Kodierung der Daten als Parameter.

4.3 Die Methodenverteilung

Die Methodenverteilung verteilt Änderungen an Objekten dadurch, daß die ändernde Methode bei den anderen Replikaten des Objektes ebenfalls ausgeführt wird. Der Knoten, der diese Methodenveränderung veranlaßt, wird **Initiator(-Knoten)** der Methodenverteilung genannt. Die Knoten, auf die die Methode verteilt wird, werden **Remote-Knoten** der Methodenverteilung genannt. Ein Knoten kann zur gleichen Zeit also sowohl Initiator als auch Remote-Knoten sein, diese Begriffe beziehen sich immer nur auf eine bestimmte Methodenverteilung.

Für die eigentliche Verteilung der Methode gibt es insgesamt drei größere Probleme zu bewältigen:

- Die eindeutige Identifikation der Methode
- Die Parameter der Methode

- Der Rückgabewert der Methode

Diese Probleme werden im folgenden besprochen (und gelöst).

4.3.1 Eindeutige Identifikation der Methode

Um eine Methode verteilen zu können, muß diese eindeutig identifizierbar sein. Abbildung 4.2 zeigt ein Beispiel. Ein Objekt der Klasse *D* „enthält“ zwei Objekte der Klasse *A*. Die Klasse A besitzt insgesamt drei Methoden *m1* bis *m3*. *m1* und *m3* sind verteilbare Methoden, *m2* ist nicht verteilbar. Auf einem Objekt der Klasse *D* können auf beide Teilobjekte *A* Methoden aufgerufen werden (z.B. `B::m1();` oder `C::m1();`). Eine Kodierung der Methoden durch eine Kombination aus Klassenname und Methodenname ist daher nicht möglich. Die Kodierung wird dadurch zusätzlich erschwert, daß sie zur Zeit der Definition der Klasse *A* schon festliegen muß, da sie in die Methoden der Klasse mit eingebaut werden muß. Die Lösung heißt auch hier wieder **Base Class Id**. Diese identifiziert die Basisklasse des Objektes, deren Methode ausgeführt wird, eindeutig. Die Methoden-Id's setzen sich deshalb aus der Base Class Id und einer innerhalb der Klasse eindeutigen Nummer der verteilbaren Methode zusammen. Im Beispiel hätte die Methode *m1* die eindeutige Nummer null und *m3* die Nummer eins. *m2* benötigt keine Nummer, da diese Methode nicht verteilbar ist. Der Aufruf von `B::m1();` auf ein Objekt der Klasse *D* hätte als eindeutige Methoden-Id die Kombination 0/0 ((Base Class Id)/(eindeutige Methodennummer in der Klasse)), der Aufruf von `C::m3();` hätte die Methoden-Id 2/1.

4.3.2 Parameter der Methoden

Mögliche Parameter

Methoden haben im allgemeinen Parameter. Die Methoden der SWS-Objekte sollen (und können) da keine Ausnahme sein. Leider reicht bei einer Methodenverteilung die Verteilung der Methode als solches nicht aus, die Parameter müssen auch verteilt werden. Dies wirft die Frage auf, wie diese Parameter kodiert werden können. Die einfachste Lösung wäre,

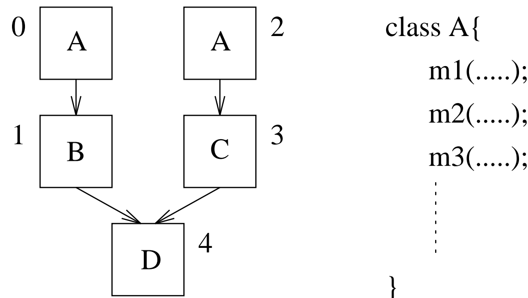


Abbildung 4.2: Problematische Methodenidentifikation

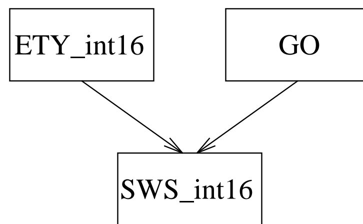


Abbildung 4.3: Vererbungshierarchie der Elementarobjekte

nur SWS-Objekte als Parameter zuzulassen – wie sollten jedoch bei diesem Ansatz Daten in den SWS gelangen?

Eine Lösung bietet die Beschränkung der Parameter der SWS-Methoden auf SWS-Objekte und auf die Grundtypen der Elementarobjekte des Basissystems. SWS-Objekte sind einfach durch die schon besprochenen Identifizierungsmöglichkeiten der Objekte (OID, Base Class Id) zu kodieren. Für die Grundtypen der Elementarobjekte muß sowieso eine Kodierung implementiert werden. Für die Elementarobjekte wurde deshalb eine Vererbungshierarchie wie in Abbildung 4.3 entworfen. Sie erben sowohl einen Grundtyp als auch das Generische Objekt. Die Abbildung zeigt ein Elementarobjekt, das einen 16-Bit Integer darstellt. Dabei definiert der Grundtyp (*ETY_int16*) den 16-Bit Integer samt Kodierungsfunktion, in *SWS_int16* werden die elementarobjektspezifischen Methoden implementiert. Die Kodierung dieser Grundtypen bietet daher auch keine Schwierigkeit. Pointer und Referenzen werden übertragen, indem der Wert des Grundtyps übertragen wird. Von der Verwendung der Elementar-Arraytypen als Parameter wird allerdings abgeraten, da dies, je nach Größe des Arrays, bei der Verteilung erheblichen Aufwand mit sich bringt.

SWS-Objekte und der Copy-Konstruktor

Prinzipiell könnte man bei SWS-Objekten alle Arten der Parameterübergabe (Wertparameter, Referenzparameter, Übergabe als Zeiger auf das Objekt) zulassen, die Verteilung wäre eigentlich kein Problem. Problematisch wird hier, wie an anderen Stellen auch, die Erzeugung einer Kopie des SWS-Objektes. Was ist eine Kopie eines SWS-Objektes? Identisch mit dem Original kann sie nicht sein. Was wird bei einer Kopie also erstellt? Es gibt zwei Möglichkeiten:

- Ein weiteres SWS-Objekt mit anderer TLOID
- Ein Objekt, welches kein SWS-Objekt mehr ist

Ersteres wäre technisch machbar, ist aber nicht sehr sinnvoll. Die Lebensspanne eines Werteparameters endet mit dem Ende der Methode (Funktion), an die er übergeben wurde — SWS-Objekte sind aber als langlebige Objekte konzipiert. Die Erzeugung eines Objektes, welches kein SWS-Objekt mehr ist, widerspricht vollkommen der Semantik des Copy-Konstruktors, der i.a. eine vollwertige Kopie des Objektes erstellt. Was würde man

machen, wenn auf einer Kopie, welche vom Typ her noch ein SWS-Objekt ist, eine verteilbare Methode aufgerufen wird und diese verteilt werden müßte? Die Erstellung von Kopien eines SWS-Objektes durch den Copy-Konstruktor ist folglich nicht erwünscht, der Einsatz des Copy-Konstruktors verboten. Da eine Definition nicht verhindert werden kann (wenn der Anwender keinen definiert, wird einer durch den C++-Compiler definiert), wird im Generischen Objekt ein Copy-Konstruktor definiert, der eine Exception „wirft“. Eine Verwendung dieses Konstruktors führt bei SWS-Objekten auf einen Fehler. Diese Einschränkung der Objektsemantik betrifft neben der Parameterübergabe die Rückgabe von Funktionswerten (darunter besonders die benutzerdefinierten Umwandlungen und allgemein Operatoren) und die implementationsabhängige Verwendung von temporären Variablen durch den Compiler. Die ersten beiden Punkte unterliegen dem direkten Einfluß des Programmierers und können umgangen werden. Das Anlegen temporärer Variable kann durch die Vermeidung komplizierter Ausdrücke i.a. auch vermieden werden.

Bezogen auf die Parameterübergabe verhindert das Verbot der Verwendung von Copy-Konstruktoren bei SWS-Objekten die Übergabe der SWS-Objekte als Wertparameter. SWS-Objekte können daher nur als Referenzparameter oder als Zeiger auf ein SWS-Objekt übergeben werden.

Verfügbarkeit der Parameter auf den Remote-Knoten

Parameter, welche Pointer oder Referenzen auf SWS-Objekte darstellen, werden durch Kodierung der OID und der Base Class Id übertragen. Es muß daher sichergestellt werden, daß von diesen Objekten auf den Remote-Knoten ein Replikat existiert. Dies wird von der Methode des OM sichergestellt, die die Methode auf die Remote-Knoten verteilt. Sie erhält dazu als Parameter die Menge der Top-Level-Objekte, zu denen jene Parameter gehören, welche SWS-Objekte sind.

4.3.3 Rückgabewert der SWS-Methoden

Was geschieht mit dem Rückgabewert einer SWS-Methode? Diese Frage stellt sich generell nicht nur für den Rückgabewert, sondern auch allgemein für Referenzparameter und Pointer, die bei SWS-Methoden übergeben werden. Wird eine Methode nur lokal ausgeführt, kann sie nur ein Ergebnis haben. Wird sie verteilt ausgeführt, kann sie z.B. durch Inkonsistenzen im SWS (bzw. durch Fehler in der Programmierung) verschiedene Ergebnisse haben. Durch Vergleich dieser Ergebnisse könnte man Fehler im SWS schneller finden. Dies würde aber zusätzliche Kommunikation benötigen. Von einem Vergleich der Ergebnisse wird daher abgesehen. Da die Rückgabewerte einer Methode nicht mehr übertragen werden müssen, ist der Rückgabetyp prinzipiell beliebig. Man muß aber hier beachten, daß SWS-Objekte nicht zurückgegeben werden können, sondern nur Referenzen und Pointer auf SWS-Objekte (um die Verwendung des Copy-Konstruktors zu vermeiden).

4.3.4 Verteilung der Methode

Aufgaben beim Initiator

Bevor die Methode verteilt werden kann, muß der Aufruf zuerst kodiert werden. Die Kodierung besteht aus der Methoden-Id (Base Class Id (variabel, je nach Objekt), eindeutige Nummer der Methode innerhalb der Basisklasse (fest)) und der Kodierung der Parameter. Diese Kodierung wird nun mitsamt einer Referenz auf das verteilende Objekt und der oben erwähnten Menge der Top-Level-Objekte folgender Methode des OM übergeben:

```
OM::methodDistribution(
     GO &oref,        // Ref. auf Obj., auf das Methode verteilt wird
     TLISet tset,     // Menge eventuell zu verteilender TLO's
     CodeBuffer *buf, // Buffer mit kodierter Methode
)
```

Anhand der Objektreferenz stellt der OM die OID des Objektes fest, auf das die Methode verteilt werden muß. Bevor er jedoch die Methodenkodierung an die Objektreplikate verteilt, muß er zuerst sicherstellen, daß die Parameter der Methode, welche SWS-Parameter sind, auf den Remote-Knoten vorhanden sind. Die Menge der Remote-Knoten kann er der TLI des Objektes entnehmen, auf das die Methode verteilt wird. Die zu verteilenden Objekte sind in tset (Menge von TLI's) enthalten. Anhand dieser TLI's kann festgestellt werden, welche der Parameter noch nicht auf allen Remote-Knoten lokal vorhanden sind. Es kann dann die Verteilung dieser Top-Level-Objekte auf die Remote-Knoten veranlaßt werden, wo sie noch nicht vorhanden sind (Verteilung von Objekten siehe Kapitel 6). Nachdem das Vorhandensein der Parameter auf den Remote-Knoten sichergestellt ist, kann die Kodierung der Methode an die Objektreplikate verteilt werden. Diese Verteilung kann ebenfalls in die lokale Sende-Warteschlange gestellt werden.

Aufgaben auf dem Remote-Knoten

Bei Ankunft einer kodierten Methode wird diese erst einmal (samt zugehöriger Objekt-Id) in die absenderspezifische Warteschlange gestellt. Sobald die Methodenverteilung an der Reihe ist, wird das zur OID gehörige Generische Objekt gesucht und auf diesem die Methode `remote_execution(CodeBuffer *buf)` aufgerufen, wobei buf die kodierte Methode enthält. Diese zerlegt die Kodierung zuerst in die Methoden-Id und die Kodierung der Parameter. Es wird dann eine virtuelle Methode aufgerufen, die anhand der in der Methoden-Id enthaltenen Base Class Id das richtige Basisobjekt findet, auf dem die Methode dann ausgeführt wird. Zur Ausführung der Methode müssen zunächst die Parameter der Methode dekodiert werden. Dann kann die Methode ausgeführt werden. Ein Beispiel für die Kodierung einer Methode und den Aufruf der Methode auf den Remote-Knoten ist in Anhang B zu finden.

4.3.5 Probleme der Methodenverteilung

Die Methodenverteilung bietet leider auch einige massive Probleme.

Die Wahrung der Konsistenz kann **NICHT** garantiert werden. Da Aufrufe von normalen Funktionen von SWS-Methoden aus nicht untersagt werden sollen (um z.B. Library-Funktionen nutzen zu können), ergeben sich hier massive Sicherheitsmängel. Wird während einer verteilt laufenden Methode auf irgendwelche programmglobale Variable zugegriffen, so sind deren Inhalte i.a. auf den verschiedenen Knoten nicht gleich. Die Folge sind unterschiedliche Ergebnisse von Berechnungen bzw. sogar unterschiedliche Abläufe der verteilten Methode. Desweiteren ergeben Aufrufe irgendwelcher IO-Operationen sehr undefinierte Ergebnisse. Wird zum Beispiel von einem File gelesen, kann dieses dank der Asynchronität des Ablaufs der Methoden sehr verschiedene Inhalte haben. Zugriffe auf globale Variable und IO-Operationen sind daher in SWS-Methoden und davon (direkt oder indirekt) aufgerufenen Funktionen streng untersagt. Kontrolliert wird dies vom Programmgenerator nicht, dazu müßte ein vollständiger C++-Parser entwickelt werden.

In inhomogenen Systemen kann es eventuell zu Problemen bei Berechnungen mit Gleitkommazahlen kommen. Die Ergebnisse können auf unterschiedlichen Rechnern (je nach Berechnung) extrem voneinander abweichen.

Einige der vom Programmgenerator selbst generierten Methoden und die Methoden der Pointer-Elementarobjekte müssen bei verteilten Methoden „synchronisiert" werden. Ein einfaches Beispiel ist die Erzeugung eines neuen SWS-Objektes. Hier muß sichergestellt werden, daß die generierten TLOID's dieselben sind, da sonst eine logische Methodenausführung mit einem Befehl eine ganze Reihe von SWS-Objekten erstellt hätte.

4.3.6 Zusätzliche Datenverteilung

Motivation

Abbildung 4.4 zeigt eine mögliche Situation im SWS. Am SWS nehmen drei Knoten teil. Knoten I und II besitzen lokal ein Replikat von den Objekten 1 und 2, Knoten III besitzt nur ein Replikat von Objekt 2. Hier kann es bei Methodenverteilung zu einem sehr unangenehmen Effekt kommen. Annahme: Knoten I führt eine Methode auf Objekt 1 aus. Während dieser Methodenausführung kommt es zur Methodenverteilung. Die Methode wird also auf Knoten II verteilt. Die verteilte Methode ruft eine Methode auf Objekt 2 auf (z.B. über den vorhandenen Pointer), welche das Objekt ändert. Diese Änderungen werden auf den Knoten I und II von der verteilten Methode durchgeführt — das Objekt auf Knoten III wird aber nicht geändert.

Als Lösungen bieten sich die drei folgenden Ansätze an:

- Aufruf der von Objekt 1 verteilten Methode auch auf Knoten III
- Verteilung der Methode, welche Objekt 2 ändert, auf Knoten III

- Verteilung der Änderungen an Objekt 2 an Knoten III via Datenverteilung

Der erste Ansatz bedingt die Verteilung von Objekt 1 auf Knoten III, damit die Methode dort auch aufgerufen werden kann. Voraussetzung ist allerdings, daß das Objekt 1 in dem Zustand übertragen wird, in dem es sich vor der Ausführung der verteilten Methode befand. Dies würde bedeuten, daß von den Objekten alte Versionen gehalten werden müssen, was als sehr umständlich erscheint.

Ansatz zwei ist in dieser Hinsicht unproblematisch. Objekt 2 ist auf Knoten III ja schon vorhanden, es besteht kein Bedarf an ursprünglichen Versionen von Objekt 1. Problematisch ist hier allerdings, wenn die Methode, welche auf Objekt 2 aufgerufen wird, eine Referenz auf Objekt 1 als Parameter erhält. Objekt 1 läßt sich zum Aufrufzeitpunkt in korrektem Zustand verteilen. Nach Beendigung der Methode auf Objekt 2 läuft die ursprünglich verteilte Methode weiter — allerdings nur auf den Knoten I und II, auf Knoten III nicht (siehe Abbildung 4.5). Änderungen, die dann noch an Objekt 1 durch die verteilte Methode durchgeführt werden, werden auf Knoten III nicht durchgeführt.

Die in dieser Arbeit gewählte Lösung ist der dritte Ansatz. Änderungen, die während einer verteilt laufenden Methode auf Objekten gemacht werden, welche auch auf Knoten existieren, die keine Remote-Knoten sind, werden an diese Knoten nach Beendigung der verteilten Methode datenverteilt. Im konkreten Beispiel würden also die in Objekt 2 geänderten Daten, nach Beendigung der ursprünglich verteilten Methode, an das Replikat des Objektes auf Knoten III per Datenverteilung übertragen. Diese Art der Verteilung wird **zusätzliche Datenverteilung** genannt.

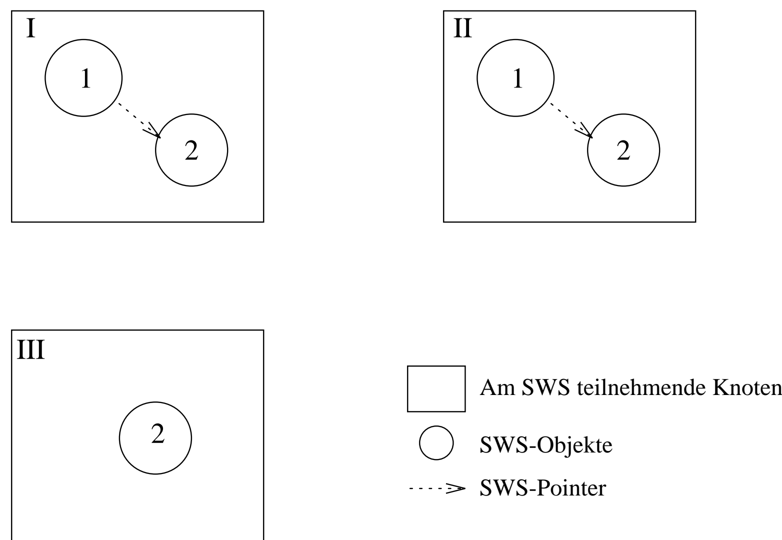


Abbildung 4.4: Ungünstige Situation für Methodenverteilung

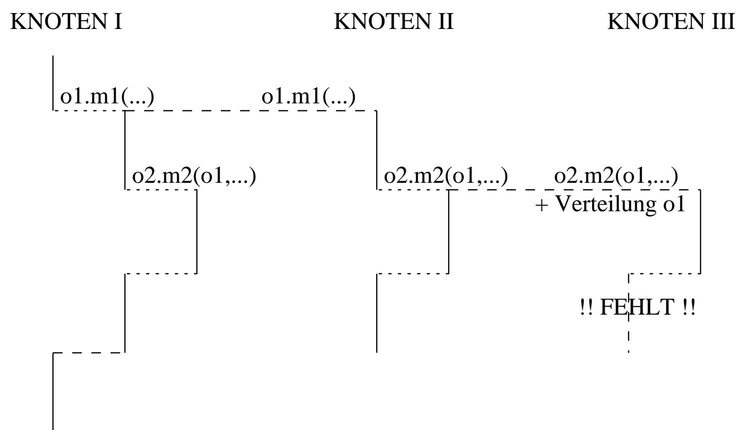


Abbildung 4.5: Problematische zusätzliche Methodenverteilung

Durchführung

Die Durchführung der zusätzlichen Datenverteilung ist der normalen Datenverteilung sehr ähnlich. Auch hier „sammeln“ die Elementarobjekte die geänderten Daten. Im Gegensatz zur normalen Datenverarbeitung müssen die zu verteilenden Daten allerdings während der Durchführung einer Methodenverteilung gesammelt werden. Prinzipiell können dafür dieselben Strukturen wie für die normale Datenverteilung verwendet werden (normale und zusätzliche Datenverteilung müssen garantiert nie parallel durchgeführt werden). Man kann hier zwischen zwei Vorgehensweisen wählen. Entweder sammelt man die Änderungen immer und entscheidet am Ende der Methodenverteilung für jedes der geänderten (Top-Level-)Objekte, ob die Änderung an zusätzliche Knoten verteilt werden muß, oder man entscheidet dies bei der ersten Änderung innerhalb eines Top-Level-Objektes und merkt sich diese Entscheidung in der TLI in einem Flag, dem **Verteilungsflag**.

Die Entscheidung fiel für die zweite Vorgehensweise. Bei beiden Vorgehensweisen muß die Überprüfung einmal stattfinden. Entscheidend war der Aufwand, der betrieben werden muß, um sich zu merken, daß ein Objekt geändert wurde. Im zweiten Ansatz muß dies nur gemacht werden, wenn wirklich eine zusätzliche Datenverteilung notwendig ist. Das Verteilungsflag kann drei Inhalte haben:

- Noch nicht geprüft : Wird ein Elementarobjekt während einer laufenden Methodenverteilung geändert, so muß geprüft werden, ob es Knoten gibt, auf denen ein Replikat des zugehörigen TLO's existiert und auf die die Methode nicht verteilt wurde. Ist dies der Fall, so muß zusätzliche Datenverteilung stattfinden. Die Empfängerknoten dieser zusätzlichen Datenverteilung werden bei der Überprüfung „automatisch“ ermittelt und in einer Liste im TLI gespeichert. Das Ergebnis der Prüfung wird im Flag gespeichert.

- Verteilung: die Änderung muß für zusätzliche Datenverteilung gemerkt werden

- keine Verteilung: Es ist keine Verteilung notwendig, Änderung muß nicht gemerkt werden

Diese Vorgehensweise bedingt allerdings eine Sonderbehandlung durch einige andere Methoden. Die Information, daß ein (Top-Level-)Objekt noch nicht geprüft wurde wird gesetzt, wenn ein Write-Lock auf dieses TLO gesetzt wird (immer, unabhängig ob während Methodenverteilung oder während Datenverteilung). Werden während einer Methodenverteilung zusätzliche Replikate eines Objektes auf beliebigen Knoten erstellt, so muß je nach Inhalt des Verteilungsflags differenziert werden:

- Es wurde noch nicht auf zusätzliche Datenverteilung geprüft: Das Verteilungsflag ändert sich nicht.

- Es muß nicht verteilt werden: Das Verteilungsflag wird auf ungeprüft gesetzt. Bei der nächsten Änderung muß dann neu entschieden werden.

- Es muß verteilt werden: Der Knoten wird zu der Liste der Knoten, auf die das Objekt zusätzlich datenverteilt wird, hinzugefügt.

Diese gesamte Verwaltung der zusätzlichen Datenverteilung wird nur auf dem Initiator der zugehörigen Methodenverteilung durchgeführt, die Remote-Knoten müssen sich darum nicht kümmern.

Für die zusätzliche Datenverteilung stehen zwei Methoden des Objektmanagers zur Verfügung, die äquivalent zu den Methoden für die normale Datenverteilung verwendet werden:

```
OM::additionalDataDistribution();
```

für die Verteilung am Ende der Methodenverteilung und

```
OM::additionalDataDistribution(TLI *tli);
```

für die Verteilung bei der Aufhebung eines Write-Locks eines TLO's durch eine Unlock-Methode. Diese Methoden verteilen die Datenänderungen aber nur auf die Knoten, auf die die zusätzliche Datenverteilung gemacht werden muß. Die Verteilung kann bei Verwendung einer Sende-Warteschlange in diese eingereiht werden.

Für den empfangenden Knoten einer solchen zusätzlichen Datenverteilung ändert sich gegenüber der normalen Datenverteilung nichts.

Kapitel 5

Locking

Potentiell gleichzeitiger Zugriff mehrerer Benutzer (Applikationen) auf dieselben Daten erfordert gewisse Synchronisationsmechanismen zur Wahrung der Datenkonsistenz. Eine Implementierung geeigneter Mechanismen für den SWS ist Thema weiterer Arbeiten. Einen Überblick über mögliche Konzepte findert man in [Kaba93].

5.1 Ein einfacher Sperrmechanismus

Für eine Realisierung des SWS ist jedoch zumindest ein einfacher Mechanismus zur Synchronisation erforderlich. Es wird deshalb ein auf **Read-** und **Write-Locks** basierendes Sperrprotokoll implementiert. Dies ist ein pessimistisches Sperrprotokoll, bei dem derjenige, der auf die Daten zugreifen möchte, eine Sperre (Lock) auf die Daten (Objekte) besitzen muß. Ein Write-Lock erlaubt sowohl den schreibenden als auch den lesenden Zugriff, ein Read-Lock erlaubt nur lesenden Zugriff. Eine Sperre (Lock) auf ein Objekt wird durch eine **Lock-Operation** angefordert. Die Gewährung eines Locks hängt davon ab, ob der geforderte Lock mit eventuell schon bestehenden Locks *kompatibel* ist. Die Kompatibilität der Locks wird durch die Kompatibilitätsmatrix in Abbildung 5.1 definiert. Diese Matrix läßt die Grundidee des Sperrmechanismus erkennen: zu einem Zeitpunkt dürfen beliebig viele lesende Zugriffe auf ein Objekt stattfinden. Das Schreiben ist dagegen eine exklusive Operation, währenddessen kein anderer als der Schreiber Zugriff (weder lesend noch schreibend) auf das Objekt hat. Dies ist insofern sinnvoll, da ein Lesezugriff auf ein Objekt, das gerade von jemand geändert wird, i.a. keine sinnvollen Ergebnisse liefert. Eine Applikation kann gleichzeitig einen Write- und einen Read-Lock auf das Objekt besitzen.

5.2 Verklemmungen

Ein Problem des implementierten Sperrmechanismus sind Verklemmungen. Diese treten bei *zyklischem Warten* auf die Gewährung von Sperranforderungen auf, d.h. Knoten 1 wartet auf einen Lock, den momentan Knoten 2 besitzt; Knoten 2 wartet auf einen Lock,

		bestehender Lock		
		Keiner	R	W
angeforderter Lock	R	+	+	- $(+)_1$
	W	+	- $(+)_2$	-

R ...... Lesesperre
W ...... Schreibsperre
\+ ...... Anforderung kann gewährt werden
\- ...... Anforderung kann nicht gewährt werden
$(+)_1$ .... gewährt, falls Write-Lock der anfordernden Applikation gehört
$(+)_2$ .... gewährt, falls nur anfordernde Applikation Read-Lock besitzt

Abbildung 5.1: Kompatibilitätsmatrix für Lockanforderungen

den momentan Knoten 3 besitzt; ...; Knoten $n-1$ wartet auf einen Lock, den momentan Knoten n besitzt und Knoten n wartet auf einen Lock, den momentan Knoten 1 besitzt. Besteht diese Situation schon, kann sie nur gelöst werden, indem einer der Knoten auf die Lockanforderung verzichtet und den Lock, den er schon besitzt, freigibt. Es gibt vier Bedingungen, unter denen solche Verklemmungen auftreten können (siehe [Lag], Verklemmung von Betriebsmitteln):

- Alleinzugriff auf ein Objekt möglich
- Entzug einer vergebenen Sperre nicht möglich
- partielle Zuteilung von Sperren möglich
- zirkuläres Warten möglich

Verklemmungen können verhindert werden, wenn eine dieser vier Bedingungen konstruktiv verhindert wird. Die ersten beiden Bedingungen sind nicht verhinderbar, sie sind für das Sperrprotokoll entscheidend. Die dritte Bedingung muß wohl zuerst näher erläutert werden. Es bestünde die Möglichkeit, Sperren nach dem „alles-oder-nichts"-Prinzip zu vergeben: Vor einer Operation müssen alle benötigten Sperren auf einmal gesetzt werden. Diese werden entweder alle gesetzt oder es wird keine gesetzt. Das Anfordern neuer Sperren ist erst nach Freigabe aller Sperren möglich. Für den SWS ist diese Art der Vermeidung nicht sehr geeignet, da eventuell wesentlich mehr Sperren gehalten werden, als im Moment nötig sind. Dies schränkt die gleichzeitige Bearbeitung der Daten im SWS ein.

Die Verhinderung des Eintretens der vierten Bedingung bietet einen im SWS gangbaren Ansatz. Die Operation des Setzens einer Sperre kann man auf zwei verschiedene Arten implementieren. Entweder dauert die Operation so lange, bis der gewünschte Lock gesetzt ist oder die Operation versucht nur, den Lock zu setzen und meldet das Ergebnis des Versuchs an die Anwendung zurück. Beim ersten Ansatz sind Verklemmungen (*Dead-Locks*) unvermeidbar, beim zweiten Ansatz kann zumindest der Applikationsprogrammierer auf die

Verklemmungssituation reagieren. Sperroperationen im SWS werden deshalb auf die zweite Art implementiert. So werden Verklemmungen durch Operationen des SWS vermieden, die Interpretation des Ergebnisses der Operation bleibt dem Anwendungsprogrammierer vorbehalten.

5.3 Schnittstelle zur Anwendung

5.3.1 Sperroperationen

Das Setzen von Sperren ist Aufgabe der Applikation (siehe auch Abschnitt 6.10). Vom Objektmanager werden dazu sogenannte **Lock-Operationen** (zum Setzen der Sperren) und **Unlock-Operationen** (zum Freigeben der Sperren) zur Verfügung gestellt. Weiterhin existieren Operationen zum Testen von Sperren. Die in einer weiteren Arbeit zu realisierenden mächtigeren Algorithmen zur Synchronisation der Zugriffe werden in einem speziellen Transaktionsmanager verwaltet werden, die Integration der hier implementierten Sperrmechanismen in den Objektmanager ist nur als Übergangslösung gedacht.

Sperren gelten immer für ein komplettes Top-Level-Objekt, werden aber als Sperren auf ein im TLO enthaltenes Objekt angefordert, d.h. eine Sperroperation kann auf ein beliebiges SWS-Objekt angewendet werden, das Ergebnis betrifft aber das gesammte Top-Level-Objekt. Wird beispielsweise ein Member eines Objektes gesperrt, so ist das Objekt selbst (mitsamt allen „umgebenden“ Objekten) auch gesperrt. Dies vereinfacht das Lockmanagement erheblich, bietet für den Anwendungsprogrammierer aber so manche Überraschung, da durch eine Unlock-Operation eventuell mehr Sperren freigegeben werden, als beabsichtigt wurde.

Lock-Operationen

Eine Lock-Operation hat prinzipiell zwei Argumente:

- Welche Sperre soll gesetzt werden?
- Was soll gesperrt werden?

Das erste Argument ist einsichtig, entweder wird ein Read- oder ein Write-Lock gesetzt. Das zweite Argument ist eigentlich auch klar — ein Objekt soll gesperrt werden. Es bietet sich jedoch die Möglichkeit an, daß mit einer Operation mehrere Objekte gesperrt werden. Wenn zum Beispiel eine mittels SWS-Pointern verkettete Liste von SWS-Objekten durchsucht werden muß, ist es mühselig, jedes dieser Objekte zum Lesen einzeln zu sperren, ein Sperren der gesamten Liste auf einmal (und auch entsprechendes Freigeben) wäre wünschenswert. Es werden deshalb zwei verschiedene Typen von Operationen zum Setzen von Sperren angeboten. Eine Operation zum Setzen einer einzelnen Sperre und eine Operation zum gleichzeitigen Sperren mehrerer Objekte (Top-Levl-Objekte). Das Ergebnis

einer Sperre auf ein einzelnes Objekt ist eindeutig, entweder konnte die Sperre gesetzt werden oder sie konnte nicht gesetzt werden. Die Lock-Operation auf mehrere Objekte kann ähnlich eindeutig gestaltet werden, wenn sie nach dem „alles-oder-nichts"-Prinzip implementiert wird. Dieses ist jedoch sehr unflexibel. Besser ist der Ansatz, daß aus der Menge der zu sperrenden Objekte so viele wie möglich gesperrt werden. Das Ergebnis der Lock-Operation muß dann allerdings darüber informieren, welche der Objekte gesperrt werden konnten. Zu diesem Zweck wird der Begriff **Lock-Handle** eingeführt. Ein Lock-Handle ist das Ergebnis einer Lock-Operation, sowohl bei Sperrung eines einzelnen Objektes als auch bei Sperrung mehrerer Objekte. Dieses Lock-Handle enthält die Menge der bei der Lock-Operation zu sperrenden Objekte, den Typ der Sperre und die Information, welche der zu sperrenden Objekte gesperrt werden konnten.

Der Programmgenerator benötigt vom Objektmanager nur die Operation zum Sperren eines Objektes:

```
LockHandle OM::lockSingleObject(
         GO &goref,     // Referenz auf zu sperrendes Objekt
         LockType ltype // Write-Lock oder Read-Lock
);
```

Zum Sperren mehrerer Objekte kann man sich mehrere verschiedene Operationen vorstellen, die sich vor allem in der Bestimmung der Menge der zu sperrenden Objekte unterscheiden. Ein Vorschlag ist z.B. das Sperren eines Ausschnittes des Objektgraphen durch Angabe eines Objektes und eines Radius. Ein Beispiel zeigt Abbildung 5.2. Alle (Top-Level-)Objekte, die von einem angegebenen (Top-Level-)Objekt (im Beispiel das Objekt 1) über eine angegebene Anzahl SWS-Pointer (hier 2) erreichbar sind, werden gesperrt.

```
LockHandle OM::lockMultipleObjects(
         GO &goref,      // Referenz auf "Objekt im Mittelpunkt"
         unsigned radius // zu sperrender Radius
         LockType ltype  // Write-Lock oder Read-Lock
);
```

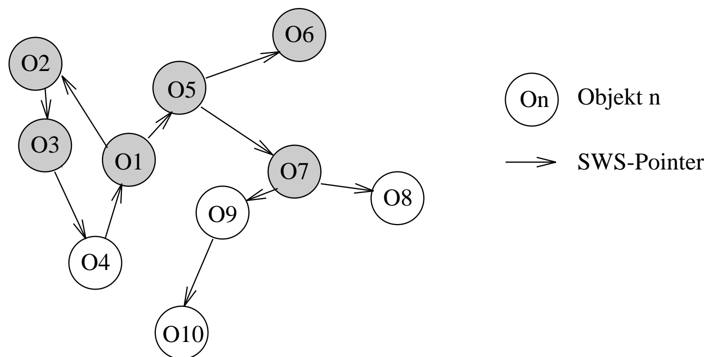


Abbildung 5.2: Sperrung der Region um Objekt 1 mit Radius 2

Die Aufgabe dieser Operation ist zuerst die Ermittlung der Menge der zu sperrenden Objekte. Auf dem sperrenden Knoten nicht vorhandene Objekte müssen dazu angefordert werden. Der Algorithmus zum Sperren der Objekte wird in Abschnitt 5.4 beschrieben.

Unlock-Operation

Die Unlock-Operation ist für das Freigeben der Sperren zuständig. Im Normalfall werden Objekte, auf die gemeinsam eine Sperre angefordert wurde, auch wieder gemeinsam freigegeben. Die Unlock-Operation bekommt deshalb als Argument ein Lock-Handle übergeben.

```
OM::unlockObjects( LockHandle lhandle);
```

Es werden alle Sperren freigegeben, die die Lock-Operation, die dieses Lock-Handle als Ergebnis hatte, gesetzt hatte. Auf ein Lock-Handle darf nur einmal eine Unlock-Operation durchgeführt werden.

Ein mögliches Problem sind „verlorene“ Lockhandles, d.h. Lockhandles die nie freigegeben worden sind, aber für das Programm nicht mehr zugreifbar sind. Die mit diesem Lock-Handle assoziierten Locks könnten bis zur Beendigung der Applikation nicht mehr freigegeben werden. Die Lock-Handles werden deshalb so implementiert, daß diese Locks automatisch bei der Destruktion der letzten vorhandenen Kopie des Lock-Handles freigegeben werden.

Test-Operationen

Der Objektmanager stellt dem Anwendungsprogrammierer zumindest zwei Testoperationen zur Verfügung:

```
bool testLockExact(GO &oref, LockType ltype);
```

welche testet, ob die Applikation einen Lock vom Typ *ltype* auf das mittels *oref* referenzierte Objekt besitzt, und

```
bool testLockCompatible(GO &oref, LockType ltype);
```

welche testet, ob die Applikation einen zum Locktyp *ltype* kompatiblen Lock auf das mittels *oref* referenzierte Objekt besitzt. Die Kompatibilität bei diesem Test wird durch die Kompatibilitätsmatrix in Abbildung 5.3 definiert. Bei diesen Testfunktionen muß bei der

		bestehender Lock		
		Keiner	R	W
getesteter Lock	R	-	+	+
	W	-	-	+

Abbildung 5.3: Kompatibilitätsmatrix für Test auf kompatiblen Lock

Implementation darauf geachtet werden, daß sie bei Methodenverteilung das richtige Ergebnis liefern. Wird eine Methode verteilt, so muß auf den Remote-Knoten geprüft werden, ob der Initiator der Methodenverteilung den entsprechenden Lock besitzt. Der Lock, den der Remote-Knoten auf das Objekt hat ist in diesem Fall nicht relevant.

Eine weitere Test-Operation wird für die Verwaltung der Objekte bei der Neuverteilung eines Objektes auf einen Knoten benötigt. Sie testet durch Anfrage bei einem als Parameter übergebenen Knoten, ob er in diesem Moment für ein bestimmtes Objekt einen Read-Lock besitzt. Diese Operation wird nur aufgerufen, falls dies lokal nicht entscheidbar ist. Applikationsprogrammierer dürfen diese Operation auf keinen Fall verwenden, da sie abhängig von der Zeit unterschiedliche Ergebnisse liefert. Eine Ausführung in einer verteilten Methode würde unterschiedliche Ergebnisse liefern.

5.3.2 Benötigte Sperren bei Methodenaufrufen

Eine Frage wurde bis jetzt noch nicht einmal angeschnitten: Wann wird auf das Vorhandensein von Locks getestet? Ein Lock wird eigentlich erst beim Zugriff auf die Elementarobjekte relevant. Erst dort wird direkt auf die Daten des SWS-Objektes zugegriffen. Ändernde Methoden der Elementarobjekte lassen deshalb einen Aufruf der Methode nur zu, wenn die Applikation einen Write-Lock für das entsprechende Objekt besitzt. Bei den lesenden Methoden der Elementarobjekte wird differenziert. Werden sie von einer „lesenden“ SWS-Methode aufgerufen, so wird für das Objekt ein Read-Lock benötigt, bei Aufruf aus „sehenden“ SWS-Methoden heraus wird kein Lock benötigt. Ein Zugriff ohne richtig gesetzten Lock wird mit einer **Exception** quittiert. Die „Kontrolle“ nur zu diesem Zeitpunkt durchzuführen ist nicht ideal. Eine datenändernde Methode eines SWS-Objektes kann beispielsweise schon einiges geändert haben, bevor sie auf ein Objekt zugreift (z.B. auf einen ihrer Parameter), auf das kein entsprechender Lock gesetzt ist und im Moment auch keiner gesetzt werden kann. Ein Aufruf einer Methode dieses Objektes endet in einer Exception. Da im Sperrprotokoll keinerlei Transaktionsmechanismen vorgesehen sind, werden die bis jetzt durchgeführten Änderungen der datenändernden Methode nicht zurückgesetzt und das Objekt in eventuell inkonsistentem Zustand zurückgelassen.

Um solche Situationen etwas besser in den Griff zu bekommen ist eine Prüfung der Locks zu Anfang jeder SWS-Methode (Ausnahme: sehende Methoden) vorgesehen. In der SWS-Klassendefinition kann bei den Methoden für jeden Parameter, der ein SWS-Objekt darstellt, und für das Objekt, auf das die Methode aufgerufen wird, ein benötigter Lock angegeben werden. Der Applikationsentwickler weiß i.a., welche Locks für das Objekt selbst und die Parameter benötigt werden und kann diese Locks spezifizieren. Wird nichts angegeben, werden Default-Werte angenommen. Der Default-Wert für Parameter ist prinzipiell ein Read-Lock. Die Default-Werte für die Objekte, auf denen die Methode aufgerufen wurde, unterscheiden sich für verteilbare und nicht verteilbare Methoden. Bei den verteilbaren Methoden wird ein Write-Lock als Default verwendet, da eine verteilbare Methode ihr Objekt i.a. ändert. Bei den nicht verteilbaren Methoden wird der Read-Lock als Default verwendet.

Die genaue Syntax der Methodendefinitionen wird in Kapitel 7 vorgestellt.

5.3.3 Automatische Anforderung von Locks

Die Definition der benötigten Locks kann auch noch für einen weiteren Zweck genutzt werden. Anstatt bei Nichtvorhandensein eines benötigten Locks gleich mit einem Fehler (Exception) zu reagieren, kann versucht werden, den entsprechenden Lock automatisch anzufordern. Erst wenn eine Anforderung dieses Locks nicht erfolgreich war, wird eine Exception ausgelöst. Die automatisch angeforderten Locks werden am Ende der Methode, in der sie angefordert wurden, wieder freigegeben. Dieses Verhalten gilt nicht nur für die vom Programmgenerator generierten Methoden der SWS-Objekte. Auch die Elementarobjekte versuchen, den entsprechenden Lock anzufordern, falls dieser nicht gesetzt ist.

5.4 Der Lock-Algorithmus

Bei Systemen, wo die zu sperrenden Daten nicht repliziert sind, ist das Setzen von Sperren relativ einfach. Die Sperre muß nur dort gesetzt werden, wo sich die Daten befinden. Im SWS ist dies komplizierter. Hier muß die Lockinformation jeweils vollständig bei den Replikaten des Objekts vorhanden sein. Ein einfacher Algorithmus zur Bewältigung dieses Problems und dessen Erweiterungen in Bezug auf den SWS wird in den folgenden Abschnitten vorgestellt.

5.4.1 Verteiltes Locking

In [KLL+91] wird ein Lock-Algorithmus für replizierte Objekte vorgestellt. Die Sperrinformation wird dort nicht verteilt, sondern nur lokal bei den Replikaten gesetzt. Es wird nur sicher gestellt, daß nicht bei zwei Replikaten gleichzeitig eine Sperre gesetzt ist. Im SWS soll die Information über auf ein Objekt gesetzte Sperren bei jedem Replikat vorhanden sein. Dies hat den Vorteil, daß sich relativ einfach lokal feststellen läßt, ob eine Sperre gesetzt werden kann oder ob dies im Moment nicht möglich ist. Das in [KLL+91] beschriebene Sperrprotokoll läßt sich deshalb nur in etwas abgeänderter Form in den SWS integrieren. Eine Sperre eines Objektes hat drei Zustände *FREE, PRELOCKED* und *LOCKED*. Abbildung 5.4 zeigt das Zustandsdiagramm. Fordert ein Knoten einen Lock auf ein Objekt an, so wird als erstes lokal überprüft, ob der gewünschte Lock gewährt werden kann (siehe Abbildung 5.1, bei den bestehenden Locks wird an dieser Stelle kein Unterschied zwischen Lock und Prelock gemacht) Ist dies der Fall, so wird der lokale Zustand des Locks auf Prelock gesetzt und an alle Replikate eine Prelock-Anforderung verschickt. Die Replikate untersuchen lokal, ob der Prelock auf dieses Objekt gewährt werden kann oder nicht und schicken das Ergebnis der Prüfung zum Absender der Anforderung zurück. Sind alle Antworten positiv, so ist der Lock akzeptiert. Daraufhin wird der Prelock lokal in einen Lock verwandelt und eine Nachricht an alle Replikate geschickt, daß der Prelock in einen Lock

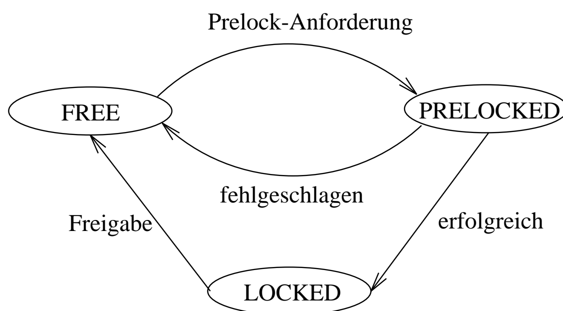


Abbildung 5.4: Mögliche Zustände eines Locks auf ein Objekt

gewandelt werden soll. Kommt jedoch von einzelnen Replikaten die Nachricht, daß der Prelock nicht möglich ist, so ist der Lockversuch fehlgeschlagen. In diesem Falle haben zwei Knoten gleichzeitig versucht, zwei sich gegenseitig ausschließende Locks zu beantragen. Der Prelock wird lokal gelöscht, alle anderen Replikate werden beauftragt, den Prelock ebenfalls zu löschen.

Abbildung 5.5 zeigt links einen erfolgreichen Lockversuch, bei dem Knoten 2 das Objekt sperrt. Auf der rechten Seite probieren Knoten 1 und 3 parallel, einen Lock (z.B Write-Lock) auf das Objekte zu bekommen, worauf keiner der beiden Knoten einen Lock bekommt.

Das Aufheben der Sperre geschieht durch eine einfache Nachricht an die Replikate des Objekts.

5.4.2 Asynchrones Locking

Bei genauer Betrachtung des dargestellten Algorithmus fällt auf, daß eigentlich anstatt eines Prelocks gleich ein Lock gesetzt werden könnte. Dies würde bei erfolgreicher Lockanforderung die Umwandlung des Prelocks in einen Lock ersparen. Der Algorithmus kann aber in der dargestellten Form immer noch nicht im SWS verwendet werden. Die Ausführung der Daten- und Methodenverteilungen werden bei den Objektmanagern in Warteschlangen gestellt. Der Lockalgorithmus muß an diese Praxis angepaßt werden. Außerdem müssen Lockanforderungen während Methodenverteilungen gesondert behandelt werden.

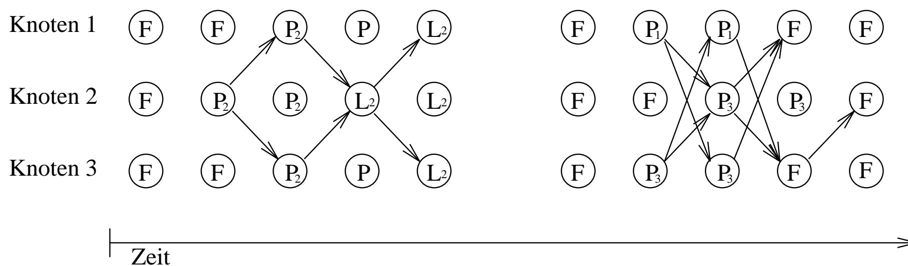


Abbildung 5.5: Erfolgreicher und nicht erfolgreicher Lockversuch auf ein Objekt

Lockanforderungen, die nicht während einer Methodenverteilung auftreten, sind der einfachere Fall. Sie werden vom Prinzip her wie schon beschrieben durchgeführt. Die Aufforderung zur Umwandlung wird jedoch bei den Knoten der anderen Replikate in die senderspezifische Warteschlange eingereiht. Lockanforderungen während einer Methodenverteilung sind etwas aufwendiger. Der Lock-Algorithmus als solches wird auf dem Initiator-Knoten der Methodenverteilung durchgeführt. Nach der lokalen Prüfung ob die Locks möglich sind, verteilt er als erstes die zu lockenden Objekte auf die Remote-Knoten, falls sie dort noch nicht lokal vorhanden sein sollten. Anschliessend verschickt er die Prelock-Anforderungen an alle Relikate der zu sperrenden Objekte (einschliesslich der Replikate auf den Remote-Knoten). Falls der Prelock erfolgreich war, wird an alle Replikate, die nicht auf Remote-Knoten sind, eine Aufforderung zur Umwandlung des Prelocks in einen Lock versandt, welche in die senderspezifische Warteschlange gestellt wird. An die Remote-Knoten wird ein kodiertes Lock-Handle verschickt, das ebenfalls in die senderspezifische Warteschlange gestellt wird. Die Lock-Operationen auf den Remote-Knoten entnehmen der Warteschlange das Lockhandle (gegebenenfalls müssen sie warten, bis es eintrifft) und wandeln entsprechend dem Inhalt des Lock-Handles die Prelocks auf dem Remote-Knoten in Locks um. Im Lockhandle ist vermerkt, welche Objekte mit der Lockoperation gesperrt wurden. Für diese Objekte muß je ein Prelock in einen Lock umgewandelt werden.

Unlock-Operationen sind vom Prinzip ähnlich. Es werden an alle Replikate der Objekte, auf denen Locks freigegeben werden sollen, entsprechende Nachrichten verschickt. Diese werden ebenfalls in die absenderspezifischen Warteschlangen gestellt. Bei Unlock-Operationen während einer Methodenverteilung wird ähnlich wie bei der Lock-Operation verfahren. Zu allen Replikaten, die sich nicht auf Remote-Knoten befinden, wird eine Unlock-Nachricht verschickt. Auf den Remote-Knoten werden diese Sperren von den dort laufenden Unlock-Operationen direkt freigegeben.

Der vollständige Algorithmus ist in Anhang C aufgeführt.

Warum alle diese Umstände? Die Einreihung der Unlock-Operation in die Warteschlange bzw. die Sonderbehandlung bei Methodenverteilung ist relativ einfach einzusehen: der Lock auf das Objekt, den der ändernde Knoten hält, darf bei den Replikaten erst dann freigegeben werden, wenn alle Änderungen, die während diesem Lock durchgeführt wurden, auf den Replikaten vollständig nachgeführt wurden. Angenommen, Knoten I ändert Objekt 1, die Änderung wird datenverteilt — unter anderem auf Knoten II, welcher auch ein Replikat des Objektes besitzt. Die Datenverteilung wird in die Warteschlange gestellt. Knoten I gibt den Lock frei und verschickt an Knoten II einen Unlock-Auftrag. Führt Knoten II nun diesen Unlock aus, bevor die Datenverteilung der Änderung auf das Objekt ausgeführt wurde (die Warteschlange kann theoretisch beliebig lang sein!), so könnte dies Konflikte geben. Knoten II könnte jetzt einen Lock auf das Objekt anfordern und noch vor der Ausführung der Datenverteilung etwas an dem Objekt ändern. In diesem Fall würden die durch Knoten II gemachten Änderungen bei Ausführung der Datenverteilung von Knoten I eventuell überschrieben. Bei Methodenverteilung ist der Fall ähnlich gelagert. Wird der Lock erst nach der Methodenverteilung freigegeben, so darf er auf den Remote-Knoten auf keinen Fall freigegeben werden, bevor die verteilte Methode dort zu

Ende gelaufen ist, da sonst eventuell benötigte Locks nicht mehr vorhanden wären. Durch die Einreihung der Unlock-Aufforderung in die Warteschlange ist gesichert, daß der Lock nicht zu früh freigegeben wird. Bei einer Unlock-Operation während einer Methodenverteilung ist dies durch die gesonderte Behandlung auf den Remote-Knoten sichergestellt. Der Lock wird dort, relativ zum Ablauf der Methode, zum gleichen Zeitpunkt wie auf dem Initiator freigegeben.

Bei den Lock-Anforderungen ist die Erklärung noch eine Stufe komplizierter. Hier liegt das Problem vor allem bei den Methodenverteilungen. Während einer Methodenverteilung müssen die Lockverhältnisse für die verteilten Methoden dieselben sein wie für die Methode auf dem Initiatorknoten. Die Tests auf Locks und die Lock-Operationen müssen auf den Remote-Knoten dieselben Ergebnisse liefern wie beim Initiator. Würden die Aufforderungen zur Umwandlung von Locks sofort ausgeführt, so könnten sie eventuell eine in der Warteschlange stehende Methodenverteilung „überholen“ — für die verteilte Methode wäre dann ein Lock gesetzt, der aus der Zukunft kommt. Werden während einer Methodenverteilung Locks angefordert, so dürfen diese auf den Remote-Knoten auch erst gesetzt werden, wenn die entsprechende Lock-Operation ausgeführt wird. Aus diesem Grund erfolgt die Unterscheidung in Prelocks und Locks. Ein Prelock eines Knotens auf ein Objekt bedeutet, daß der Knoten einen Lock auf das Objekt im Moment anfordert oder daß die Anforderung schon gewährt wurde, in der Warteschlange aber noch Aufträge stehen, die vor der Lockoperation verschickt wurden. Für Lockanforderungen anderer Knoten auf ein Objekt wird deshalb nicht zwischen Prelock und Lock unterschieden. Für verteilte Methoden jedoch sind die Prelocks unsichtbar, ein Prelock zählt für einen während einer Remote-Methode ausgeführten Lock-Test nicht als Lock.

Ein Beispiel bringt vielleicht etwas Licht ins Dunkel. Eine verteilbare Methode auf Objekt 1 wird methodenverteilt. Im Laufe dieser Methode wird getestet, ob auf das Objekt 2, auf welches ein SWS-Pointer des Objektes 1 zeigt, ein Read-Lock gesetzt ist. Wenn nicht, so wird dieser Lock angefordert. Würde auf den Remote-Knoten nun bei dieser Lock-Anforderung der Lock gesetzt bevor die Ausführung der verteilten Methoden an der Stelle des Testes angekommen wäre, dann würde der Test auf den Remote-Knoten anders ausfallen wie auf dem Initiator. Die Remote-Ausführung würde anders verlaufen, was i.a. auf den Remote-Knoten zu anderen Ergebnissen führen würde als beim Initiator. Beim hier beschriebenen Algorithmus fällt der Test auf den Remote-Knoten gleich aus. Der gesetzte Prelock wird beim Test nicht als Lock interpretiert. Der Lock wird für die Remote-Methoden erst nach der Lock-Operation innerhalb der Methode sichtbar.

Erwähnenswert in diesem Zusammenhang ist ein etwas ungewöhnliches Phänomen. Wenn bei Knoten II die absenderspezifische Warteschlange von Knoten I sehr lange ist, kann es vorkommen, daß Knoten I einen Prelock für ein Objekt anfordert, für das auf Knoten II schon (noch ?) ein Prelock für Knoten I gesetzt ist, der durch einen der Einträge in der Warteschlange erst noch umgewandelt werden soll. Es muß deshalb gestattet werden, daß auf einem Knoten für einen anderen Knoten mehrere Prelocks auf ein Objekt gesetzt sind. Wäre dies nicht gestattet, so müßte die Prelock-Anforderung unnötigerweise zurückgewiesen werden.

Kapitel 6

Die Objektverwaltung

Die Objektverwaltung ist Aufgabe des Objektmanagers. In diesem Kapitel wird zuerst geklärt, welche speziellen Aufgaben in diesem Zusammenhang anfallen. Anschließend werden diese Aufgaben genauer beleuchtet und Algorithmen zu ihrer Bewältigung präsentiert. Den Abschluß bildet die Betrachtung der Aufgaben der Elementarpointer des Basissystems bei der Objektverwaltung.

6.1 Die Aufgaben des Objektmanagers bei der Objektverwaltung

Bei der Objektverwaltung fallen einige zum Teil recht komplexe Aufgaben an:

1. Bereitstellung der von der laufenden Applikation angeforderten/benötigten Objekte.
2. Auf Anforderung Verteilung von Objekten auf andere Knoten (z.B. Parameter bei Methodenverteilungen)
3. Bereitstellung eines Mechanismus zum Benennen von Objekten mit Namen. SWS-Objekte können mittels eines vergebenen Namens vom OM angefordert werden.
4. Hinzufügen neuer Objekte zum SWS.
5. Sicherstellen der Persistenz der SWS-Objekte.
6. Explizites Löschen von Objekten.

Diese einzelnen Aufgaben hängen alle mehr oder weniger zusammen. Die Bereitstellung der benötigten Objekte kann teilweise mittels Verteilungsanforderungen durchgeführt werden. Die Sicherstellung der Persistenz der Objekte spielt auch schon beim Hinzufügen von Objekten zum SWS eine Rolle.

Einige der hier aufgeführten Aufgaben benötigen die Objekte in einer „transportablen“ Form, um sie zum Beispiel zu anderen Knoten transportieren zu können oder sie auf

Sekundärspeicher zu speichern. Die Transformation in diese Form ist Inhalt des nächsten Abschnittes.

6.2 Kodierung von Objekten

Um Objekte über eine Kommunikationsverbindung zu transportieren wird eine dafür geeignete Form benötigt. Eine Kopie auf Bitebene scheidet aus. Der SWS soll auch in inhomogenen Netzen benutzt werden können. Da die Darstellung der Daten auf unterschiedlichen Rechnern i.a. nicht gleich ist (man denke nur an die Byte-Orders bei Integern) kann eine solche Kopie nicht funktionieren. Auch bei homogenen Systemen gibt es bei der Verwendung unterschiedlicher Compiler Schwierigkeiten.

Die Verwaltung der Objekte wird vom Objektmanager auf Top-Level-Objekt-Ebene durchgeführt. Es wird daher eine Kodierung für komplette Top-Level-Objekte benötigt. Bei der Kodierung kann man unterscheiden zwischen der Kodierung der Verwaltungsinformation (Top-Level-Objekt-Id's, Klassen-Id's, ...) und den eigentlichen Nutzdaten des Objektes. Diese Nutzdaten befinden sich in den im TLO enthaltenen Elementarobjekten. Eine einfache Kodierung dieser Daten kann man, analog zu der Kodierung bei der Datenverteilung, durch eine Erweiterung der Membertabelle in der TLI erreichen. Die Membertabelle wird um ein Flag erweitert welches angibt, ob der Member ein Elementarobjekt ist oder nicht. Um die Daten des TLO's zu kodieren muß nur die Membertabelle nach Elementarobjekten durchsucht werden und deren Inhalt in einen Puffer kodiert werden. Die Reihenfolge der Kodierungen im Puffer ist durch die Reihenfolge der Elementarobjekte in der Membertabelle definiert. Daher kann auf die Kodierung der MOID's verzichtet werden, was die Datenkodierung sehr kompakt macht. Die Dekodierung gestaltet sich ähnlich einfach, es müssen nur die entsprechenden Dekodierfunktionen der im TLO enthaltenen Elementarobjekte der durch die Membertabelle definierten Reihenfolge nach aufgerufen werden. Die Kodierung der Verwaltungsdaten ist nicht weiter schwierig, da diese an fester Stelle in der TLI stehen.

6.3 Die Objektfabrik

Der Objekt-Manager hat ein großes Problem — es müssen Objekte erstellt werden, die der Objektmanager nicht kennt. Ein Beispiel: Ein 3D-Zeichenprogramm hat eine sehr allgemeine Klasse *Körper* mit diversen virtuellen Funktionen. Eine dieser Funktionen könnte eine Skalierung sein. Die Skalierung hat ein Ergebnis, zum Beispiel die Koordinaten des kleinsten Quaders, der den Körper vollständig einschließt. Von dieser Klasse *Körper* gibt es zur Übersetzungszeit die zwei Spezialisierungen *Quader* und *Kugel*. Weiterhin gibt es eine Klasse *Körpermenge*, welche eine beliebige Menge Körper beinhalten kann. Einer der Member dieser Klasse ist der kleinstmögliche Quader, der die Menge der Körper vollständig umschließt. Auch diese Klasse besitzt eine Methode zur Skalierung, welche die entsprechenden Methoden der in der Menge enthaltenen Körper aufruft und aus den

Ergebnissen der Aufrufe den umschließenden Quader neu berechnet. Ein etwas weiter entwickeltes 3D-Zeichenprogramm setzt auf der gleichen Objektstruktur auf. Es besitzt aber mehr Spezialisierungen der Klasse *Körper*, zum Beispiel noch eine Klasse *Pyramide*. Es wird nun eine Zeichnung von den beiden Programmen gemeinsam bearbeitet (eventuell ist das ältere Programm einfacher zu bedienen). Diese Zeichnung enthält unter anderem auch eine Körpermenge. Diese wird beim älteren Programm im Moment nur durch den umgebenden Quader dargestellt. Mit dem neueren Programm wird eine Pyramide in diese Körpermenge eingefügt. Diese Änderung wird per Datenverteilung übertragen. Die Zeichnung enthält jetzt ein Objekt, das beim älteren Programm unbekannt ist. Die Körpermenge ist als Sammlung von Zeigern auf Körper implementiert. Solange auf die Pyramide nicht zugegriffen wird, besteht bei dem Knoten, auf dem das ältere Programm ausgeführt wird, kein Bedarf, von diesem Pyramidenobjekt lokal ein Replikat anzulegen. Nun soll aber diese Körpermenge neu skaliert (z.B. vergrössert) werden. Jetzt ist ein lokales Replikat der Pyramide dringend erforderlich. Das Objekt kann jetzt von dem Knoten, auf dem das neuere Programm abgearbeitet wird, angefordert werden. Das Ergebnis ist ein kodiertes Objekt, aus dem ein reales Objekt erstellt werden muß.

Dieses Beispiel zeigt gleich zwei Probleme:

- Eine Körpermenge enthält eine Menge von (Pointern auf) allgemeinen Körpern. Wird diese um ein Objekt erweitert, indem ein weiterer Zeiger auf einen Körper in die Menge aufgenommen wird, so wird bei Datenverteilung erst einmal dieser neue Zeiger verteilt. Erst bei Zugriff auf diesen Zeiger wird das entsprechende Objekt verteilt. Der genaue Typ dieses Objektes ist dabei aber nicht bekannt. Er kann aus der Kodierung des Objektes ermittelt werden. Der OM muß nun anhand dieses Typs ein Objekt erstellen können.
- Das obige Beispiel zeigt, daß hierbei eventuell Objekte erstellt werden müssen, deren Klasse es zur Übersetungszeit der Applikation noch nicht gab.

Die Lösung ist die sogenannte **Objektfabrik**. Diese Objektfabrik gehört zum Objektmanager und kann beliebige Objekte produzieren. Die Klasse eines zu produzierenden Objektes wird durch eine **Klassen-Id** festgelegt. Diese ID muß eindeutig sein und wird im Header der Klassendefinition angegeben. Zu jeder Klasse wird durch den Programmgenerator eine Funktion generiert, welche ein Objekt der Klasse erstellt und einen Zeiger auf die zugehörige Top-Level-Information zurückgibt. Diese Funktionen werden in der Objektfabrik mitsamt der zugehörigen Klassen-Id verwaltet. Ein Objekt kann erstellt werden, indem anhand der Klassen-Id die richtige Funktion herausgesucht wird und diese dann ausgeführt wird. Dies ist bei den in der Applikation verwendeten Klassen kein Problem, sie können beim Start des Programms durch geeignete Maßnahmen in die Objektfabrik eingetragen werden. Alle nicht bekannten Klassen werden durch die Objektfabrik dynamisch nachgeladen. Dazu muß der Code der Klassen für die Objektfabrik anhand der Klassen-Id erreichbar sein (zum Beispiel über DFS[1]).

[1] Distributed File System, Bestandteil des DCE

Eine weitere benötigte Funktion ist den Casts von C++ recht ähnlich. Die Aufgabe dieser Funktion ist die Umwandlung von Pointern einer Klasse in einen Pointer einer ihrer Unterklassen. Diese Umwandlung erfolgt anhand der Klassen-Id's und ist deshalb nur möglich, wenn die Unterklasse eindeutig ist. Diese Funktion wird ebenfalls durch die Objektfabrik zur Verfügung gestellt. Dazu wird durch den Programmgenerator zu jeder Klasse eine Funktion *doCast(void *, ClassId &)* definiert, welche einen Pointer auf diese Klasse anhand der Klassen-Id einer Unterklasse in einen Pointer auf die Unterklasse umwandelt (alles auf der Basis von void-Pointern). Diese Funktion wird zusammen mit der Funktion zur Objekterstellung in der Objektfabrik verwaltet.

Die von der Objektfabrik bereitgestellten Methoden

```
TLI *newObject(ClassId &cid);
```

Diese Funktion erstellt ein neues TLO der durch cid eindeutig bestimmten Klasse und gibt einen Pointer auf die TLI zuück.

```
void *doCast(void *obj, ClassId &cid1, ClassId &cid2);
```

Diese Methode wandelt einen void-Pointer auf ein Objekt der Klasse mit Klassen-Id cid1 (unter der Zuhilfenahme der Funktion *doCast(void *, ClassId &)* dieser Klasse, welche in der OF gespeichert ist) in einen void-Pointer auf ein Objekt mit Klassen-Id cid2 um und gibt diesen Pointer zurück. Die Klasse mit Klassen-Id cid2 muß dabei eine (eindeutige) Basisklasse der Klasse mit Klassen-Id cid1 sein. Ist dies nicht der Fall (keine Basisklasse oder keine eindeutige Umwandlung möglich), so wird **NULL** zurückgegeben. Ist b ein Zeiger auf ein Objekt der Klasse B und a ein Zeiger auf ein Objekt der Klasse A, welche eine Basisklasse von B ist, so kann die Zuweisung a=b mittels der Objektfabrik OF folgendermaßen ausgedrückt werden: a = (A*) OF.doCast((void *)b, cidB, cidA), wobei cidB die Klassen-Id von B und cidA die Klassen-Id von A ist. Dieses leider sehr unschöne Konstrukt wird zur Flexibilisierung der Anforderung von Objekten aus dem SWS benötigt.

```
newClass(ClassId cid, GO *(*makeObject)(),
         void *(*doCast)(void *, ClassId &));
```

Mit dieser Methode werden neue Klassen in der Objektfabrik eingetragen. Die Parameter sind die Klassen-Id der neu einzutragenden Klasse, die Funktion *makeObject(...)*, welche ein Objekt dieser Klasse produziert und die Funktion *doCast(void *, ClassId &)*, welche einen Pointer auf ein Objekt dieser Klasse in einen Pointer auf eine der Basisklassen umwandelt.

Diese Methoden dürfen durch den Anwender nicht verwendet werden, sie dienen alleine der Verwaltung der Objekte durch den vom Programmgenerator erzeugten Code und das Basissystem.

6.4 Objektpersistenz

SWS-Objekte sind persistente Objekte, d.h. ihre Lebensdauer endet nicht mit der letzten Applikation, die ein lokales Replikat des Objektes besitzt. Es muß also dafür gesorgt werden, daß Objekte in irgendeiner Form auf Sekundärspeicher gespeichert werden und von dort auch wieder geladen werden können. Zur Speicherung von Objekten gibt es mehrere verschiedene Möglichkeiten. Eine davon ist die Verwendung einer Objektorientierten Datenbank, in der die Objekte als Objekte abgelegt werden können. Eine andere Möglichkeit ist das „Ebnen" der Objekte in eine serielle Datenstruktur, welche dann in einer „normalen" Datenbank oder in Dateien abgelegt werden kann. In [Gro93] werden Objekte durch einen Programmgenerator um Methoden zum „Ebnen" des Objektes in einen Stream erweitert, womit das Objekt problemlos in ein File geschrieben werden kann. Einige Konstrukte kann der Generator jedoch nicht behandeln, diese müssen „per Hand" erweitert werden. Ausserdem werden für die Grundtypen die normalen Ausgaben auf einen Stream verwendet, was die Kodierung der Objekte unnötig groß macht.

Im SWS werden die Objekte „eingeebnet", die dazu notwendigen Kodierfunktionen wurden schon beschrieben. Durch den speziellen Aufbau der SWS-Objekte sind diese Funktionen vollständig im Basissystem integriert. Die kodierten Objekte werden jeweils in eine Datei gespeichert, auf die per DFS zugegriffen werden kann. Der Dateiname ist eine eindeutige Abbildung der TLOID auf einen im DCE erlaubten Dateinamen. In der Datei werden die zur Konstruktion des Objektes benötigten Verwaltungsinformationen und die Kodierung des Dateninhaltes des Objekts abgelegt. Die Verwaltungsinformationen beschränken sich hier auf die TLOID und die Klassen-Id des Top-Level-Objektes. Es werden jedoch noch zusätzliche Informationen benötigt. Wenn eine Applikation frisch startet und ein Objekt anfordert, so muß der dafür zuständige Objektmanager wissen, von wo das Objekt zu holen ist. Entweder gibt es schon Replikate auf anderen Knoten, dann muß das Objekt von dort angefordert werden. Existiert noch kein Replikat des Objektes im SWS, so muß es vom Sekundärspeicher gelesen werden. Da ein Sessionmanagement erst in weiteren Arbeiten zu realisieren ist, werden auf dem Sekundärspeicher zu jedem Objekt zusätzliche Informationen darüber abgelegt, auf welchen Knoten lokale Replikate des Objektes vorhanden sind. Sind keine Knoten abgelegt, so muß das Objekt von Datei gelesen werden, ansonsten von einem der angegebenen Knoten angefordert werden. Dies bedingt, daß ein neu erzeugtes SWS-Objekt bei Eintritt in den SWS sofort gespeichert werden muß, damit diese Knoten-Informationen sofort verfügbar sind.

Ansonsten wird das Speichern der Objekte von den Objektmanagern bei Beendigung einer Applikation automatisch erledigt. Der OM dieser Applikation speichert alle Objekte, auf die im Moment kein anderer Knoten einen Write-Lock hat, ab. Bei den Objekten, die er nicht abspeichern kann, muß er sich überzeugen, daß noch ein anderer Knoten existiert der das Objekt abspeichern kann. Ist dies nicht der Fall, dann kann das Objekt zwar gespeichert werden, befindet sich jedoch wahrscheinlich nicht in einem konsistenten Zustand (der Knoten der den Write-Lock hat ist während einer ändernden Methode zusammengebrochen, ...).

Das automatische Speichern der Objekte findet also nicht regelmäßig statt, ein Zusammenbruch des SWS könnte den Verlust vieler Daten nach sich ziehen. Der Objektmanager bietet dem Anwendungsentwickler daher eine Methode

```
bool saveObject(GO *obj);
```

zum Speichern des übergebenen (Top-Level-)Objektes an. Diese Methode aktualisiert nur die Daten des Files, in dem das Objekt abgespeichert ist. Ein Rücksetzen auf diesen gespeicherten Status ist nicht möglich, nach einem Zusammenbruch des SWS ist aber der zuletzt gespeicherte Zustand des Objektes verfügbar. Der Rückgabewert gibt an, ob das Speichern erfolgreich war (der beinahe einzige Grund (außer einem vollen Sekundärspeicher) für einen negativen Bescheid ist ein Write-Lock auf das Objekt durch einen anderen Knoten). Die Methode saveObject() wird während einer Methodenverteilung nur auf dem Initiator durchgeführt, das Ergebnis wird auf die Remote-Knoten verteilt.

6.5 Benennung von Objekten

Objekte können anhand ihrer Objekt-Id angefordert werden. Diese Praxis ist nicht sehr benutzerfreundlich. Wenn ein im SWS schon existierendes Objekt von einem Benutzer mit einer Applikation bearbeitet werden möchte, so muß dieses Objekt in irgendeiner Weise ermittelt werden. Dies soll unter anderem Aufgabe des Sessionmanagements werden. Bis dieses existiert, wird dem Applikationsentwickler die Möglichkeit gegeben, einem Objekt einen Namen zuzuordnen, anhand dessen das Objekt zu späterem Zeitpunkt vom SWS angefordert werden kann. Hier stellt sich die Frage, was benannt werden kann. Können nur tatsächlich vorhandene Objekte benannt werden oder auch deren Basisobjekte? Ein Beispiel zeigt Abbildung 6.1. Hier ist die Vererbungshierarchie der Klasse D abgebildet. Angenommen man hat eine Sammlung von Zeigern auf Objekte der Klasse C und möchte die Objekte, auf die die Zeiger zeigen, benennen. Die Zeiger zeigen zum Teil auch auf Objekte der Klasse D. Werden jetzt nur die tatsächlichen Objekte benannt (was im Endeffekt nichts anderes ist als eine Zuordnung eines Namens zu einer Objekt-Id), so wird die umgekehrte Aktion, das Anfordern anhand des Namens, recht schwierig, da man ja ein Objekt der Klasse D erhält. Dieses ließe sich mit der doCast-Methode der Objektfabrik in diesem Falle korrigieren, setzt man aber an die Stelle der Klasse C die Klasse A, so könnte auch

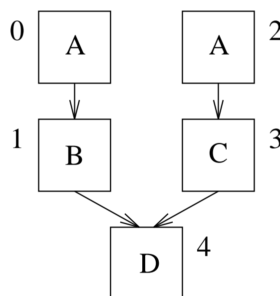


Abbildung 6.1: Vererbungshierarchie eines zu benennenden Objektes

diese Methode nicht weiterhelfen, da die Umwandlung eines *D's* in ein *A* nicht eindeutig ist. Hier hilft die Base Class Id aus der Klemme. Anhand der Objekt-Id und Base Class Id kann das gespeicherte „Objekt“ eindeutig bestimmt werden. Streng genommen wird dabei eigentlich kein Objekt benannt, sondern ein Zeiger auf ein Objekt. Um die Benennung ganz komfortabel zu machen, wird zusätzlich auch noch die entsprechende Klassen-Id mit abgespeichert (im Beispiel die Id der Klasse *C*). Damit ist es im Beispiel möglich, einen Zeiger auf ein Objekt vom Typ *C*, der auf ein Objekt der Klasse *D* zeigt zu benennen, und sich anhand dieses Namens einen Zeiger auf *A* geben zu lassen, der dann auf das über *C* erreichbare *A* im Objekt zeigt. Der OM kann dies längst nicht mehr ohne die Hilfe der zu speichernden Klassen, da er keinen Zugriff auf die Base Class Id's und die Klassen-Id's der zu speichernden Objekte hat (er kennt nur TLI's und Generische Objekte). Deshalb werden zu jeder Klasse zwei Methoden generiert, die ein Objekt benennen bzw. anhand des Namens vom SWS anfordern können. Die Methode zum Benennen generiert die eindeutige Kodierung des zu benennenden Objektes (OID,BaseClassId,ClassId). Der Objektmanager ist dann nur noch dafür verantwortlich, die Kodierung unter dem vergebenen Namen zu speichern und diese Kodierung auf Anfrage anhand des Namens wieder zur Verfügung zu stellen.

Ein möglicher Programmausschnitt einer Applikation könnte folgendermaßen aussehen:

```
D *d = D::newObject();
C *c = d;

c->nameObject("testname"); // Methode von C
.......
A *a = A::getObjectByName("testname"); // statische Methode von A
```

Die Klassenhierarchie zeigt Abbildung 6.1. A::getObjectByName() benötigt hier die Klassen-Id des Objektes, auf das der durch die gespeicherte Base Class Id ermittelbare Zeiger zeigt. Nur wenn die Klassen-Id mit abgespeichert wird ist es möglich, mittels der doCast-Methode der Objektfabrik den Zeiger auf *A* zu ermitteln.

6.6 Neuverteilung von Objekten

Als Neuverteilung eines Objektes wird die Erstellung eines Objekt-Replikates auf einem am SWS teinehmenden Knoten bezeichnet. Es werden nur komplette Top-Level-Objekte verteilt. Die Verteilung kann in unterschiedlichen Situationen geschehen:

- Die Anwendung fordert ein Objekt an.
- Bei einer Methodenverteilung werden die Parameter der zu verteilenden Methode, welche SWS-Objekte sind, auf die Remote-Knoten verteilt, auf denen bis jetzt noch keine Replikate dieser Objekte vorhanden sind

- Bei einer Lock-Anforderung während einer Methodenverteilung verteilt der Initiator die zu sperrenden Objekte auf die Remote-Knoten (sofern dort nicht schon Replikate der Objekte vorhanden sind)

- Während Methodenverteilung muß bei Zugriffen auf die Elementarpointer sichergestellt werden, daß die Ergebnisse auf den Remote-Knoten dieselben sind wie das Ergebnis beim Initiator der Methodenverteilung. Zu diesem Zweck müssen eventuell Objektverteilungen durchgeführt werden.

6.6.1 Die Verteilung auf einen anderen Knoten

Es wird also auf jeden Fall eine Methode benötigt, die ein Objekt auf einen anderen Knoten verteilt. Wird diese Methode mittels RPC von einem anderen Knoten aufgerufen, so kann damit auch eine Objektanforderung realisiert werden (Verteilung des Objekts auf den Knoten, der den RPC aufgerufen hat). Es ist deshalb nur ein Algorithmus für das Verteilen als solches notwendig. Dieser Algorithmus soll nun etwas näher erläutert werden.

Ein Problem der Verteilung eines Objektes sind zur Verteilung parallele Aktionen auf diesem Objekt. Ändert zum Beispiel ein Knoten etwas an dem zu verteilenden Objekt, so ist die konsistente Verteilung dieses Objektes nur vom ändernden Knoten aus möglich. Sollte also ein Knoten einen Write-(Pre-)Lock auf das zu verteilende Objekt besitzen, so wird dieser Knoten mit der Verteilung des Objektes beauftragt. Aber selbst dort muß noch aufgepaßt werden. Die Verteilung der Datenänderungen und die Neuverteilung des Objektes dürfen nicht parallel stattfinden, da eine korrekte Neuverteilung nicht garantiert werden könnte. Ein weiteres Problem sind Lockanforderungen bzw. das Freigeben von Locks während ein Objekt verteilt wird. Es ist relativ schwierig sicherzustellen, daß die Lockinformationen bei dem neu erstellten Replikat korrekt sind.

Um diesen Problemen aus dem Weg zu gehen, wird eine neue Art Sperre eingeführt, der **Distribution Lock**, kurz **D-Lock** genannt. Dieser Lock hat nichts mit den schon beschriebenen Locks zu tun. Er soll dazu dienen, daß während einer Neuverteilung keine anderen Verteilungen betreffs dieses Objekts (Lock-Operationen, Unlock-Operationen, Datenverteilungen, Methodenverteilungen, zusätzliche Datenverteilungen, Neuverteilung des gleichen Objektes von anderem Knoten aus) durchgeführt werden. Der D-Lock wird auf eine ähnlich Weise angefordert wie die anderen Locks. Der Knoten, der ein Objekt verteilen will schickt (nach lokaler Prüfung) eine D-Lock-Anforderung für das zu verteilende Objekt an alle anderen Objekt-Replikate. Diese setzen den D-Lock und schicken eine Antwort zurück. Der Lock ist dann gewährt. Läuft auf einem der Knoten gerade eine der Operationen ab, die mit dem D-Lock ausgeschlossen werden sollen, so wird der D-Lock erst am Ende dieser Operation gesetzt. Versuchen 2 Knoten gleichzeitig, einen D-Lock für dasselbe Objekt zu setzen, so wird der Lock für keinen von beiden gewährt, die schon gesetzten D-Locks werden wieder gelöscht. Das Objekt muß jedoch verteilt werden. Es wird daher kurze Zeit gewartet (die Länge der Wartepause ist zufällig) und der D-Lock erneut angefordert. Ist der D-Lock gesetzt, so kann mit der Verteilung begonnen werden. Da-

zu werden die Daten und die Verwaltungsinformationen des (Top-Level-)Objektes kodiert und mittels einem Objekterstellungsauftrag (z.B. in Form eines RPC's) an den Zielknoten der Verteilung geschickt. Auf dem Zielknoten wird nun anhand der Objektkodierung das Objekt erstellt. Der verteilende Knoten muß jetzt den Knoten, auf den das Objekt neu verteilt wurde, den anderen Objektreplikaten mitteilen und diesen Knoten auch auf Sekundärspeicher sichern (siehe Abschnitt über Objektpersistenz). Am Schluß wird noch der D-Lock freigegeben. Eine genaue Ausführung des Algorithmus und der hiermit verbundenen Algorithmen findet man in Anhang D.1.

6.6.2 Anforderung eines Objektes durch die Anwendung

Die Anwendung kann von ihr benötigte, schon existierende Objekte anfordern. Dies geschieht i.a. mittels statischer Methoden der SWS-Klassen (getObjectByName(...), ...). Diese fordern vom Objektmanager einen Zeiger auf das Generische Objekt des gewünschten Objektes anhand der OID des Objektes an. Der OM muß also eine Methode

```
GO *getObject(OID oid);
```

bereitstellen, die das zu einer OID gehörende Objekt ermittelt und gegebenenfalls anfordert.

In dieser Methode wird zuerst gesucht, ob das Objekt lokal vorhanden ist. Ist dies nicht der Fall, so wird versucht, das zugehörige Top-Level-Objekt anzufordern. Dazu wird erst einmal untersucht, ob die zu diesem TLO gehörende Datei auf dem Sekundärspeicher existiert. Existiert diese nicht, so gibt es das Objekt nicht. Existiert die Datei, so wird dort nachgeschaut, auf welchen Knoten dieses Objekt lokal vorhanden ist. Existieren solche Knoten, so wird bei einem dieser Knoten das Objekt angefordert, indem er mit der Erstellung des Objektes beauftragt wird (siehe Abschnitt 6.6.1). Existiert das Objekt auf noch keinem Knoten, so wird die Objektkodierung aus der Datei gelesen, das Objekt erstellt und auf Sekundärspeicher vermerkt, daß das Objekt jetzt auf dem anfordernden Knoten vorhanden ist.

Bei diesem Algorithmus muß beachtet werden, daß er auch während einer Methodenverteilung aktiviert werden kann (Anforderung eines Objektes während einer Methodenverteilung). In diesem Falle läuft der eigentliche Algorithmus nur auf dem Initiator der Methodenverteilung ab, auf den Remote-Knoten wird nur auf ein Ergebnis vom Initiator gewartet. Der Initiator der Methodenverteilung veranlaßt die Verteilung des angeforderten Objektes auf die Remote-Knoten (sofern nötig). Das Ergebnis der Operation (Anforderung des Objektes und Verteilung auf Remote-Knoten) wird dann auf die Remote-Knoten verteilt. Bei negativem Ergebnis wird nur ein NULL-Pointer zurückgegeben, bei positivem Ergebnis wird der gewünschte Zeiger auf das GO zurückgegeben. In Anhang D.2 wird der Algorithmus ausführlich aufgeführt.

6.7 Neue SWS-Objekte

Neuerstellte SWS-Objekte müssen in den SWS eingefügt werden. Dies ist Aufgabe des Objektmanagers, der dazu eine Methode

```
bool newObject(TLI &tli);
```

zur Verfügung stellt. Diese Methode wird von den für die Erstellung neuer Objekte zuständigen, vom Programmgenerator erzeugten Methoden der SWS-Objekte aufgerufen. Sie trägt die Verwaltungsinformationen in die Top-Level-Information ein (TLOID; Knoten, auf denen Replikate existieren; Lockinformationen (bis jetzt noch kein Lock); ...) und hängt die TLI in die Verwaltungsstruktur des OM ein. Außerdem wird das Objekt auf Sekundärspeicher abgelegt (siehe Abschnitt über Objektpersistenz).

Dieses Vorgehen funktioniert allerdings nur, wenn die neuen SWS-Objekte nicht während einer Methodenverteilung erstellt werden. Wird ein neues SWS-Objekt während einer Methodenverteilung erstellt, muß zweierlei sichergestellt werden:

- Die Erzeugung des Objektes muß auf allen Knoten, auf denen die verteilte Methode läuft, entweder erfolgreich sein oder fehlschlagen.
- Die Verwaltungsinformationen (insbesondere die TLOID) müssen übereinstimmen — es darf nicht auf jedem Remote-Knoten ein anderes SWS-Objekt erzeugt werden.

Der zweite Punkt ist relativ einfach sicherzustellen, die einzutragenden Verwaltungsinformationen könnten vom Initiator der Methodenverteilung an die Remote-Knoten verteilt werden. Der erste Punkt ist ein Problem. Die verteilte Methode wird auf den Remote-Knoten erst einmal in die senderspezifische Warteschlange gestellt. Soll sichergestellt werden, daß das neu zu erzeugende Objekt auch überall erzeugt werden konnte, müßte gewartet werden, bis die Methode auf allen Remote-Knoten zur Ausführung kommt und das Objekt erstellt. Dies kann beliebig lange dauern und würde die Applikation auf dem Initator unangenehm verzögern. Es wird deshalb ein anderer Weg eingeschlagen. Die vom Programmgenerator erzeugten Methoden zur Neuerstellung eines Objektes erstellen dieses während einer Methodenverteilung nur auf dem Initiator, auf dem auch die Methode newObject(...) des OM aufgerufen wird. Im Falle einer Methodenverteilung versendet newObject(...) einen Objekterstellungsauftrag an die Remote-Knoten. Dieser enthält nur die notwendigen Verwaltungsinformationen. Wurde das Objekt auf allen Remote-Knoten erfolgreich erstellt, wird das Objekt vom Initiator auf Sekundärspeicher gespeichert. Kann das Objekt auf einzelnen Knoten nicht erstellt werden, so wird das Objekt wieder gelöscht. Die Erstellung des Objektes hat dann nicht funktioniert und wird nicht wiederholt. War die Erstellung erfolgreich, muß dies den Remote-Knoten explizit unter der Angabe der TLOID mitgeteilt werden, war sie nicht erfolgreich, muß dies auch gemeldet werden. Auf den Remote-Knoten landet diese Meldung in der absenderspezifischen Warteschlange.

Auf den Remote-Knoten laufen die vom Programmgenerator erzeugten Methoden zur Objekterstellung anders ab. Sie rufen die Methode

`GO *getNewObject()`

des Objektmanagers auf. Diese entnimmt der Warteschlange die Meldung des Initiators (bzw. wartet auf deren Ankunft). War die Objekterstellung erfolgreich, so wird anhand der in der Meldung des Initiators enthaltenen TLOID das Objekt ermittelt und ein Zeiger darauf zurückgegeben. Ansonsten wird NULL zurückgegeben.

6.8 Löschen von Objekten

Zum Löschen von Objekten wird durch den Objektmanager die Methode

`destroyObject(TLI *tli);`

zur Verfügung gestellt. Sie setzt voraus, daß der aufrufende Knoten einen Write-Lock für das zu löschende Objekt besitzt. Auch bei dieser Methode wird bei Methodenverteilung zwischen Initiator und Remote-Knoten unterschieden. Auf den Remote-Knoten wird abgewartet, bis der Initiator das Objekt gelöscht hat. Auf dem Initiator wird zuerst ein D-Lock auf das Objekt gesetzt. Danach wird das Objekt aus der Objektverwaltungsstruktur des OM entfernt und das Objekt mit Ausnahme der TLI gelöscht. Allen Knoten, auf denen das Objekt existiert, wird das Löschen des Objektes bekanntgegeben. Zuletzt wird das Objekt auf Sekundärspeicher gelöscht und eventuellen Remote-Knoten bekanntgegeben, daß das Objekt jetzt gelöscht ist. Die TLI des gelöschten Objektes wird mittels eines Flags als ungültig gekennzeichnet. Eventuell noch auf die TLI verweisende Referenzen bleiben dadurch definiert.

Diese Methode wird nur von der Methode `bool destroyObject();` des Generischen Objektes aufgerufen, welche für das Löschen eines Objektes durch die Anwendung zuständig ist. Diese Methode gestattet nur das Löschen von Top-Level-Objekten, Unterobjekte können nicht gelöscht werden.

6.9 Beenden einer Applikation

Bei Beendigung einer Applikation muß der Objektmanager den SWS geordnet verlassen:

- Alle von dieser Applikation noch gehaltenen Locks auf Objekte werden freigegeben.
- Die lokal gehaltenen Objektreplikate werden gespeichert. Falls dies nicht möglich ist (anderer Knoten hat Write-Lock) muß sichergestellt werden, daß noch ein Knoten existiert, der das Objekt sichern kann. Ist keiner mehr vorhanden, wird das Objekt trotzdem gesichert.
- Lösche für alle auf dem Knoten gehaltenen Objekte auf dem Sekundärspeicher die Information, daß die Objekte hier lokal vorhanden sind.
- Benachrichtige alle Replikate der auf diesem Knoten vorhandenen Objekte, daß die Objekte hier nicht mehr lokal vorhanden sind. Dazu wird auf jedes Objekt ein D-Lock gesetzt.

Nach dem Speichern der Objekte werden keine Aufträge mehr von anderen Knoten angenommen, die Bearbeitung der Warteschlangen wird eingestellt.

6.10 Pointer-Elementarobjekte und die Objektverwaltung

Bisher war nur allgemein von der Anforderung existierender SWS-Objekte durch die Applikation die Rede. Vom Programmgenerator werden dafür zu jeder Klasse Methoden generiert, die die Anforderung eines Objektes entweder anhand eines zuvor vergebenen Namens bzw. anhand der Objekt-Id durch den Applikationsprogrammierer erlauben. Diese Methoden fordern immer maximal ein Objekt aus dem SWS an — Objekte, die mittels im angeforderten Objekt enthaltenen Elementarpointern referenziert werden, werden dabei nicht automatisch angefordert. Auf einem Knoten vorhandene Elementarpointer können also global auf ein Objekt zeigen, das lokal auf diesem Knoten nicht vorhanden ist. Bei lesenden Zugriffen auf einen Elementarpointer wird deshalb das referenzierte Objekt automatisch angefordert, falls es bisher nicht lokal vorhanden ist. Bei lesendem Zugriff auf Elementarpointer während einer Methodenverteilung muß garantiert werden, daß die Operationen auf allen Knoten dasselbe Ergebnis liefern, wobei auch hier wieder der Initiator den Hauptteil der Arbeit erledigt. Er versucht sicherzustellen, daß das durch den Elementarpointer referenzierte Objekt auf allen Remote-Knoten lokal vorhanden ist. Nur wenn dies der Fall ist, darf die lesende Operation erfolgreich sein. Kann nicht sichergestellt werden, daß das referenzierte Objekt auf den Remote-Knoten vorhanden ist, so schlägt die lesende Operation auf den Elementarpointer fehl. Hierbei wird immer eine Exception ausgelöst, da auf dem Ergebnis der lesenden Operation eventuell sofort eine Methode ausgeführt wird.

Pointer-Operationen auf Elementarpointern sind aus diesem Grund sehr komplex und deshalb relativ langsam und sollten sparsam verwendet werden.

Kapitel 7

Der Programmgenerator

Der Applikationsprogrammierer entwirft SWS-Klassen in leicht modifiziertem C++. Der genaue Aufbau einer SWS-Klassendefinition wird im nächsten Abschnitt definiert. Der Programmgenerator hat die Aufgabe, aus dieser Definition reines C++ zu erzeugen. Dabei werden die für die Verteilung (und die Sperrmechanismen) notwendigen Erweiterungen vorgenommen.

7.1 Aufbau einer SWS-Klasse

Bei SWS-Klassen wird, wie es auch in C++ üblich ist , zwischen einem Header-File, das die Klassendefinition enthält, und einem Implementationsfile, in dem die Methoden implementiert werden, unterschieden. Der Aufbau dieser beiden Files wird im folgenden beschrieben.

7.1.1 Die Klassendefinition

Die Klassendefinition wird in einem speziellen Header-File abgelegt. Der Filename wird dabei aus dem Namen der zu definierenden SWS-Klasse und der Endung „.scd“ (= SWS Class Definition) zusammengesetzt. Die Form dieser Header-Files ist im Vergleich zu C++ stark eingeschränkt. Sie enthalten normale *#include*-Anweisungen des C++-Preprocessors, spezielle *#SWSinclude*-Anweisungen des SWS-Programmgenerators, Vorausdeklarationen von SWS-Klassen und die eigentliche Definition der SWS-Klasse (in dieser Reihenfolge). Die komplette Syntax ist in Anhang E.1 zu finden. Ein Beispiel zeigt Abbildung 7.1. Die normalen *#include*-Anweisungen des Preprocessors werden für die beliebigen Rückgabetypen der SWS-Methoden benötigt, sie werden durch den Programmgenerator nicht weiter ausgewertet. Mit den *#SWSinclude*-Anweisungen müssen in der SWS-Klassendefinition verwendete SWS-Klassen eingebunden werden. Vorausdeklarationen von SWS-Klassen erlauben Deklarationen von Klassen, die sich aufeinander beziehen. Dabei wird ein Bezeichner als SWS-Klassenname eingeführt, die SWS-Klasse jedoch noch nicht definiert.

```
#include<aheader.h>
#include"anotherheader.h"

#SWSinclude"C1.scd"

SWSclass C2;

SWSclass C3: virtual public C1, protected C2 RC(RI_DATA,RG_LATE){...};
```

Abbildung 7.1: Aufbau des SWS-Headerfiles C3.scd

Die Definition der SWS-Klasse ist der eigentliche Zweck des Header-Files. Die Syntax der SWS-Klassendefinition ist stark an der Klassendefinition von C++ orientiert, genaugenommen ist es nur eine Erweiterung. Sie setzt sich aus dem Klassenkopf, der Definition der Klassen-Id und einer Liste der Members (Daten und Methoden) zusammen:

class-definition:

class-head { *class-id-spec member-list* }***;***

Der Klassenkopf

Der Klassenkopf besteht aus dem Schlüsselwort **SWSclass**, gefolgt vom Klassennamen, einer Liste von Basisklassen und der Angabe einer für alle verteilbaren Methoden der Klasse gültigen Replikationscharakteristik:

class-head:

SWSclass *identifier* $\textit{base-spec}_{opt}$ $\textit{rc-spec-dist}_{opt}$

Der Klassenname kann ein beliebiger C++-Bezeichner sein. Die Liste der Basisklassen hat den gleichen Aufbau wie bei einer Klassendefinition in C++. Die Basisklassen müssen SWS-Klassen sein, die (direkt oder indirekt) mittels *#SWSinclude* eingebunden wurden.

base-spec:

: *base-list*

base-list:

base-specifier

base-list **,** *base-specifier*

base-specifier:

$\textbf{\textit{virtual}}_{opt}$ $\textit{access-specifier}_{opt}$ *SWS-class-name*

acess-specifier:

public

protected

private

Die Replikationscharakteristik wird in der Form **RC(Replikationsinformation, Replikationsgranularität)** angegeben, sie ist durch mindestens ein Leerzeichen von der Liste der Basisklassen getrennt. Für die Replikationsinformation *ri* können die Schlüsselworte **RI_DATA, RI_METHOD oder RI_DEFAULT** angegeben werden, für die Replikationsgranularität *rg* kann **RG_EARLY, RG_LATE oder RG_DEFAULT** angegeben werden.

rc-spec-dist:

  RC(*ri , rg*)

Die Angabe der Liste der Basisklassen ist optional, ebenso die Angabe der Replikationscharakteristik. Bei Nichtangabe einer Replikationscharakteristik wird automatisch RC(RI_DEFAULT,RG_DEFAULT) angenommen.

Ein Beispiel des Klassenkopfes der SWS-Klasse *C3* zeigt Abbildung 7.1. *C3* erbt die beiden SWS-Klassen *C1* (public virtual) und *C2* (protected). Für die verteilbaren Methoden der Klasse wird als gewünschte Replikationscharakteristik RC(Data, Late) angegeben.

Die Definition der Klassen-Id

Bevor die Member der Klasse deklariert werden, wird zuerst die Klassen-Id mittels dem Schlüsselwort *SWSClassId* definiert:

class-id-spec:

  SWSClassId* "*uuid-string*"*;

Diese Klassen-Id ist eine UUID des DCE, eine UUID in String-Form erhält man zum Beispiel durch Ausführung des Kommando's **uuidgen**. Jede Klasse muß eine eigene, eindeutige Klassen-Id besitzen, bei Änderung einer Klasse im SWS sollte diese Id geändert werden.

Die Memberliste

Die Liste der Members einer SWS-Klasse ist ähnlich aufgebaut wie bei einer Klassendefinition in C++:

member-list:

  access-specifier **:** *member-list*
  member-declaration $member\text{-}list_{opt}$
  member-declaration

member-declaration:

  data-member
  function-member

Die Verwendung der *access specifiers* ist äquivalent zur Verwendung in C++.

Data Members

Die Daten eines SWS-Objektes sind auf Elementarobjekte und SWS-Objekte beschränkt. Storage-Class-Specifier (*auto, register, static, extern*), Type-Specifier wie *const* und *volatile* und Pointer-Operatoren (&,*) sind nicht zugelassen.

data-member:
: *SWS-class-name identifier* ***;***
: *elementary-object-class identifier* ***;***

elementary-object-class:
: ***SWS_int8***
: ***SWS_int16***
: ...

Function members

Bei den Methoden, den *function members*, werden die schon besprochenen drei Typen unterschieden. Da sind auf der einen Seite die „sehenden“ („peeping“) Methoden, die ohne jede Prüfung von Sperren auf die Daten des Objektes lesend zugreifen. Auf der anderen Seite sind die „lesenden“ Methoden, zu denen auch die ändernden Methoden gehören. Diese werden weiter in verteilbare und nicht verteilbare Methoden unterteilt. Diese drei Methodentypen unterscheiden sich lediglich durch die Replikationscharakteristika und die Definition der benötigten Sperren.

function-member:
: ***virtual*** *nonvirtual-function-member pure-spec*$_{opt}$ ***;***
: *nonvirtual-function-member* ***;***

nonvirtual-function-member:
: *type*$_{opt}$ *identifier* ***(*** *param-list*$_{opt}$ ***)*** *rc-spec*$_{opt}$ *lock-spec*$_{opt}$
: *type*$_{opt}$ ***operator*** *operator* ***(*** *parameter*$_{opt}$ ***)*** *rc-spec*$_{opt}$ *lock-spec*$_{opt}$
: ***operator*** *type* ***()*** *rc-spec*$_{opt}$ *lock-spec*$_{opt}$

parameter:
: *param-typ identifier param-lock-spec*$_{opt}$

param-typ:
: *elementary-type ptr-op*$_{opt}$
: *SWS-class-name ptr-op*
: *elementary-object-class ptr-op*

param-lock-spec:
: ***L_READ***
: ***L_WRITE***

Die komplette Syntax für *function-member* findet man in Anhang E.1. Es gibt hier mehrere Einschränkungen gegenüber C++:

- Die Definition von Konstruktoren und Destruktoren ist nicht gestattet,
- Der Typ des Rückgabewertes ist zwar prinzipiell frei wählbar, er darf jedoch nur einen Bezeichner und eventuell einen Pointer-Operator umfassen. Sollen komplexere Typen zurückgegeben werden, so können diese in einem Header-File, welches mittels *#include* eingebunden wird, definiert werden. Zu beachten ist die Einschränkung, daß SWS-Objekte nur als Referenz oder Zeiger zurückgegeben werden dürfen.
- Die Parameter beschränken sich auf die Elementartypen des Basissystems (*ETY_...*) und auf SWS-Objekte (einschließlich der Elementarobjekte des Basissystems (*SWS_...*)). Bei den Elementartypen sind sowohl Werteparameter als auch Zeiger und Referenzen erlaubt, SWS-Parameter können nur als Referenz bzw. als Zeiger übergeben werden.
- Die Operatoren *new, delete*, "`,`" , "`->`" und "`->*`" können nicht überladen werden, bei `&` *und* `*` ist das Überladen der unären Operatoren(Zeiger-Operationen) verboten.
- Der Typ der benutzerdefinierten Umwandlungen hat die gleichen Einschränkungen wie der Typ des Rückgabewertes, d.h. er darf auch nur aus einem einzelnen Identifier und eventuell einem Pointer-Operator bestehen.
- Der Funktionsrumpf kann nicht in der Klassendefinition angegeben werden.
- Inline-Funktionen sind nicht vorgesehen (alle Methoden außer den „sehenden" Methoden werden durch die Bearbeitung durch den Programmgenerator sowieso zu groß, als daß sie sich als Inline-Funktionen eignen würden.)

„Sehende" Methoden greifen ohne jegliche Prüfung von Sperren lesend auf das Objekt zu. Es werden deshalb bei den Parametern keine Locks angegeben, die Replikationscharakteristik fällt ebenfalls weg. Gekennzeichnet werden diese Methoden durch den Lock-Specifier *L_PEEPING* am Ende.

Nicht verteilbare Methoden werden durch die Replikationscharakterik RC(NotDistributed) gekennzeichnet. Bei diesen Methoden können an zwei Stellen Locks angegeben werden: Bei den Parametern (nur bei SWS-Objekten und Elementarobjekten, nicht bei Elementartypen) und ganz am Schluß. Die bei den Parametern angegebenen Locks spezifizieren den Lock, den die Parameter beim Aufruf dieser Methode benötigen. Der ganz am Schluß angegebene Lock ist der Lock, den das Objekt benötigt, auf das die Methode aufgerufen wird. Die Angabe der Locks ist optional. Wird kein Lock angegeben, so wird als Default-Wert *L_READ* angenommen.

Verteilbare Methoden werden durch die Replikationscharakteristik RC(ri,rg) gekennzeichnet. Für die Locks gilt das bei den nicht verteilbaren Methoden ausgeführte. Im Gegensatz

zu den nicht verteilbaren Methoden wird hier jedoch für das Objekt, auf dem die Methode aufgerufen wird, als Lock ein Default-Wert von *L_WRITE* angenommen.

Die Angabe der Replikationscharakteristik ist optional. Wird sie bei einer Methode nicht angegeben, ist diese eine verteilbare Methode, die Replikationscharakteristik ist RC(RI_DEFAULT, RG_DEFAULT).

Sowohl bei den nicht verteilbaren als auch den verteilbaren Methoden ist bei Verwendung virtueller Methoden Vorsicht geboten. Es ist nicht unbedingt notwendig, daß die Replikationscharakteristik einer als virtuell deklarierten Methode bei der Definition dieser Methode in einer erbenden Klasse beibehalten wird. Unbedingt notwendig ist jedoch, daß die notwendigen Locks in allen Definitionen gleich bleiben, da sonst die benötigten Locks für einen Methodenaufruf auf einen Pointer eines Typs unterschiedlich sein könnten, je nachdem, auf was dieser Pointer zeigt!! Bei nicht definierten Locks werden bei virtuellen Methoden die der virtuellen Methoden der Basisklassen verwendet.

Abbildung 7.2 deutet die Definition zweier SWS-Klassen *A* und *B* an. In *A* wird die virtuelle Methode *m1* definiert, welche, da keine RC angegeben ist, per Default eine verteilbare Methode mit der RC(default, default) ist. Die Methode bekommt als Parameter einen SWS_int32, eine Referenz auf ein SWS-Objekt der Klasse *D* und einen ETY_int8 übergeben. Der SWS_int32 benötigt beim Aufruf einen Write-Lock, das Objekt der Klasse *D* einen Read-Lock. Der ETY_int8 ist kein SWS-Objekt und benötigt keinen Lock. In Klasse *B* wird diese Methode *m1* nochmals mit derselben Signatur definiert. Im Gegensatz zur Klasse *A* ist diese Methode eine nicht verteilbare Methode. Da beim SWS_int32 hier kein Lock definiert ist, würde per Default ein Read-Lock definiert. Da *m1* jedoch virtuell ist, wird der in *A* angegebene Write-Lock übernommen. Die Definition des zweiten Parameters von *m1* in *B* ist falsch, da in *B* schon ein Read-Lock für diesen Parameter spezifiziert wurde. Der Lock, der bei Aufruf der Methode für das Objekt, auf dem die Methode aufgerufen

```
SWSclass A {
public:
virtual void m1(SWS_int32 p1 L_WRITE, D &p2 L_READ, ETY_int8 p3);
...
};

SWSclass B : public A RC(RI_METHOD, RG_EARLY){
public:
void m1(SWS_int32 p1, D &p2 L_Write, ETY_int8 p3) RC(NotDistributed);
int m2(SWS_Pointer<A> *p4, ETY_int16 &p5) L_PEEPING;
...
};
```

Abbildung 7.2: Klassendefinitionen der Klasse *A* und *B* (in getrennten Headerfiles)

wird, nötig ist, wird von der Definition in A übernommen. Die Methode *m2* von B ist eine „sehende“ Methode, hier werden überhaupt keine Locks spezifiziert. Der erste Parameter der Methode ist (ein Pointer auf) ein Pointer-Elementarobjekt, das auf ein Objekt der (SWS-)Klasse A zeigt.

7.1.2 Die Implementation der Methoden einer Klasse

Im Implementationsfile werden die Methoden einer SWS-Klasse implementiert. Der Filename setzt sich zusammen aus dem Namen der zu implementierenden SWS-Klasse und der Endung „.sci“ (= SWS Class Implementation). Dieses File besteht nur aus *#include*-Anweisungen und den Methodendefinitionen. Die Methodendefinitionen haben folgende Syntax:

function-definition:

type$_{opt}$ *classname*::*identifier* (*def-param-list*$_{opt}$) { *fct-body* }
type$_{opt}$ *classname*::**operator** *operator* (*def-parameter*$_{opt}$) { *fct-body* }
classname::**operator** *type* () { *fct-body* }

def-param-list:

def-param-list , *def-parameter*
def-parameter

def-parameter:

param-typ identifier

classname ist hierbei der Name der Klasse, deren Methoden in diesem File implementiert werden. Die Lock- und RC-Spezifikationen werden in den Funktionsköpfen nicht mehr angegeben. *fct-body* ist der Funktionsrumpf, der beliebigen C++-Code enthalten darf.

7.2 Die Replikationscharakteristik

Die Berechnung der Replikationscharakteristik wird in [Sem93] definiert. Dies wird hier beinahe ohne Änderung übernommen.

7.2.1 Die gewünschte RC eines Methodenaufrufes

Die Angaben der RC, die zur Berechnung der gewünschten RC eines Methodenaufrufes einer verteilbaren Methode verwendet werden, können an bis zu drei verschiedenen Stellen stehen:

- Im Kopf der Klasse, zu der die Methode gehört. Die dort angegebene RC gilt für alle Methoden der Klasse.

- Bei der Methodendefinition einer verteilbaren Methode (im Header-File). Die dort angegebene RC gilt für alle Aufrufe der Methode.
- Beim Aufruf der verteilbaren Methode. Die dort angegebene RC gilt für diesen Aufruf.

Die Priorität nimmt in der Liste nach unten hin zu, d.h. die Replikationsdirektiven beim Aufruf überschreiben beispielsweise die bei der Methodendeklaration angegebenen Direktiven. Nicht angegebene RC's werden durch Default-Werte ersetzt.

Bisher wurde die Angabe der RC beim Methodenaufruf noch nicht erwähnt. In welcher Form wird sie dort angegeben? Eine Form wie in der Klassendefinition ist nicht möglich, da dazu das gesamte Programm geparst werden müßte. Da dieses vermieden werden soll, kann die für einen Methodenaufruf gewünschte RC nicht statisch berechnet werden, obwohl die Information vorläge. Die für einen Methodenaufruf gewünschte RC wird deshalb teilweise statisch, teilweise dynamisch berechnet. Der Teil der gewünschten RC, der sich aus der Angabe im Klassenkopf und der Angabe der RC bei der Methodendeklaration zusammensetzt, kann statisch berechnet werden.

Die Angabe der RC beim Methodenaufruf wird durch Parameter durchgeführt. Der Generator erweitert die Parameterlisten der verteilbaren Methoden um zwei weitere optionale Parameter:

`m1(...)` → `m1(..., ri = RI_DEFAULT, rg = RG_DEFAULT)`

Aus der statisch berechneten RC und dieser beim Aufruf übergebenen RC wird dann dynamisch die gewünschte RC für diesen Aufruf berechnet. Die Berechnung der für einen Methodenaufruf gewünschten RC wird durch die Relation '→' in Tabelle 7.1 definiert. Die Berechnung soll anhand der Abbildung 7.3 erläutert werden. Der statische Teil der Berechnung der gewünschten RC des Aufrufes der Methode *m1* ist für alle Aufrufe derselbe und wird durch den Programmgenerator vollzogen:

$$\mathtt{RI}_{m1,stat} = \mathtt{ri_0} \rightarrow \mathtt{ri_1}$$
$$\mathtt{RG}_{m1,stat} = \mathtt{rg_0} \rightarrow \mathtt{rg_1}$$

Der dynamische Teil der Berechnung der gewünschten RC des Aufrufes der Methode *m1* wird beim Aufruf der Methode durch vom Programmgenerator erzeugten Code gemacht:

$$\mathtt{RI}_{m1,dyn} = \mathtt{RI}_{m1,stat} \rightarrow \mathtt{ri_2}$$
$$\mathtt{RG}_{m1,dyn} = \mathtt{RG}_{m1,stat} \rightarrow \mathtt{rg_2}$$

→	default	Method	Data
default	default	Method	Data
Method	Method	Method	Data
Data	Data	Method	Data

→	default	late	early
default	default	late	early
late	late	late	early
early	early	late	early

Tabelle 7.1: Relation '→' für RI (links) und RG (rechts)

```
SWSclass A RC(ri_0, rg_0){
public:
void m1(...) RC(ri_1, rg_1);
...
};

...
A *a = A::newObject();
a->m1(...,ri_2, rg_2);
```

Abbildung 7.3: Klassendefinitionen der Klasse *A* mit Beispielaufruf der Methode *m1*.

7.2.2 Die tatsächliche RC eines Methodenaufrufs

Um die tatsächliche RC eines Methodenaufrufes zu berechnen, muß noch die „Vorgeschichte“ dieses Aufrufes beachtet werden, da eine Methode oft vor vollendete Tatsachen gestellt wird. Wurde schon die aufrufende Methode methodenverteilt, dann ist die gewünschte RC nicht mehr relevant.

Die Berechnung der tatsächlichen RC eines Methodenaufruf errechnet sich aus der bisher angesammelten RC und der für den Aufruf gewünschten RC. Die Berechnungsvorschrift wird durch die Relation '⇒' in Tabelle 7.2 definiert. Links steht hierbei die angesammelte RC, rechts die gewünschte. Die Berechnung der tatsächlichen RC für das Beispiel aus dem letzten Ausschnitt sieht dann so aus:

$$\mathtt{RI}_{m1,tats} = \mathtt{RI}_{m0,tats} \Rightarrow \mathtt{RI}_{m1,dyn}$$
$$\mathtt{RG}_{m1,tats} = \mathtt{RG}_{m0,tats} \Rightarrow \mathtt{RG}_{m1,dyn}$$

$\mathtt{RC}_{m0,tats}$ ist die tatsächliche Replikationscharakteristik der aufrufenden Methode.

Die RC *default* kann sich nicht ansammeln, da bei jedem Aufruf einer verteilbaren Methode zumindest entschieden werden muß, ob jetzt oder später verteilt werden soll. Für den „äußersten“ Aufruf einer verteilbaren Methode (d.h. diese Methode wurde weder direkt noch indirekt von einer verteilbaren Methode aufgerufen) ist die tatsächliche RC gleich der gewünschten RC. Enthält diese gewünschte RC ein *default* , so wird für dieses *default*

⇒	default	Method	Data
Method	Method	Method	Data
Data	Data	Data	Data

⇒	default	late	early
late	late	late	early
early	local	local	local
local	local	local	local
remote	remote	remote	remote

Tabelle 7.2: Relation '⇒' für RI (links) und RG (rechts)

ein Standardwert eingesetzt. Dieser Standardwert ist als RC(Method,early) vordefiniert und kann durch die Funktion **bool setRC_Default(ri, rg)** geändert werden (dabei darf kein *default* gesetzt werden).

Die Replikationsgranularität muß für die tatsächliche RC um zumindest einen Zustand erweitert werden, der angibt, daß die Verteilung schon in übergeordneten Methoden geregelt wurde. Dafür wurde der Zustand *local* eingeführt. Bei einigen Aktionen ist es wichtig zu wissen, ob eine Methode verteilt abläuft und ob die momentan ausgeführte Methode auf dem Initiator der Methodenverteilung oder auf einem Remote-Knoten abläuft. Es wird daher noch ein weiterer Zustand *remote* eingeführt. Die Methoden auf den Remote-Knoten haben immer die Replikationsgranularität *remote*. Weiterhin wird ab dem Zeitpunkt der Verteilung in untergeordneten Methodenaufrufen die tatsächliche Replikationsinformation der aufgerufenen Methoden immer auf den Wert der aufrufenden Methode gesetzt (damit die Information nicht verloren geht, ob Daten- oder Methodenverteilung durchgeführt wird).

Wie wird die tatsächliche RC durch die Aufrufe gereicht? Ein Durchreichen mittels Parameter scheidet aus, da dies beim Aufrufen von beliebigen, nicht zum SWS gehörigen Funktionen von SWS-Methoden aus Probleme gibt. Diese Funktionen könnten wieder SWS-Methoden aufrufen, müßten also auch die RC durchreichen können. Dies ist i.a. nicht möglich. Aus diesem Grund wird die RC in einem globalen Objekt gehalten, auf das von überall zugegriffen werden kann. Das Problem hierbei ist, daß auf diese globale RC nicht nur die Applikation, sondern auch die von anderen Knoten verteilten Methoden, die in einem eigenen Thread laufen, zugreifen. Dieses Objekt ist daher so implementiert, daß der Inhalt sozusagen threadlokal ist, d.h. das Objekt hat für jeden Thread eine RC, auf die nur dieser Thread zugreift. Auf die globale RC darf von der Applikation direkt nicht zugegriffen werden. Dies wird durch geeignete Mechanismen verhindert.

7.3 Die Arbeitsweise des Generators

Ein Generatorlauf zur Erzeugung einer SWS-Klasse ist in zwei Abschnitte unterteilt. Im ersten Abschnitt wird das Header-File der SWS-Klasse bearbeitet. Dabei werden (rekursiv) alle durch *#SWSinclude* eingebundenen SWS-Klassendefinitionen geparst und die C++-Headerdatei für dieses SWS-Objekt erstellt. Im zweiten Abschnitt wird das Implementationsfile geparst, die darin enthaltenen Methoden bearbeitet und zusätzliche Methoden erstellt.

7.3.1 Die Bearbeitung des Header-Files

Parsen der Header-Files

Der erste Schritt des Generators ist das Parsen des Header-Files der zu bearbeitenden SWS-Klasse und mittels *#SWSinclude* eingebundener Header-Files anderer SWS-Klassen.

Der implementierte Parser ist ein LALR-Parser, der unter Zuhilfenahme der an die **GNU**-Tools **bison/flex** angelehnten Tools **bison++/flex++** erstellt wurde. Die dabei verwendete Grammatik ist eine leicht modifizierte Version der im Anhang E.1 aufgeführten Grammatik.

#SWSinclude-Anweisungen werden sofort behandelt, erst nach dem Parsen der (rekursiv) eingebundenen Header-Files wird an der Stelle nach der *#SWSinclude*-Anweisung weitergemacht. Bei diesem Prozess wird darauf geachtet, daß ein schon eingebundenes Header-File kein zweites Mal gelesen wird (Zyklen!), wobei ein Header-File als eingebunden gilt, sobald seine Bearbeitung startet.

Die in diesen Header-Dateien definierten SWS-Klassen werden in einer Klassentabelle eingetragen. Ein solcher Eintrag enthält die gesamte in der Klassendefinition verfügbare Information, d.h. den Klassennamen, die Klassen-Id, die Basisklassen und die Member der Klasse (sowohl Daten als auch Methoden (Signatur,...)). Die Information über die Basisklassen enthält Verweise auf die Einträge der Basisklassen in der Klassentabelle. Zu jeder Klasse werden noch zusätzliche Informationen der in der Klasse und den Basisklassen definierten virtuellen Funktionen gehalten. Anhand dieser Informationen kann die notwendige Sonderbehandlung der Lock-Spezifikationen der virtuellen Methoden während des Parsens erledigt werden.

Das Header-File der zu generierenden Klasse wird während des Vorgangs des Parsens der SWS-Klassendefinition erstellt. Der Filename setzt sich aus dem Namen der zu generierenden Klasse und der in C++ üblichen Endung „**.h**" zusammen. **#include**-Anweisungen werden dabei übernommen, **#SWSinclude"**name**.scd"** werden in **#include"**name**.h"** umgewandelt. Die Definition der Klassen-Id wird als Kommentar übernommen. Die Klassendefinition wird auch beinahe komplett übernommen. Lediglich die Lock- und RC-Direktiven werden entfernt und am Schluß die Definitionen der zusätzlichen Data und Function Members angehängt. Eine Aufstellung dieser Ergänzungen ist in Anhang F zu finden.

Das „Modell" des Objektes

Nach erfolgtem Parsen wird anhand der Klassentabelle ein „Modell" eines Objektes der zu generierenden Klasse erstellt. Dieses Modell stellt den Vererbungsgraph der Klasse dar. Der Graph ist ein gerichteter azyklischer Graph, dessen Ecken die Klassen darstellen, die gerichteten Kanten (a,b) stellen die Vererbungsbeziehungen (a erbt b) dar. Für die zu generierende Klasse X darf keine Kante (y,X) existieren. Im Gegensatz zur Klassentabelle sind in diesem Graph mehrfach vorhandene Basisklassen auch tatsächlich mehrfach aufgeführt. Beim Aufbau dieses Modells wird eine Tabelle sämtlicher virtueller Basisklassen angelegt. Abbildung 7.4 zeigt die Definition der Klasse E, die daraus resultierende Klassentabelle und das Modell der Klasse. Die Knoten des Graphen besitzen Verweise in die Klassentabelle, die im Beispiel nicht dargestellt sind. Auch die beim Aufbau des Modells generierte Tabelle der virtuellen Basisklassen fehlt in der Abbildung. In dieser sind Zeiger auf die

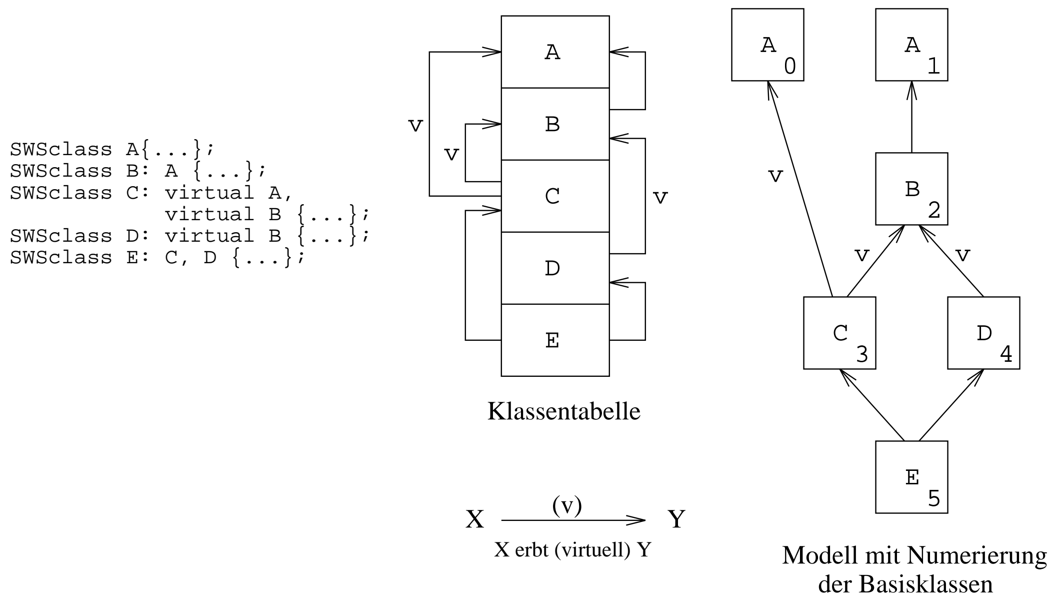


Abbildung 7.4: Definition der Klasse E mit resultierender Membertabelle und Modell

Knoten des Graphen abgelegt, welche virtuelle Basisklassen darstellen. Durch den Aufbau des Modells sind die virtuellen Basisklassen in der Reihenfolge ihrer Konstruktoraufrufe in dieser Tabelle abgelegt.

Der Aufbau des Modells ist relativ einfach. Die in der Klassentabelle abgelegten Klassenobjekte besitzen eine Methode **buildTree(VTab &vtab)**, welche als Parameter eine Referenz auf die Tabelle der virtuellen Basisklassen (VTab) übergeben bekommt und als Ergebnis einen Zeiger auf einen Knoten im Graph zurückgibt. Dieser Knoten repräsentiert ein Modell der Klasse, welche durch das Klassenobjekt definiert wird. Die Methode erstellt dazu zuerst einen Baumknoten mit Referenz auf die Klassentabelle. Dieser Knoten enthält genau so viele Zeiger auf andere Baumknoten, wie die durch das Klassenobjekt beschriebene Klasse direkte Basisklassen besitzt. Nun werden die direkten Basisklassen in der Reihenfolge ihrer Definition im Klassenkopf bearbeitet. Bei einer virtuellen Basisklasse wird zuerst nachgeschaut, ob diese in der Tabelle der virtuellen Basisklassen schon eingetragen ist. Wenn nein, wird die Methode *buildTree(...)* auf die Basisklasse aufgerufen. Der zurückgegebene Pointer wird sowohl in die Tabelle der virtuellen Basisklassen als auch in den entsprechenden Pointer des Graph-Knotens eingetragen. Ist die virtuelle Basisklasse schon in der Tabelle vorhanden, so wird der dort gespeicherte Pointer in den Knoten eingetragen. Bei nicht virtuellen Basisklassen wird die Methode *buildTree(...)* immer aufgerufen und der zurückgegebene Pointer im Knoten gespeichert. Nachdem alle Basisklassen abgearbeitet sind, wird ein Pointer auf den neu erstellten Graphknoten zurückgegeben. Das Modell des Objektes E im Beispiel erhält man durch den Aufruf von *buildTree(...)* auf das Klassenobjekt in der Klassentabelle, das die Klasse E beschreibt. Der Parameter ist dabei eine Referenz auf eine leere Tabelle. Die Methode gibt einen Pointer auf das Modell zurück, die Tabelle enthält dann die virtuellen Basisklassen.

Ermittlung der Base Class Id's

Auf diesem Model werden nun einige Algorithmen durchgeführt, z.B. ob das Modell den Anforderungen entspricht (siehe Einschränkung der möglichen Vererbungsstrukturen in 3.2.4). Diese Algorithmen basieren größtenteils auf einer Art „Tiefensuche“ im Graph, die virtuellen Basisklassen werden dabei teilweise gesondert behandelt. Ein Beispiel ist die Numerierung der Basisklassen. Diese Numerierung soll die Aufrufreihenfolge der Konstruktoren im Objekt widerspiegeln (die Aufrufreihenfolge der Konstruktoren eines Objektes wird in [ES92] definiert). Sie, die schon beschriebene Base Class Id, dient als eindeutige Numerierung der Basisklassen in einem Objekt. Die Knoten des Graphen enthalten eine Variable, in der diese Base Class Id abgelegt wird. Diese wird beim Bau des Baumes mit einem negativen Wert vorbelegt. Die Erzeugung der Nummern ist wieder relativ einfach. Die Objekte, welche die Graph-Knoten realisieren, besitzen eine Methode **numberTree(int &highestNumber)**. Diese Methode bekommt als Parameter eine Referenz auf einen Integer, in welchem die bis jetzt höchste vergebene Base Class Id gespeichert ist. Die Methode prüft zuerst, ob dieser Knoten schon numeriert wurde (nicht negative Nummer). Ist dies der Fall, ist die Methode beendet. Ist dies nicht der Fall, wird diese Methode auf alle im Knoten gespeicherten Zeiger aufgerufen. Dadurch werden zuerst alle Basisklassen dieses Knotens numeriert, *highestNumber* wird dabei durch die Basisklassen i.a. erhöht. Nach der Abarbeitung der Basisklassen wird die aktuelle *highestNumber* incrementiert und im Knoten gespeichert Der Aufrufmechanismus der Konstruktoren in C++ bedingt, daß diese Methoden zuerst auf die virtuellen Basisklassen in der Reihenfolge ihres Erscheinens in der VTab aufgerufen werden, bevor sie auf das gesamte Modell aufgerufen werden. Im obigen Beispiel wurde die Methode also zuerst auf das A mit der Nummer 0 aufgerufen, dann auf B und erst dann auf E.

7.3.2 Die Bearbeitung des Implementationsfiles

Nach der Bearbeitung des Header-Files wird das Implementationsfile bearbeitet. Dabei werden die Methoden um Code zur Behandlung der Verteilungs- und Lockaspekte erweitert und zusätzliche neue Methoden und Funktionen generiert.

Parsen des Implementationsfiles

Das Implementationsfile wird wie das Header-File durch einen LALR-Parser bearbeitet. Es wird dabei das Implementationsfile **classname.scd** gelesen, das Ergebnis wird im File **classname.C** abgelegt (*classname* ist der Name der zu generierenden SWS-Klasse). Eine vollständige Analyse der Syntaxstruktur war leider in der vorgegebenen Zeit nicht zu realisieren. Es werden daher bei der Bearbeitung des Implementationsfiles nur die Anfänge (und die Enden) der Methoden gesucht. Eine Erweiterung ist nur an diesen Stellen notwendig.

Die **#include**-Anweisungen am Anfang des Files werden werden einfach übernommen, **#include**-Anweisungen für das Header-File der zu generierenden Klasse und die vom Basissystem benötigten Header-Files werden dazugefügt.

Anschliessend werden die im Implementationsfile enthaltenen Methoden bearbeitet. Die dabei notwendigen Erweiterungen der Methoden werden in den nächsten zwei Abschnitten beschrieben.

Zum Schluß werden die in Anhang F aufgeführten Methoden generiert und die Klassen-Id und die Navigationsstruktur initialisiert.

Problem der Erweiterung der Methoden

Die Methoden von SWS-Objekten müssen zur Behandlung der Replikation und der Sperrmechanismen erweitert werden. Diese Erweiterungen sollen sich auf eine „Schale“ um die Methoden beschränken. Logisch gesehen schließt diese Schale die Methoden auf beiden Seiten ein — die Verteilung einer Methode wird zu Anfang der Methode abgehandelt, Daten können erst nach Beendigung der Methode verteilt werden. Dadurch ergeben sich zwei Schwierigkeiten:

- Eine Methode kann zu jedem beliebigen Punkt mittels *return* verlassen werden.
- Bei Auftreten einer Exception innerhalb einer Methode wird diese durch den „naheliegendsten“ Exception-Handler dieser Exception behandelt. Liegt dieser ausserhalb der Methode, wird der Rest der Methode nicht ausgeführt.

Die erste Schwierigkeit ließe sich durch Zwischenspeicherung des Rückgabewertes und einem Sprung zum „unteren“ Teil der Schale (der Teil der Schale am Ende der Methode) mittels *goto* lösen. Dies erfordert die genaue Analyse des Inhaltes der Methode und einen Umbau der Methode. Die zweite Schwierigkeit ließe sich durch Einsatz von Default-Exception-Handlern lösen. Die Schale würde durch diese Lösungen jedoch ziemlich kompliziert. Bei genauerer Untersuchung der notwendigen Ergänzungen der Methoden zeigte sich, daß der untere Teil der Schale unabhängig vom Inhalt der Methode ist. Es wird deshalb zu einem Trick gegriffen. Die unteren Teile der Schale werden in Destruktoren von extra dafür definierten Objekten realisiert. Diese Objekte werden lokal in den Methoden deklariert (siehe auch Anhang G). Beim Verlassen der Methode werden die Destruktoren dieser Objekte und damit auch der untere Teil der Schale ausgeführt. Diese Lösung funktioniert auch bei einer auftretenden Exception, da diese den Stack abräumt und dabei auch diese speziellen Objekte destruiert.

Erweiterung der Methoden

Bei den Methoden werden hier wieder die drei verschiedenen Typen — „sehend“, nicht verteilbar und verteilbar — unterschieden.

Bei den „sehenden“ Methoden muß sichergestellt werden, daß sie nicht von „lesenden“ Methoden (das sind alle Methoden im SWS, welche nicht „sehend“ sind) direkt oder indirekt aufgerufen werden. Auf der anderen Seite muß dafür gesorgt werden, daß die „sehenden“ Methoden weder direkt noch indirekt „lesende“ Methoden aufrufen. Es wird dazu ein **Zugriffsmodus („Access Mode“)** eingeführt. Dieser Modus gibt in einer globalen Variablen an, ob im Moment normal auf SWS-Objekte zugegriffen wird oder ob im Moment eine „sehende“ Methode abläuft. Am Anfang der „sehenden“ Methoden wird zuerst die globale RC geprüft. Diese darf nur Default-Werte enthalten. Ist dies nicht der Fall, dann wurde die „sehende“ Methode direkt oder indirekt von einer „lesenden“ Methode aufgerufen. In diesem Falle wird die Exception **AccessModeError** ausgelöst. Ansonsten wird der Access Mode geprüft. Ist dieser noch nicht auf “lesend“ gesetzt, wird dies gemacht. Mehr muß am Anfang der Methode nicht gemacht werden. Falls der Access Mode auf „sehend“ gesetzt wurde, muß dies am Ende der Methode rückgängig gemacht werden. Diese gesamte Behandlung wird in einem extra hierfür entworfenen Objekt durchgeführt. Im Konstruktor wird die RC geprüft und eventuell der Acess Mode auf „sehend“ gesetzt, im Destruktor wird der Access Mode eventuell wieder auf „normal“ gesetzt.

Bei nicht verteilbaren Methoden müssen zweierlei Dinge erledigt werden. Zuerst muß der Access Mode geprüft werden. Steht dieser auf „sehend“, so wird die Exception **AccessModeError** ausgelöst. Ist der Access Mode in Ordnung, wird getestet, ob die mit diesem Methodenaufruf zusammenhängenden Objekte (Parameter der Methode (nur bei SWS-Objekten) und das Objekt, auf das die Methode aufgerufen wurde) mit den in der Klassenspezifikation spezifizierten Locks kompatible Locks besitzen. Ist dies nicht der Fall, so wird versucht, die geforderten Locks anzufordern. Sollte das Anfordern nicht möglich sein, wird die Exception **LockError** ausgelöst. Das Freigeben der automatisch angeforderten Locks am Ende der Methode wird durch die beim Anfordern zurückgegebenen Lock-Handles sichergestellt.

Verteilbare Methoden müssen zusätzlich zu dem, was bei nicht verteilbaren Methoden gemacht werden muß, noch eine aktuelle RC berechnen und eine eventuell aus der aktuellen RC folgende Verteilung abhandeln. Die Berechnung der RC wird, wie auch die Prüfung des Access Mode bei „sehenden Methoden“, in einem speziell dafür entworfenen Objekt erledigt. Im Konstruktor wird die aktuelle RC berechnet, im Destruktor wird die RC auf den alten Wert zurückgesetzt. Die Behandlung der Verteilung wird etwas anders durchgeführt. Die Methodenverteilung, welche am Anfang behandelt wird, wird direkt in die Methode codiert. Der Abschluß der Methodenverteilung bzw. die Durchführung der Datenverteilung wird im Destruktor eines dafür entworfenen Objektes behandelt. Abschnitt 2.2.5 beschreibt die Notwendigkeit der Meldung der Änderungen an die View-Objekte bei Methoden- bzw. Datenverteilung. Bei Methodenverteilung werden diese Meldungen beim Initiator durch die verteilende Methode durchgeführt, auf den Remote-Knoten wird dies durch die Methode gemacht, die die verteilte Methode aufruft. Bei Datenverteilung werden die Meldungen durch den Objektmanager durchgeführt. Die genaue Spezifikation der Schnittstelle zu den View-Objekten steht noch aus. Deshalb werden in dieser Implementierung bei Methodenverteilung nur der die Verteilung veranlassende Knoten, die Objekt-Id

und die Kodierung der verteilten Methode an den Event-Manager weitergeleitet. Bei Datenverteilung wird der ändernde Knoten und die TLOID des geänderten Objektes an den Event-Manager gemeldet.

In Anhang G werden die "Schalen" um diese drei Typen von Methoden in Pseudcode dargestellt.

7.4 Aufbau der Navigationsstruktur

Die in Abschnitt 3.2.3 beschriebene Navigationsstruktur wird als statische Datenstruktur im generierten „*.C"-File abgelegt. Die Dimension der anzulegenden Tabelle ist bekannt (Base Class Id der zu generierenden Klasse plus eins), die Tabelle kann also in einem zweidimensionalen Feld abgelegt werden. Dieses zweidimensionale Feld wird zuerst komplett im Programmgenerator in einem Feld aufgebaut und danach als Initialisierungsliste eines zweidimensionalen Feldes abgelegt. Die Grundidee beim Aufbau der Struktur ist folgende: von einer bestimmten Basisklasse aus kann man deren direkte Basisklassen und alle von diesen direkten Basisklassen aus erreichbaren Basisklassen erreichen.

Der Aufbau geschieht durch die Methode *buildNavStruct(...)* der Baumknoten des Modells. Diese sorgt zuerst einmal durch Aufruf derselben Methode auf den Knoten der direkten Basisklassen dafür, daß die entsprechenden Zeilen der Navigationstabelle der Basisklassen ausgefüllt werden. Die eigene Zeile in der Navigationsstruktur kann dann durch einfaches „Mischen" der Zeilen der Basisklassen und das Eintragen der Basisklassen erfolgen. Knoten, zu denen keine Basisklassen existieren initialisieren ihre Zeilen mit Null.

Abbildung 7.5 zeigt den Aufbau der Navigationsstruktur anhand des Beispieles aus Abschnitt 7.3.1 Der Algorithmus ist schon so weit fortgeschritten, daß die Aufrufe der Methode *buildNavStruct(...)* auf die Knoten C und D vollendet ist. Knoten E füllt jetzt „seine" Zeile, die unterste, aus. Dazu untersucht er die Zeilen seiner direkten Basisklassen in der Reihenfolge ihrer Aufzählung im Klassenheader. Angefangen wird also mit der Zeile 3 (die Numerierung der Zeilen und Spalten beginnen bei 0), der Zeile der Klasse C. Der Inhalt dieser Zeile wird auf die Zeile 5 übertragen: Steht in Zeile 3 eine Null, wird in Zeile 5 nichts eingetragen. Steht in Zeile 3 eine Zahl, wird in Zeile 5 die „Nummer" der gerade untersuchten Basisklasse eingetragen. Die Nummer von C ist eins, sie ist im Klassenkopf von E als erste aufgezählt. Als Ergebnis erhält man den Zeilenvektor (1,1,1,-,-,-). Nun wird dasselbe mit der Zeile der zweiten Basisklasse von E, der Klasse D, wiederholt. Dabei werden teilweise die im ersten Schritt eingetragenen Nummern überschrieben. Das Ergebnis des zweiten Schrittes ist der Zeilenvektor (1,2,2,-,-,-). Nach der Abarbeitung der Vektoren der direkten Basisklassen werden im letzten Schritt noch die direkten Basisklassen selbst eingetragen und die noch nicht ausgefüllten Felder mit 0 ausgefüllt. Das Ergebnis ist der Spaltenvektor (1,2,2,1,2,0). Alle Felder mit Ausnahme des letzten Feldes dieses Vektors müssen hier in der letzten Zeile einen Eintrag ungleich 0 enthalten — jede Basisklasse muß auf irgend einem Wege erreichbar sein. Die Felder [x][x] der Tabelle müssen 0 sein, die Felder der letzten Spalte auch (man könnte diese letzte Spalte folglich weglassen).

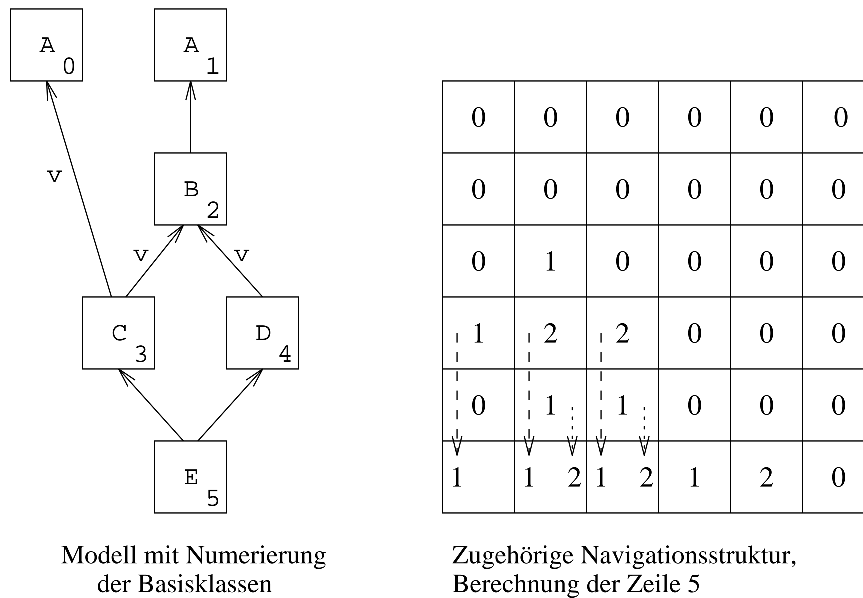


Abbildung 7.5: Aufbau der Navigationsstruktur der zum Modell gehörigen Klasse *E*.

Anhand dieses Feldes kann jetzt die Initialisierungsliste der statischen Navigationsstruktur erstellt werden:

```
static int bcnt[6][6]= { {0, 0, 0, 0, 0, 0}, {0, 0, 0, 0, 0, 0},
                         {0, 1, 0, 0, 0, 0}, {1, 2, 2, 0, 0, 0},
                         {0, 1, 1, 0, 0, 0,} {1, 2, 2, 1, 2, 0}}
```

In den Elementarobjekten kann jedoch nur ein *int *** eingetragen werden:

```
static int *bcnv[6] = { (int *)bcnt[0], (int *)bcnt[1],...}
static int **baseClassNavigation = (int **) bcnv;
```

7.5 Vom Generator erzeugte Methoden und Funktionen

Für jede SWS-Klasse werden durch den Generator zusätzliche Methoden, aber auch normale Funktionen, generiert. Eine Aufzählung dieser Methoden findet man in Anhang F. Diese Methoden und Funktionen kann man grob in zwei Arten unterteilen. Auf der einen Seite sind die für die Verwaltung benötigten Methoden und Funktionen. Diese dürfen vom Applikationsprogrammierer nicht verwendet werden. Auf der anderen Seite gibt es Methoden, welche speziell zur Verwendung durch den Applikationsprogrammierer entworfen wurden. Auf einige dieser Methoden soll nun in den folgenden Abschnitten eingegangen werden.

7.5.1 Methoden für die Verwaltung

In diesem Abschnitt wird kurz auf einige der vom Generator zur Verwaltung des SWS erzeugten Methoden der SWS-Objekte eingegangen. Eine vollständige Dokumentation dieser Methoden ist in [ZSa] enthalten, einige Beispiele sind in den Anhängen A und B zu finden.

Die Konstruktoren

Die Konstruktoren sind nur für den Aufruf der Konstruktoren der Basisklassen und Data-Members und die Initialisierung der Base Class Id's zuständig. Der Rumpf des Konstruktors enthält nur die Initialisierung der Base Class Id's, der dazu notwendige Zähler ist Member des von allen SWS-Klassen virtuell geerbten Generischen Objekts. Der Aufbau der in Kapitel 3 beschriebenen Objektstruktur wird durch den Konstruktor des Generischen Objektes durchgeführt. Der hierzu benötigte Zeiger auf die Top-Level-Information wird beim Aufruf des Konstruktors des Generischen Objektes durch den Konstruktor des Top-Level-Objektes initialisiert und als Parameter der Konstruktoren durch die Konstruktorhierarchie durchgegeben.

Der generierte Copy-Konstruktor löst die Exception **IllegalCopyConstructor** aus (siehe Kapitel 4.3.2).

Methoden zur Ermittlung von Pointern auf Basisklassen

Die Methode **virtual void *getBaseClassPointer(BaseClassId &)** wird zur Umwandlung eines sws-globalen Zeigers auf ein Objekt in einen lokalen Zeiger auf dieses Objekt benötigt. Diese Methode nutzt die Navigationsstruktur. Ein Beispiel ist in Anhang A abgedruckt.

Die Methode **virtual void *getBaseClassPointer(ClassId &)** ermittelt den Pointer auf die durch die übergebene ClassId identifizierte Basisklasse des Objektes. Diese Methode wird zum Beispiel bei Anforderung eines Objektes aus dem SWS anhand der Objekt-Id benötigt. Der Objektmanager liefert nur einen Pointer auf das zugehörige Generische Objekt. Eine Umwandlung in einen Zeiger auf ein Objekt wird durch diese Methode bewerkstelligt. Dieser Pointer kann ein Pointer auf eine beliebige Basisklasse sein, solange diese eindeutig ist. Die Generierung dieser Methode ist ziemlich aufwendig. Es muß für jede Basisklasse im Objekt geprüft werdem, ob sie eindeutig und über welche Basisklasse sie erreicht werden kann. Dies geschieht anhand von auf dem Modell ausgeführten Algorithmen.

7.5.2 Frei zugängliche Methoden

Diese Methoden sind für die Programmierung von SWS-Applikationen notwendige Methoden. Eine vollständige Dokumentation über die Realisierung dieser Methoden findet man in [ZSa], die Verwendung wird in [ZSb] näher erläutert.

Erstellung eines neuen Objektes

Ein neues SWS-Objekt der Klasse „classname“ wird mittels der statischen Methode **classname *classname::newObject()** erstellt. Die Methode liefert einen Pointer auf ein Objekt der Klasse zurück. Die Herstellung neuer SWS-Objekte auf andere Art ist verboten.

In dieser Methode wird nicht nur ein neues Objekt mittels *new* erzeugt. Das neue Objekt wird auch im SWS angemeldet (d.h. es bekommt eine TLOID, ...). Dabei wird, in Kooperation mit dem Objektmanager, darauf geachtet, daß bei Ausführung dieser Methode während einer Methodenverteilung nur ein einziges Objekt entsteht.

Anfordern existierender Objekte

Zu jeder Klasse existieren zwei statische Methoden zum Anfordern im SWS schon existierender Objekte: **classname *classname::getObject(OID &)** und **classname *classname::getObjectByName(char *)**. Die erste erlaubt die Anforderung eines Objektes mittels einer Objekt-Id. „classname“ muß eine eindeutige Basisklasse dieses Objektes sein. Die zweite der Methoden erlaubt die Anforderung anhand eines für ein Objekt mittels **bool nameObject(char *)** vergebenen Namens. **nameObject(...)** ist ebenfalls eine vom Generator erzeugte Methode einer Klasse und erlaubt die Benennung eines Teilobjektes (siehe Kapitel 6.5). Einem Teilobjekt können damit beliebig viele Namen zugeordnet werden.

7.6 Benutzung der SWS-Objekte

Um eine SWS-Klasse zu verwenden, muß lediglich, wie bei anderen Klassen auch, das Header-File (*.h) mittels *#include* eingebunden werden. Das *.C-File wird separat übersetzt. Zusätzlich muß die Klasse noch in den SWS eingefügt werden, damit sie dynamisch durch die Objektfabrik nachgeladen werden kann. Nähere Information hierzu findet man in [ZSb].

Kapitel 8

Ausblick

In dieser Diplomarbeit und der damit verbundenen Studienarbeit wurde ein Grundstein für den zu entwickelnden Shared Workspace gelegt. Einige der zu verwirklichenden Konzepte des SWS wie zum Beispiel das Session Management wurden dabei durch die Aufgabenstellung komplett ausgeklammert und müssen in weiteren Arbeiten entwickelt werden. Andere Aspekte hingegen wurden durch einfache Algorithmen abgehandelt, die unbedingt durch wirkungsvollere Mechanismen zu ersetzen sind. Es bleibt daher auch nach Abschluß dieser Arbeiten noch genug Stoff für weitere Arbeiten übrig.

Die momentane Lösung der Speicherung der SWS-Objekte in Dateien muß auf eine solidere Grundlage gestellt werden. Denkbar ist hier die Verwendung einer (eventuell sogar objektorientierten) Datenbank. Bei Verwendung einer objektorientierten Datenbank kann in diesem Zusammenhang auch über eine Speicherung des zum Nachladen benötigten Codes in dieser Datenbank nachgedacht werden.

Eine große Aufgabe ist die Ersetzung des in dieser Arbeit entwickelten Sperrmechanismus durch dem SWS angemessenere Mechanismen. Die Palette der Möglichkeiten ist dabei sehr weit gestreut — verschiedene Arten an Transaktionskonzepten, History-Mechanismen, ... In diesem Zusammenhang ist sicher auch eine genaue Definition der Schnittstelle zwischen den SWS- und den Viewobjekten festzulegen.

Auch am Programmgenerator kann noch vieles verbessert werden. Die Bearbeitung des Implementationsfiles beruht auf einer einfachen Art der Analyse, die sehr empfindlich gegenüber Fehlern in der Eingabe ist. Eine sehr sinnvolle Ergänzung wäre die vollständige syntaktische Analyse des Implementationsfiles. Hierdurch könnte einerseits die Stabilität bei Fehlern im Programm wesentlich verbessert werden. Auf der anderen Seite könnte so durch genaue Prüfung die Verwendung von Methoden verhindert werden, die eigentlich durch den Applikationsprogrammierer nicht verwendet werden sollen (z.B. alle durch den Programmgenerator generierten Verwaltungsmethoden).

Die Krönung der Entwicklung wäre ein vollständiger Compiler für verteiltes C++, in dem das Prinzip des SWS komplett realisiert ist.

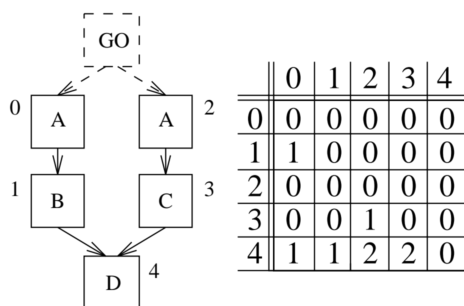
Anhang A

Verwendung der Navigationsstruktur

`void *getBaseClassPointer(BaseClassId bCId) = 0;`
ist eine virtuelle Methode des Generischen Objekts. Diese Methode dient der Ermittlung eines Zeigers auf eine Basisklasse innerhalb des zum GO gehörigen Objektes. Die folgenden Implementationen zeigen die Verwendung der Navigationsstruktur (siehe Kapitel 3.2.3) anhand der unten abgebildeten Vererbungshierarchien.

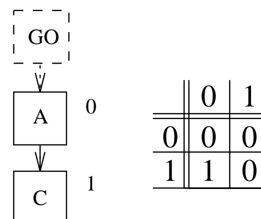
```
void *A::getBaseClassPointer(BaseClassId bCId){
   if (bCId == baseClassId) // baseClassId von A
      return (void *)this;
   else
      // verheerender Fehler, Basisklasse nicht existent
}

void *B::getBaseClassPointer(BaseClassId bCId){
   if (bCId == baseClassId) // baseClassId von B
      return (void *)this;
   else
```



	0	1	2	3	4
0	0	0	0	0	0
1	1	0	0	0	0
2	0	0	0	0	0
3	0	0	1	0	0
4	1	1	2	2	0

Objekt D mit Navigationsstruktur



	0	1
0	0	0
1	1	0

Objekt C mit Navigationsstruktur

```
      switch(nav_structure[baseClassId][bCId]){
       case 1  : return A::getBaseClassPointer(bCId);
                 break;
       default : // verheerender Fehler, Basisklasse nicht existent
      }
}

void *C::getBaseClassPointer(BaseClassId bCId){
   if (bCId == baseClassId) // baseClassId von C
      return (void *)this;
   else
      switch(nav_structure[baseClassId][bCId]){
       case 1  : return A::getBaseClassPointer(bCId);
                 break;
       default : // verheerender Fehler, Basisklasse nicht existent
      }
}

void *D::getBaseClassPointer(BaseClassId bCId){
   if (bCId == baseClassId) // baseClassId von D
      return (void *)this;
   else
      switch(nav_structure[baseClassId][bCId]){
       case 1  : return B::getBaseClassPointer(bCId);
                 break;
       case 2  : return C::getBaseClassPointer(bCId);
                 break;
       default : // verheerender Fehler, Basisklasse nicht existent
      }
}
```

Anhang B

Beispiel zur Methodenverteilung

Abbildung B.1 zeigt die Definition der verteilbaren Methode `method1` der SWS-Klasse A. Der für eine Methodenverteilung dieser Methode benötigte Code wird hier beispielhaft dargestellt (vgl. Kapitel 4.3).

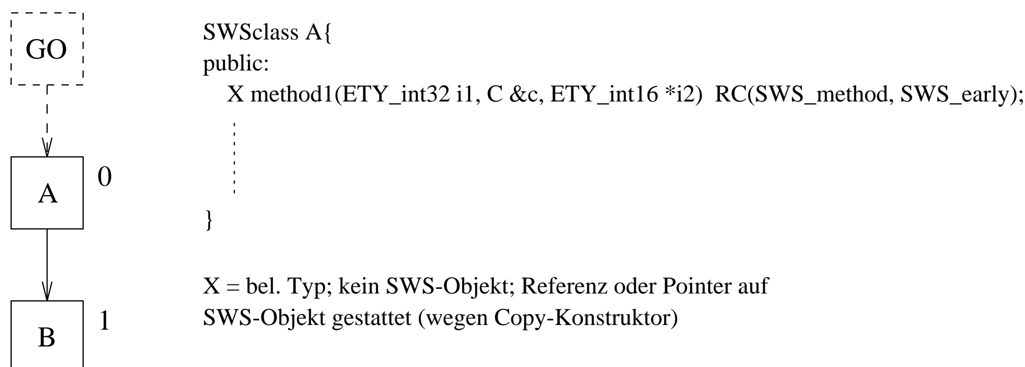


Abbildung B.1: Vererbungshierarchie der Klasse B und Ausschnitt der SWS-Klassendefinition von A.

```
// Replikationscharakteristik schon ausgewertet
// Entscheidung fuer sofortige Verteilung der Methode gefallen
CodeBuffer *buf = new CodeBuffer(29);// 29 Byte, statisch berechenbar
TLIset tliset; // leere Menge von Top-Level-Objekten
MethodId mid(baseClassId,0); // baseClassId von A(dynam.), erste
                  // verteilbareMethode in A (statisch), je ein Byte
mid.encode(buf);// Methoden-Id in Buffer kodieren,2 Byte
i1.encode(buf); // ETY_int32 in Buffer kodieren, 4 Byte
OID oid = c.getOID(); // Objekt-Id des SWS-Objektes c ermitteln
oid.encode(buf); // Objekt-Id von c kodieren, 20 Byte
c.baseClassId.encode(buf); // baseClassId von c, 1 Byte
tliset.add(c.getTliptr()); // Top-Level-Objekt von c muss auf
```

```
                              // Remote-Knoten existieren
   i2->encode(buf); // ETY_int16 in Buffer kodieren, 2 Byte
   om.methodDistribution(*this, tliset, buf, 29);
```

`om` ist der Objektmanager (globale Variable)

Die im Generischen Objekt implementierte Methode zur Zerlegung der Kodierung des Methodenaufrufes in Methoden-Id und Parameter-Kodierung:

```
GO::remote_execution(CodeBuffer *buf){
  MethodId mid;
  mid.decode(buf);
  remote_execution(mid, buf);
  em.message(getNodeId(),getOid(), buf);
}
```

`em` ist der Eventmanager. Die Methode *message* des em realisiert die Meldeschnittstelle. In diesem Fall wird die (vollendete) Ausführung der Methode gemeldet. `getNodeId()` ist eine Funktion, die die Id des Knotens ermittelt, für den diese Funktion ausgeführt wird. Im Falle einer Methodenverteilung wird auf den Remote-Knoten die Id des Initiators der Methodenverteilung ermittelt.

Die Methode `GO::remote_execution(CodeBuffer *buf)` ruft folgende, ebenfalls im Generischen Objekt definierte (rein) virtuelle Methode auf:

```
virtual GO::remote_execution(MethodId, CodeBuffer *) = 0;
```

Dies hat den Aufruf einer der hier implementierten Funktionen zur Folge:

```
A::remote_execution(MethodId mid, CodeBuffer *buf){
  if (mid.baseClassId == baseClassId)
    switch(mid.methodId){
     case 0 : { // erste verteilbare Methode, hier "method1"
                ETY_int32 param1;
                param1.decode(buf);
                OID oid;
                BaseClassId bcid;
                oid.decode(buf);
                bcid.decode(buf);
                GO *object = om.getObject(oid);
                C *param2 = (C *)object->getBaseClassPointer(bcid);
                ETY_int16 param3;
                param3.decode(buf);
                X result = A::method1(param1, *param2, &param3);
              }
```

```
              break;
     case 1 : // weitere verteilbare Methoden
     .......
     default : // Fehler, keine weitere Methode
    }
  else
   // Fehler, keine weiteren Basisklassen
}

B::remote_execution(MethodId mid, CodeBuffer *buf){
  if (mid.baseClassId == baseClassId)
    switch(mid.methodId){
     case 0 : // 1. verteilbare Methode von B
              ......
              break;
     case 1 : // weitere verteilbare Methoden
     .......
     default : // Fehler, keine weitere Methode
    }
  else // remote_execution(...) in einer der Basisklassen aufrufen
    switch(nav_structure[baseClassId][mid.baseClassId]){
       case 1  : A::remote_execution(mid, buf);
                 break;
       default : // Fehler, keine weiteren Basisklassen
    }
}
```

remote_execution(CodeBuffer *buf) ist Methode des Generischen Objektes und wird vom Objektmanager aufgerufen. Diese ruft dann die virtuelle Methode remote_execution(MethodId mid, CodeBuffer *buf) auf. Wird der Aufruf von *method1* auf ein Objekt der Klasse B per Methode verteilt, so wird beim Aufruf von remote_execution(MethodId mid, CodeBuffer *buf) die Implementation B::remote_execution(. . .) aufgerufen. Von dieser wird A::remote_execution(MethodId mid, CodeBuffer *buf) aufgerufen und von da letztendlich nach Dekodierung der Parameter A::method1(. . .).

Die bei der Dekodierung der SWS-Objekte verwendete Methode getBaseClassId wird in Anhang A genauer erläutert.

Anhang C

Die Lockalgorithmen

C.1 Der Lockalgorithmus beim Initiator

Wie in Kapitel 5 ausgeführt wurde, muß bei einer Methodenverteilung die Lockoperation auf den Remote-Knoten gesondert behandelt werden. In diesem Abschnitt wird der Algorithmus aufgeführt, der auf dem Knoten abläuft, der den Lock beantragt. Der Algorithmus für die Remote-Knoten wird im nächsten Abschnitt vorgestellt.

Der hier vorgestellte Algorithmus gehört zu jenen Algorithmen, deren Ausführung sich nicht mit der Neuverteilung eines der betroffenen Objekte verträgt. Aus diesem Grund darf während der Ausführung des Lockalgorithmuses auf ein Objekt dessen D-Lock nicht gesetzt sein.

Der Algorithmus bekommt als Parameter das Lock Request Set (LRS) und den Locktyp ltype übergeben. Das LRS ist eine Menge von Top-Level-Objekten (eigentlich von Top-Level-Informationen), auf die ein Lock vom Typ ltype gesetzt werden soll. Das Ergebnis des Algorithmus ist ein Lock-Handle. Die in Kapitel 5 beschriebenen Lockoperationen müssen vor Ausführung dieses Algorithmus die Menge der zu lockenden TLO's bestimmen und diese TLO's lokal verfügbar machen (dies gilt nur für die Operationen auf dem anfordernden Knoten, eventuelle Remote-Ausführungen müssen die LRS nicht selbst bestimmen).

1. Ist LRS leer ? Wenn ja, dann gebe ein leeres Lock-Handle zurück
2. Lock Accepted Set LAS = {}, Lock Denied Set LDS = {}
3. Für alle Objekte aus LRS führe folgende Schritte aus:
   (a) Untersuche lokal, ob der gewünschte Lock auf das Objekt möglich bzw. nötig ist (siehe Abbildung C.1).
   (b) Wenn Lockalgorithmus während Methodenverteilung ausgeführt wird: Verteile das Objekt auf die Remote-Knoten, wo es noch nicht lokal verfügbar ist. Ist diese Verteilung nicht möglich, kann das Objekt nicht gelockt werden

	vorhanden					
		Kein Lock	(Pre-)Lock			
			Read		Write	
			sk	ak	sk	ak
gefordert	Read	+	-	+	+	--
	Write	+	+	--	-	--

sk : der anfordernde Knoten hat einen Lock (Prelock nicht möglich)
ak : ein anderer Knoten hat einen (Pre-)Lock
+ : Lock ist möglich
- : Lock ist nicht nötig
-- : Lock ist nicht möglich

Abbildung C.1: Kompatibilitätsmatrix für lokale Lockentscheidung

(c) Setze lokal einen Prelock vom Typ ltype auf das Objekt. Ist dies zwischenzeitlich nicht mehr möglich (eventuell hat in der Zwischenzeit ein anderer Knoten einen entsprechenden (Pre-)Lock gesetzt), kann das Objekt nicht gesperrt werden.

(d) Verschicke Prelock-Anforderung vom Typ ltype für diesen Knoten an alle anderen Replikate des Objektes

(e) Erwarte Antworten. Bei einer negativen Antwort (Prelock nicht möglich) kann der Lock nicht gesetzt werden. Verschicke an alle Replikate, die den Prelock gesetzt haben die Nachricht, daß Prelock gelöscht werden soll

(f) Wandle den lokalen Prelock in einen Lock um.

(g) Wenn Lockalgorithmus während Methodenverteilung durchgeführt wird: Verteile an alle anderen Replikate des Objektes die sich nicht auf Remote-Knoten der Methodenverteilung befinden die Aufforderung zur Umwandlung des Prelocks in einen Lock. Ansonsten verteile diese Aufforderung an alle anderen Replikate des Objektes.

(h) Falls ltype==Write-Lock, dann setze das Verteilungsflag des Objektes auf „noch nicht geprüft“.

4. Jedes Objekt, das in den Schritten 3a bis 3g gelockt werden konnte, wird zu LAS hinzugefügt. Jene, die nicht gelockt werden konnten, werden zu LDS hinzugefügt.

5. Lock-Handle = (LAS, LDS, ltype)

6. Falls Lockalgorithmus während Methodenverteilung ausgeführt wird: Verteile das Lock-Handle auf alle Remote-Knoten.

7. Gebe Lock-Handle zurück

Die Ausführung der Schritte 3a bis 3g für ein Objekt wird abgebrochen sobald festgestellt wird, daß auf das Objekt die Sperre nicht gesetzt werden kann.

C.2 Der Lockalgorithmus auf einem Remote-Knoten

Auf dem Remote-Knoten sind theoretisch keine Parameter notwendig, alles Wissenswerte wird vom Initiator der Methodenverteilung mittels des Lock-Handles übertragen.

1. Warte auf Lock-Handle vom Initiator (falls noch nicht in Warteschlange vorhanden)
2. Für alle Objekte aus LAS (LAS aus Lock-Handle): Wandle EINEN für den Initiator gesetzten Prelock vom Typ ltype in einen Lock vom Typ ltype um. Dabei ist zu beachten
   - Es können mehrere solche Prelocks des Typs ltype für den Initiator auf dieses Objekt gesetzt sein
   - Mehrere Knoten können Prelocks besitzen (dann darf allerdings kein Write-(Pre-)Lock dabei sein)
   - Ein Knoten darf keine zwei Locks desselben Typs auf ein Objekt besitzen

C.3 Ankommende Prelock-Anforderung

Eine ankommende Prelock-Anforderung wird immer sofort bearbeitet. Als Parameter stehen das zu lockende Objekt (TLI), der gewünschte Locktyp ltype und der Knoten, der die Sperre setzen möchte, zu Verfügung.

1. Teste lokal, ob Prelock für anfordernden Knoten für das gewünschte Objekt gewährt werden kann (siehe Abbildung C.2).
2. Wenn Prelock möglich ist dann den entsprechenden Prelock für den anfordernden Knoten setzen und eine positive Antwort geben („Prelock möglich"), ansonsten negative Antwort („Prelock nicht möglich") zurückgeben.

		vorhanden				
		Kein Lock	(Pre-)Lock			
			Read		Write	
			sk	ak	sk	ak
gefordert	Read	+	+	+	+	-
	Write	+	+	-	+	-

sk : der anfordernde Knoten hat einen (Pre-)Lock
ak : ein anderer Knoten hat einen (Pre-)Lock
\+ : Prelock kann gewährt werden
\- : Prelock kann nicht gewährt werden

Abbildung C.2: Kompatibilitätsmatrix für Lockentscheidung bei Prelock-Anforderung durch andere Knoten

C.4 Der Unlockalgorithmus

Parameter dieses Algorithmus ist ein Lock-Handle. Während der Ausführung des Unlock auf ein Objekt darf dessen D-Lock nicht gesetzt sein.

Für alle Objekte o aus LAS (LAS aus Lock-Handle):

1. Bei Bedarf dataDistribution(o) bzw. additionalDataDistribution(o) aufrufen (bei momentan laufender Datenverteilung bzw. bei momentan laufender Methodenverteilung)

2. Lösche den Lock in der Lockinformation des Objektes. Bei Methodenverteilung muß hier auf den Remote-Knoten darauf geachtet werden, daß ein vom Initiator der Methodenverteilung gesetzter Lock gelöscht werden muß.

3. Unlock-Nachrichten an Replikate versenden. Bei Methodenverteilung macht dies nur der Initiator und spart dabei die Remote-Knoten aus.

Anhang D

Algorithmen zur Objektverwaltung

D.1 Algorithmus zur Verteilung eines Objekts

In diesem Abschnitt wird der in Abschnitt 6.6.1 kurz besprochene Algorithmus und die damit verbundenen Algorithmen ausführlich dargestellt.

D.1.1 Der Algorithmus

Parameter des Algorithmus : TLOID des zu verteilenden Objektes und die Menge der Zielknoten, auf die das Objekt verteilt werden soll.

Ergebnis : OK (Objekt konnte auf alle Zielknoten verteilt werden) oder NOK (O. konnte nicht auf alle Zielknoten verteilt werden).

1. Teste lokal, ob Verteilung des Objektes von diesem Knoten möglich ist:

   - Hat anderer Knoten Write-(Pre)Lock auf das Objekt? Wenn ja, beauftrage diesen Knoten mit der Verteilung (z.B. durch Aufruf dieses Algorithmus auf dem anderen Knoten per RPC). Antwort des beauftragten Knotens: OK, NOK: gebe dies als Ergebnis zurück
   - in allen anderen Fällen ist Verteilung von hier möglich

   Dieser Schritt könnte entfallen, da in 4 derselbe Test nochmals durchgeführt wird. Man spart sich aber durch diese doppelte Ausführung eventuell das Anfordern eines Distribution-Locks.

2. Setze Distribution-Lock (s. unten) für das zu verteilende Objekt

3. Existiert Objekt noch? Wenn nein, dann Ergebnis NOK

4. Teste nochmal wie bei 1 (eventuell hat in der Zwischenzeit ein anderer Knoten einen Write-Lock angefordert. Gebe in diesem Fall den Distribution-Lock frei, bevor der andere Knoten mit der Verteilung beauftragt wird)

5. Falls die Applikation auf dem Knoten im Moment eine verteilte Methode laufen hat, muß je nach Zustand des Verteilungsflags des Objektes einer der folgenden Schritte durchgeführt werden:

   - Verteilungsflag==Verteilung: füge die Menge der Knoten, auf die das Objekt jetzt verteilt werden soll, zu der Menge der Knoten hinzu, auf die Änderungen des Objektes zusätzlich datenverteilt werden (stelle sicher, dass keine Remote-Knoten und der eigene Knoten dabei sind)
   - Verteilungsflag==keine_Verteilung: Das Objekt wurde seit Beginn der Methodenverteilung geändert. Zum Zeitpunkt der Änderung war festgestellt worden, daß keine zusätzliche Datenverteilung stattfinden muß. Das Objekt wird jetzt in einem inkonsistenten Zustand verteilt, nachfolgende Änderungen während der Methodenverteilung müssen eventuell nachträglich auf die Knoten verteilt werden, auf die das Objekt jetzt neu verteilt wird. Dazu wird das Flag auf „nicht_gepüft" gesetzt, bei der nächsten Änderung muß dadurch erneut auf zusätzliche Datenverteilung geprüft werden.

6. Beauftrage Kodierung der Objektdaten

7. Kodiere Verwaltungsdaten

   - Kodiere TopLevelObjektId, ClassId
   - Kodiere die Liste der Knoten, auf denen das Objekt vorkommt (ohne die neuen Knoten)
   - Kodiere Liste der Lockinformationen
     - Anderer Knoten hat Write-(Pre)Lock: Fehler, darf nicht sein
     - Eigener Knoten hat Write-Lock: einfach die Lockinformationen kodieren
     - Keiner hat Write-(Pre)Lock. Für alle Knoten, die Read-(Pre)Lock haben: Durchsuche knotenspezifische Warteschlange von hinten (d.h. beginne bei neuestem Eintrag). Es wird als erstes gefunden:
       * Methodenverteilung : frage bei Knoten an, ob er im Moment Read-Lock für Objekt hat
       * Lock-Auftrag für zu verteilendes Objekt: Knoten hat im Moment Read-Lock für Objekt
       * Unlock-Auftrag für zu verteilendes Objekt: Knoten hat im Moment keinen Read-Lock für Objekt

       Wenn der betrachtete Knoten im Moment Read-Lock hat, füge diese Info zu den kodierten Lockinfos dazu

8. Verschicke Objekterstellungsauftrag an die Zielknoten der Verteilung (Parameter: kodierte Verwaltungs- und Objektdaten) Antworten der Zielknoten: OK, NOK, warten, bis alle Antworten da
9. neue Lokationen = die Knoten, die OK geantwortet haben. Trage diese Knoten lokal in die TopLevelInformation ein
10. Schicke an alle Knoten, auf denen das Objekt existiert (siehe lokale TopLevelInformation), die Liste der neuen Lokationen, warte, bis überall eingetragen
11. Trage die neuen Lokationen in Information zum Objekt auf Sekundärspeicher ein
12. lösche Distribution-Lock (lokal und global)
13. Ergebnis : NOK, wenn eine der Antworten in 8 NOK war, OK sonst

Die Ermittlung des momentanen Lockzustandes des Objektes in 7 bedingt, daß die Einträge in den senderspezifischen Warteschlangen der Objektmanager erst dann entfernt werden, wenn der Eintrag komplett abgearbeitet ist (das heißt zum Beispiel bei einer Methodenverteilung, daß die Methode fertig abgelaufen sein muß bevor der Auftrag zur Ausführung der Methode aus der Warteschlange genommen wird).

D.1.2 Setzen des Distribution-Locks

Algorithmus zum Anfordern des D-Locks

Parameter: Objekt (TLOID) des Objektes, auf das der D-Lock gesetzt werden soll.

Der Lock wird auf jeden Fall gesetzt. Falls zwei Knoten diesen Lock gleichzeitig setzen wollen, kann das Setzen des Locks etwas länger dauern. Ein Dead-Lock ist nicht möglich, da ein gesetzter D-Lock auf jeden Fall nach der Verteilung des Objektes wieder gelöscht wird, wechselseitiges Warten kann nicht auftreten.

1. Ist schon ein Distribution-Lock auf das Objekt gesetzt oder in Wartestellung (Wartestellung: Ein anderer Knoten hat schon einen D-Lock angefordert. Das Setzen dieses Locks ist im Moment lokal nicht möglich, da gerade eine der Operationen läuft, die mit dem D-Lock ausgeschlossen werden sollen. Es wird auf das Ende der Operation gewartet, bevor der D-Lock gesetzt wird)? Wenn ja, kurze Zeit warten (bis der andere D-Lock freigegeben wird), gehe zu 1
2. Setze D-Lock lokal, falls im Moment nicht möglich (Applikation macht gerade etwas auf dem Objekt, was einen D-Lock ausschließt) dann warte bis möglich (trage dazu die Anforderung irgendwo ein, damit keine D-Lock-Anforderung von außen zuvorkommen kann (Wartestellung))
3. Verteile an alle Knoten, auf denen das Objekt vorkommt, eine D-Lock-Anforderung, warte auf Antworten

4. Alle Antworten positiv : D-Lock gewährt. Nicht alle Antworten positiv: versende auf alle Knoten, die positiv geantwortet haben die Nachricht, dass D-Lock gelöscht werden soll, lösche lokalen D-Lock, warte nichtdetermin. und gehe dann nach 1

D-Lock-Anforderung von anderem Knoten

Parameter: anfordernder Knoten, Objekt

Die Ausführung wird nicht in die absenderspezifische Warteschlange gestellt, sondern sofort durchgeführt.

1. Ist auf das Objekt schon ein D-Lock gesetzt (oder in Wartestellung)? Wenn ja, dann gebe negative Antwort.
2. Setze D-Lock auf Objekt. Falls dies im Moment nicht möglich ist, dann warte, bis dies wieder möglich ist und setze D-Lock
3. positive Antwort

D.1.3 Ankommender Objekterstellungsauftrag

Dieser Auftrag kommt vom Verteilungsalgorithmus aus D.1.1.

Parameter: kodierte Verwaltungs- und Objektdaten

Ergebnis: OK, NOK

1. Dekodiere Verwaltungsdaten: TopLevelId; KlassenId; Menge der Knoten, auf denen das Objekt vorkommt; Lock-Infos
2. Fordere von der Objektfabrik ein Objekt mit der gewünschten Klassen-Id an (gibt Pointer auf Top-Level-Information zurück) siehe Abschnitt 6.3
3. Veranlasse Dekodierung der Objektdaten durch TopLevelInformation
4. Trage Verwaltungsdaten in die TopLevelInformation ein
5. Setze Distribution-Lock in Objekt für verteilenden Knoten
6. Hänge Objekt in die Objektstruktur ein, stelle dabei sicher, daß das Objekt nicht schon vorhanden war
7. Falls kein unerwarteter Fehler aufgetreten ist : Antwort OK, sonst : NOK

D.2 Anforderung eines existierenden Objektes

Die hier ausgeführte Methode des Objektmanagers dient zur Anforderung eines bereits existierenden Objektes. Sie wird von durch den Programmgenerator erzeugten Methoden der SWS-Klassen aufgerufen. Nähere Betrachtungen zu dieser Methode findet man in Abschnitt 6.6.2

Parameter: Objekt-Id (Nicht TopLevelObjekt-Id !!)

Ergebnis: GO * auf Objekt mit gewünschter Objekt-Id, falls nicht möglich: NULL

GO *getObject(OID oid){

1. Ermittle zum Objekt gehörige TLOID
2. Wurde der Algorithmus von Methode aufgerufen, die durch eine Methodenverteilung auf einem Remote-Knoten läuft (RC.rg==remote)? Wenn ja, dann warte auf Übermittlung des Ergebnisses durch Initiator der Methodenverteilung:
   - Anforderung möglich gewesen: ermittle den gewünschten Pointer aus der lokalen Objektstruktur, gebe Pointer zurück
   - Anforderung nicht möglich gewesen: gebe NULL zurück
3. Existiert Objekt schon lokal und ist noch gültig ? Wenn ja, gehe nach 17. Wenn Objekt nicht mehr gültig : negatives Ergebnis an eventuelle Remote-Knoten und return NULL
4. Existiert Objekt auf Sekundärspeicher ?
   - nein: negatives Ergebnis auf eventuelle Remote-Knoten, return NULL (da dann nicht vorhanden)
   - ja: locke Information zu Objekt (auf Sekundärspeicher) falls Lock nicht möglich da Objekt im Moment gelöscht wurde: negatives Ergebnis auf eventuelle Remote-Knoten, return NULL
5. Ist in Information zu Objekt (auf Sekundärspeicher) schon ein Knoten eingetragen? wenn ja, dann gehe zu 6, wenn nein, dann gehe zu 10
6. Entnehme einen der Knoten aus Information zu Objekt, prüfe ob dieser Knoten noch aktiv ist, falls nicht, lösche den Knoten aus Objektinformation und gehe zu 6
7. gebe Lock auf Objekt-Information frei
8. Beauftrage den OM des in 6 ermittelten Knotens mit der Verteilung des gewünschten Objekts auf eigenen Knoten und auf eventuelle Remote-Knoten.

9. konnte Verteilung durchgeführt werden?
   - ja :
     - falls Methode auch auf anderen Knoten läuft : positives Ergebnis auf Remote-Knoten
     - ermittle den gewünschten Pointer aus Objektstruktur und gebe diesen zurück
   - nein
     - falls Methode auch auf anderen Knoten läuft : negatives Ergebnis auf Remote-Knoten
     - gebe NULL zurück
10. Hole Objektkodierung von Sekundärspeicher
11. Dekodiere Verwaltungsinformation TLOID, ClassId
12. Fordere von Objektfabrik ein Objekt mit der ermittelten ClassId an
13. Trage Verwaltungsinfo in Objekt (TLI) ein:
    - TLOID, (ClassId ist schon von Konstruktor eingetragen)
    - keine Locks
    - als Lokation nur eigener Knoten
14. Trage Objekt in lokale Objektstruktur ein
15. Schreibe eigene KnotenId in Objekt-Info auf Sekundärspeicher, gebe Lock auf Objekt-Info frei
16. falls Fehler in 10 – 15:
    - falls Methode auch auf anderem Knoten läuft : negatives Ergebnis auf Remote-Knoten
    - return NULL
17. - Falls Methode auch auf anderem Knoten läuft: verteile TL-Objekt auf jene Remoteknoten, die das Objekt noch nicht lokal haben
      - Ergebnis der Verteilung OK: Positives Ergebnis auf Remote-Knoten, gebe Pointer auf gewünschtes Objekt zurück
      - Ergebnis der Verteilung NOK: Negatives Ergebnis auf Remote-Knoten, gebe NULL zurück
    - sonst : gebe Pointer auf gewünschtes Objekt zurück

} // getObject(...)

Die in dem Algorithmus mehrfach erwähnte Objekt-Information (auf Sekundärspeicher) ist die beim Objekt gespeicherte Liste der Knoten, auf denen das Objekt im Moment lokal gehalten wird.

Anhang E

Grammatik der SWS-Klassen

E.1 Die Klassendefinition

header-file:

  includes$_{opt}$ *pre-declarations*$_{opt}$ *class-definition*

includes:

  cpp-includes sws-includes$_{opt}$
  sws-includes

cpp-includes:

  cpp-includes cpp-include
  cpp-include

cpp-include:

  #include **"***filename***"**
  #include **<***filename***>**

sws-includes:

  sws-includes sws-include
  sws-include

sws-include:

  #SWSinclude **"***SWS-class-name****.scd*** **"**

pre-declarations:

  pre-declarations pre-declaration
  pre-declaration

pre-declaration:

  SWSclass *identifier* ***;***

class-definition:

class-head { *class-id-spec member-list* }***;***

class-head:

SWSclass *identifier base-spec*$_{opt}$ *rc-spec-dist*$_{opt}$

base-spec:

: *base-list*

base-list:

base-specifier
base-list **,** *base-specifier*

base-specifier:

virtual$_{opt}$ *access-specifier*$_{opt}$ *SWS-class-name*

acess-specifier:

public
protected
private

rc-spec-dist:

RC(*ri , rg*)

ri:

RI_DATA
RI_METHOD
RI_DEFAULT

rg:

RG_EARLY
RG_LATE
RG_DEFAULT

class-id-spec:

SWSClassId **“***uuid-string***“;**

member-list:

access-specifier ***:*** *member-list*
member-declaration member-list$_{opt}$
member-declaration

member-declaration:

data-member
function-member

data-member:

SWS-class-name identifier ***;***
elementary-object-class identifier ***;***

elementary-object-class:
SWS_int8
SWS_int16
...

function-member:
virtual *nonvirtual-function-member pure-spec*$_{opt}$ **;**
nonvirtual-function-member **;**

nonvirtual-function-member:
type$_{opt}$ *identifier* **(** *param-list*$_{opt}$ **)** *rc-spec*$_{opt}$ *lock-spec*$_{opt}$
type$_{opt}$ **operator** *operator* **(** *parameter*$_{opt}$ **)** *rc-spec*$_{opt}$ *lock-spec*$_{opt}$
operator *type* **()** *rc-spec*$_{opt}$ *lock-spec*$_{opt}$

type:
identifier ptr-op$_{opt}$

operator: one of
```
+    -    *    /    %    ^    &    |    ~    !    =    <
>    +=   -=   *=   /=   %=   ^=   &=   |=   <<   >>   >>=
<<=  ==   !=   <=   >=   &&   ||   ++   --   []
```

param-list:
param-list **,** *parameter*
parameter

parameter:
param-typ identifier param-lock-spec$_{opt}$

param-typ:
elementary-type ptr-op$_{opt}$
SWS-class-name ptr-op
elementary-object-class ptr-op

elementary-type:
ETY_int8
ETY_int16
...

param-lock-spec:
L_READ
L_WRITE

rc-spec:
rc-spec-dist
RC(RC_NOT_DISTRIBUTED)

lock-spec:
param-lock-spec
L_PEEPING

ptr-op:

  *
  &

pure-spec:

  = 0

E.2 Die Implementation

implementation-file:

  includes$_{opt}$ *function-definition-list*

function-definition-list:

  function-definition-list function-definition
  function-definition

function-definition:

  type$_{opt}$ *classname****::****identifier* ***(*** *def-param-list*$_{opt}$ ***)*** { *fct-body* }
  type$_{opt}$ *classname****::operator*** *operator* ***(*** *def-parameter*$_{opt}$ ***)*** { *fct-body* }
  *classname****::operator*** *type* ***()*** { *fct-body* }

def-param-list:

  def-param-list ***,*** *def-parameter*
  def-parameter

def-parameter:

  param-typ identifier

Anhang F

Zusätzliche Members der generierten Klassendefinitionen

Zusätzliche Daten der Klassen:

- static ClassId classId : Klassen-Id der Klasse
- int baseClassId : Eindeutige Nummer dieser Klasse innerhalb einer Vererbungshierarchie.

Zusätzliche Methoden der Klassen:

- Konstruktoren:
  - classname(); wird von newObject() aufgerufen
  - classname(TLI *) : für alle anderen Gelegenheiten, bekommt Zeiger auf zugehörige TLI übergeben
  - Copy Constructor. Der Aufruf dieses Konstruktors ist verboten. Bei Aufruf wird die Exception **IllegalCopyConstructor** ausgelöst.
- Destruktor
- remote_execution(MethodId, CodeBuffer *); Methode zum Aufruf einer durch Methodenverteilung verteilten Methode (siehe Beispiel in Anhang B)
- virtual void *getBaseClassPointer(BaseClassId &); Methode zum ermitteln des Zeigers auf eine Basisklasse anhand der Base Class Id (Beispiel siehe Anhang A)
- virtual void *getBaseClassPointer(ClassId &); Methode zum ermitteln des Zeigers auf eine Basisklasse anhand der Klassen-Id. Diese Klasse darf nicht mehrmals als Basisklasse vorkommen.

- static classname *newObject(); Statische Methode zum Erstellen eines Objektes der Klasse classname. Dies ist die einzige legale Möglichkeit, ein Top-Level-Objekt zu erzeugen.
- bool nameObject(char *); Methode zur Zuweisung eines Namens an ein SWS-Objekt. Ein Objekt kann mehrere Namen besitzen.
- static classname *getObject(OID &); Methode zum Anfordern eines Objektes anhand der OID.
- static classname *getObjectByName(char *); Methode zum Anfordern eines Objektes anhand eines mittels nameObject(...) vergebenen Namens

Von der Objektfabrik benötigte Funktionen (wahlweise statische Funktionen im „.C"-File oder statischer Member der Klasse):

- static void *doCast(void *obj, ClassId cId); Funktion zur Ermittlung eines Pointers auf eine Basisklasse des Objektes anhand der übergebenen Objekt-Id
- static GO *produceObject(); Funktion zu Produktion eines Objektes der Klasse. Es erfolgt kein Eintrag in den SWS (keine neue TLOID, ...).

Einige der hier aufgeführten Methoden und Funktionen haben noch einen weiteren Parameter. Dieser Parameter ist eine im GO protected definierte Klasse. Er dient einzig und allein zur Verhinderung des Aufrufes der Methoden von „außen". Ein Beispiel ist der Konstruktor classname(TLI *). Dieser Konstruktor muß public deklariert werden, damit SWS-Objekte Member in anderen SWS-Objekten sein können. Top-Level-Objekte sollen aber nur durch die entsprechende statische Methode der Klasse erstellt werden. Die Erweiterung der Parameterliste des Konstruktors um diesen Typ verhindert die Verwendung des Konstruktors außerhalb des Bezugsrahmens der SWS-Klasse. Die Verwendung des Konstruktors in Methoden der Klasse könnte z.B. durch den Generator verhindert werden.

Anhang G

Die Erweiterungen der Methoden

G.1 „Sehende“ Methoden

Schale um „sehende“ Methode:

```
X classname::aPeepingMethod(...){

  AccessModeTestAndSet automaticAccessMode;

  // hier steht die eigentliche Methode aus dem Implementationsfile
};
```

AccessModeTestAndSet ist für das Prüfen der globalen RC und das Setzen des Access Modes zuständig. Das Prinzip der Implementierung dieser Klasse wird im Folgenden verdeutlicht:

```
AccessModeTestAndSet{
  int changed;
  public:
  AccessModeTestAndSet():changed(0){
    if (RCglobal != (default,default)) throw AccessModeError;
    if (AccessMode == AM_Normal){
      AccessMode = AM_Peeping;
      changed=1;
    }
  }
  ~AccessModeTestAndSet(){
    if (changed) AccessMode = AM_Normal;
  }
};
```

G.2 Nicht verteilbare Methoden

Die Schale um nicht verteilbare Methoden hat folgenden Aufbau:

```
X classname::aNonDistributedMethod(...){

  if (AccessMode == AM_Peeping)
       throw AccessError;
       // Aufruf waehrend 'sehenden' Methoden nicht erlaubt


  // testen, ob die vom Applikationsprogrammierer geforderten Locks
  // gesetzt sind :
  LockHandle lh0 = testLockAndTryToSetIfWrong(this, specifiedLock)
  LockHandle lh1 = testLockAndTryToSetIfWrong(&parX, specifiedLockParX)
  .... fuer jeden Parameter, der SWS-Objekt ist, folgt ein solcher Aufruf

  // hier steht die eigentliche Methode aus dem Implementationsfile


  // hier werden die LockHandles automatisch freigegeben
};
```

testLockAndTryToSetIfWrong(...) prüft, ob das übergebene Objekt einen zum übergebenen Lock kompatiblen Lock besitzt. Ist dies der Fall, wird ein „leeres" Lockhandle zurückgegeben. Besitzt das Objekt keinen entsprechenden Lock, wird versucht, den übergebenen Lock zu setzen. Ist dies nicht möglich, wird die Exception LockError ausgelöst.

G.3 Verteilbare Methoden

Die Schale um verteilbare Methoden hat folgenden Aufbau:

```
X classname::aDistributedMethod(...,RI ri, RG rg){

  if (AccessMode == AM_Peeping)
       throw AccessError;
       // Aufruf waehrend 'sehenden' Methoden nicht erlaubt


  // testen, ob die vom Applikationsprogrammierer geforderten Locks
  // gesetzt sind :
  LockHandle lh0 = testLockAndTryToSetIfWrong(this, specifiedLock);
```

```
... analog zu nicht verteilbarer Methode


RCdistributedMethod rc(ri_stat,rg_stat,ri,rg);
// im Konstruktor wird zuerst die gewuenschte RC dieses Aufrufes
// anhand der statisch berechneten RC (ri_stat, rg_stat) und der
// uebergebenen RC (ri,rg) berechnet. Danach wird die tatsaechliche
// RC berechnet und in der globalen RC gespeichert. Im Destruktor
// wird der urspruengliche Wert der globalen RC wieder hergestellt.

AutoDistribution ad;
// Der Konstruktor berechnet, ob Verteilung notwendig oder nicht, und
// wenn ja, welche. Dies kann ueber Methoden einfach abgefragt werden.
// Der Destruktor ruft bei Bedarf die fuer die Beendigung der
// Verteilung notwendigen Methoden.

AutoMessage am;
// Im Destruktor werden mittels der Methode sendMessage(...)
// in Auftrag gegebene Nachrichten an den Eventmanager
// abgesetzt (mittels em.message(...))
// Hiermit wird die Meldung der Verteilung einer Methode nach
// der lokalen Ausfuehrung der Methode realisiert.


if (ad.MethodDistribution()) { // Methode wird verteilt
  // siehe Anhang "Beispiel zur Methodenverteilung"
  ....
  am.Message(getNodeId(),getOid(), buf);
  // siehe auch Anhang "Beispiel zur Methodenverteilung"
  // Aufruf von em.message(...) in remote\_execution(CodeBuffer *)
}


// hier steht die eigentliche Methode aus dem Implementationsfile


// falls die Methodenverteilung stattgefunden hat, wird hier durch
// den Destruktor von am die Meldung an den Event-Manager abgesetzt

// falls diese Methode verteilt wurde, erfolgt hier im Destruktor
// von "ad" automatisch der entsprechende Methodenaufruf zur
// Beendigung der Verteilung
```

```
  // hier wird im Destruktor von rc die urspruengliche globale
  // RC wieder hergestellt

  // hier werden eventuelle automatisch gesetzte Locks durch die
  // Destruktoren der Lockhandles automatisch wieder freigegeben
}
```

RCdistributedMethod und *AutoDistribution* sind Klassen. In ihnen sind Teile der oberen Schale (im Konstruktor) und Teile der unteren Schale (im Destruktor) implementiert (vgl. *AccessModeTestAndSet* im Abschnitt über „sehende“ Methoden).

Abbildungsverzeichnis

2.1 Grobarchitektur des SWS . . . 5

2.2 Trennung in Daten- und Viewobjekte . . . 5

2.3 Architektur des SWS . . . 16

3.1 Schnittstelle des Objekts *Würfel* in Pseudo-Notation . . . 21

3.2 Mögliche Vererbungshierarchien der Klasse *Zoo-Bär* . . . 23

3.3 Normale Vererbung des GO und virtuelle Vererbung des GO . . . 26

3.4 Generatorerzeugte Klassendefinitionen der Klasse C und ihrer Basisklassen 26

3.5 Mögliche Strategien zur eindeutigen Numerierung der Member innerhalb eines Objektes . . . 27

3.6 Aufbau eines SWS-Objektes . . . 29

3.7 Verwaltung der Objekte im Objektmanager . . . 29

3.8 Mehrdeutigkeit bei Basisklassen und deren mögliche Auflösung . . . 30

3.9 Zwei Fälle von Vererbung, die nicht möglich sind . . . 34

4.1 Kodierung eines Integers (4 Byte) in MSB-Order . . . 38

4.2 Problematische Methodenidentifikation . . . 40

4.3 Vererbungshierarchie der Elementarobjekte . . . 41

4.4 Ungünstige Situation für Methodenverteilung . . . 45

4.5 Problematische zusätzliche Methodenverteilung . . . 46

5.1 Kompatibilitätsmatrix für Lockanforderungen . . . 49

5.2 Sperrung der Region um Objekt 1 mit Radius 2 . . . 51

5.3 Kompatibilitätsmatrix für Test auf kompatiblen Lock . . . 52

5.4 Mögliche Zustände eines Locks auf ein Objekt . . . 55

5.5 Erfolgreicher und nicht erfolgreicher Lockversuch auf ein Objekt . . . 55

6.1 Vererbungshierarchie eines zu benennenden Objektes 63

7.1 Aufbau des SWS-Headerfiles C3.scd 71
7.2 Klassendefinitionen der Klasse A und B (in getrennten Headerfiles) 75
7.3 Klassendefinitionen der Klasse A mit Beispielaufruf der Methode *m1*. 78
7.4 Definition der Klasse E mit resultierender Membertabelle und Modell . . . 81
7.5 Aufbau der Navigationsstruktur der zum Modell gehörigen Klasse E. 86

B.1 Vererbungshierarchie der Klasse B und Ausschnitt der SWS-Klassendefinition von A. 92

C.1 Kompatibilitätsmatrix für lokale Lockentscheidung 96
C.2 Kompatibilitätsmatrix für Lockentscheidung bei Prelock-Anforderung durch andere Knoten 97

Tabellenverzeichnis

3.1 Navigationsstruktur der Klasse D 32

7.1 Relation '$\rightarrow$' für RI (links) und RG (rechts) 77

7.2 Relation '$\Rightarrow$' für RI (links) und RG (rechts) 78

Literaturverzeichnis

[CMB+90] T. Crowley, P. Milazzo, E. Baker, H. Forsdick, R. Tomlinson, „MMConf: An Infrastructure for Building Shared Multimedia Applications“, CSCW 90 Proceedings, October 1990

[EGR91] C. A. Ellis, S. J. Gibbs, G. L. Rein, „Groupware — Some Issues and Experiences“, Communications of the ACM, 34(1), January 1991

[ES92] M. A. Ellis, B. Stroustrup, „The Annotated C++Reference Manual“, ISBN 0-201-51459-1, Addison-Wesley, 1992

[Gro93] M. Grossman, „Object I/O and runtime type information via automatic code generation in C++“, Journal of Object Oriented Programing, July-August 1993

[GS87] I. Greif, S. Sarin, „Data Sharing in Group Work“, ACM Transactions on Office Information Systems, 5(2), April 1987

[GSW92] I. Greif, R. Seliger, W. Weihl, „A Case Study Of CES: A Distributed Collaborative Editing System Implemented in Argus“, IEEE Transactions on Software Engineering, Vol. 18, No. 9, September 1992

[Kaba93] Karl Baader, „Objektmodell für einen Shared Workspace“, Diplomarbeit Nr. 1027, Fakultät Informatik,Universität Stuttgart, 1993

[KLL+91] T. Kirsche, R. Lenz, H. Lührsen, K. Meyer-Wegener, H. Wedekind, „CoDraft — eine verteilte Architektur zur Unterstützung von Gruppenarbeit durch Multimediale Objekte“. In Wolfgang Effelsberg, Kurt Rothermel (Hrsg.), *Verteilte Multimedia-Systeme*. GI/ITG, 1993

[Lag] Prof. Lagally, Script zur Vorlesung *Betriebssysteme* an der Uni Stuttgart

[Lip91] S. B. Lippman, „C++Einführung und Leitfaden“, ISBN 3-89319-375-8, Addison-Wesley, 1991

[MLF92] T. W. Malone, K-Y. Lay, C. Fry, „Experiments with Oval: A Radically Tailorable Tool for Cooperative Work“, CSCW 92 Proceedings, November 1992

[ND92] C. Nascimento, J. Dollimore, „Behavior maintenance of migrating object in a distributed object-oriented environment“, Journal of Object Oriented Programing, September 1992

[PHR90] J. F. Patterson, R. D. Hill, S. L. Rohall, „Rendezvous: An Architecture for Synchronous Multi-User Applications“, CSCW 90 Proceedings, October 1990

[RG92] M. Roseman, S. Greenberg, „GROUPKIT — A Groupware Toolkit for Building Real-Time Conferencing Applications“, CSCW 92 Proceedings, November 1992

[Rot] Prof. K. Rothermel, Script zu Vorlesung *Verteilte Systeme* an der Uni Stuttgart, SS 1993

[Sem93] F. Sembach, „Gedanken zur Architektur des Shared Workspace“, internes Arbeitspapier der Abteilung Verteilte Systeme, Fakultät Informatik an der Universität Stuttgart, Juni 1993

[SM93] A. Santos, A. Marcos „ An Algorithm and Architecture to Support Cooperative Multimedia Editing“, 4th Workshop on Future Trends in Distributed Systems, 1993

[SR93] F. Sembach, K. Rothermel, „ *TEATIME*: Gemeinsamer Arbeitsbereich für kooperativ bearbeitete multimediale Objekte “. In Wolfgang Effelsberg, Kurt Rothermel (Hrsg.), *Verteilte Multimedia-Systeme*. GI/ITG, 1993

[ZSa] W. M. Zhang, M. Straßer, „Dokumentation zum SWS“

[ZSb] W. M. Zhang, M. Straßer, „Anwenderhandbuch zum SWS“

[Wir93] S. Wirag, „Entwicklung eines objektorientierten kooperativen Graphikeditors“, Studienarbeit Nr. 1225, Fakultät Informatik, Universität Stuttgart, 1993

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

(Markus Straßer)