Nameserver- und Autorisierungsprotokolle für verteilte Systeme in Oberon

Martin Hasch
Fakultät für Mathematik und
Wirtschaftswissenschaften
Universität Ulm

Juli 1996

Inhaltsverzeichnis

Einleitung

Verteilte Systeme gewinnen heute zunehmend an Bedeutung. Im Gegensatz zum zentralistisch organisierten System kann ein verteiltes System unabhängig von Begrenzungen durch Hardware skaliert werden; es ermöglicht höhere Effizienz, weil verschiedene Ressourcen — insbesondere auch Rechenzeit — tatsächlich parallel genutzt werden können; ein redundant ausgelegtes verteiltes System ist besonders zuverlässig, weil der Ausfall einzelner Komponenten nicht zum Ausfall des Systems führt; und nicht zuletzt verspricht die Untersuchung verteilter Systeme neue Erkenntnisse sowohl über die Modellierung komplexer, „verteilter“ Vorgänge als auch darüber, wie sich heterogene Bestandteile gleichsam „organisch“ zu einem funktionalen Ganzen verbinden lassen.

Die besondere Stärke des verteilten Systems, seine nahezu unbeschränkte Komplexität, würde zum Problem, wenn sie sich direkt auf die Komplexität der einzelnen Module auswirken würde. Besondere Aufmerksamkeit sollte daher bei der Entwicklung verteilter Systeme darauf gerichtet werden, daß der für sie spezifische Teil der Funktionalität für den übrigen Kontext so transparent wie möglich gehalten wird.

Die vorliegende Arbeit beschäftigt sich mit den in einem verteilten System notwendigen Mechanismen zur Lokalisierung fremder, d.h. nicht a priori bekannter Objekte und zur Verwaltung dezentral zu vergebender Zugriffsberechtigungen.

Die Darstellung gliedert sich in vier Kapitel. Das erste Kapitel geht in knapper Form auf das Ulmer Oberon-System ein, also auf den Kontext, in dem die hier behandelten Konzepte entwickelt und praktisch realisiert wurden. Das zweite Kapitel zeigt eine Abstraktion für Autorisierungsprotokolle und damit gleichzeitig ein ausführlicheres Beispiel für objektorientierte Programmierung in diesem System. Im dritten Kapitel wird das zentrale Thema der Arbeit entfaltet: eine Abstraktion für Namensräume für Objekte. Das letzte Kapitel schließlich demonstriert an praktischen Beispielen, wie verteilte Systeme die genannten Abstraktionen nutzen können.

Kapitel 1

Das Ulmer Oberon-System

1.1 Form und Sprache

Das Ulmer Oberon-System stellt die verschiedensten Programmierkonzepte in Form einer erweiterbaren Bibliothek zur Verfügung. Ein großer Teil dieser Konzepte wurde nicht eigens in diesem Rahmen erfunden, sondern gehört sicherlich zu den Standardwerkzeugen des modernen Informatikers. Neu und richtungsweisend ist dagegen, wie konsequent die verschiedenen Paradigmen weg von der globalen Integration in den Tiefen eines Systems hin zu separaten, auf Programmierniveau zugänglichen Einzelbestandteilen an der Oberfläche verlagert wurden.

Die Bibliothek stellt sich dar als eine Sammlung von Modulen, die in der Programmiersprache Oberon (wie in [Wirth88] und ergänzend [Borchert95a] beschrieben) formuliert wurden. In der Beschränkung, gewissermaßen dem Fortlassen jeglicher Komfortausstattung — samt zahlreicher serienmäßig eingebauter Ungereimtheiten — erweist dieses Sprachkonzept seine eigentliche Stärke. Die verschiedenen Aspekte einer Problemlösung zeigen sich vermehrt in expliziter Form und sind dadurch leichter voneinander zu differenzieren, was von Vorteil ist: Verborgene Querbeziehungen zu vermeiden — auch auf Kosten der Kompaktheit des Quellcodes —, kann ja als einer der Schlüssel zur Softwarequalität angesehen werden (vgl. [Borchert94, Kap. 9]). Daß die vielen theoretisch zur Verfügung stehenden Mittel andererseits gerade *nicht* in Erscheinung treten, wo sie nicht auch eingesetzt werden, ist von mindestens ebenso unschätzbarem Wert für die Lesbarkeit von Programmen. Dies steht im Gegensatz zur Situation bei mächtigeren Sprachen, wo nicht ohne weiteres klar ist, ob ein Codefragment auf „elegante“ Art sehr viel Komplexität versteckt oder es tatsächlich einen bis zur Einfachheit optimierten Sachverhalt wiedergibt.

Die Bibliothek enthält also viele Elemente, die bisher wohl kaum in ähnlich umfassender Weise ausschließlich im Rahmen einer portablen Laufzeitbibliothek realisiert worden sein dürften. Dazu gehören insbesondere Techniken zur Ausnahmenbehandlung, Adreßraumverwaltung, automatischen Speicherrückgewinnung, Verteilung des Kontrollflusses, Synchronisierung, plattformunabhängigen Datenweitergabe und Persistenz, sowie verschiedene objektorientierte Techniken einschließlich der Zugriffsmöglichkeit auf entfernte Objekte und weiterer Hilfsmittel für verteilte Systeme, die zum Teil Gegenstand der folgenden Kapitel sind. Hier kann nur einführend auf einige wesentliche Eigenschaften des umfangreichen Projekts „Ulmer Oberon-System“ eingegangen werden.

1.2 Systemumgebung

Ein Thema, das von der allgemeinen Sprachbeschreibung nicht abgedeckt wird, ist die Frage nach der Laufzeit-Umgebung, in der der Programmcode konkret ausgeführt wird. Der Begriff des Programms als in sich abgeschlossener Einheit mit definiertem Einstiegspunkt, Ablauf und letztendlicher Termination wird in Oberon ja gar nicht angesprochen oder vorausgesetzt. Die Antworten können daher unterschiedlich ausfallen.

Das Züricher Oberon-System beispielsweise ist bzw. emuliert ein Ein-Benutzer-Betriebssystem, das keine separaten Prozesse kennt, sondern neue Module einfach zur Laufzeit in den einen Adreßraum hinzulädt und initialisiert [Wirth92]. Nach dem Laden sind bestimmte, *commands* genannte Prozeduren von der Benutzeroberfläche aus erreichbar und erfüllen so die Rolle, die dem herkömmlichen Programmbegriff vielleicht am nächsten kommt.

Demgegenüber kommt das Ulmer System ohne eine residente Umgebung aus. Übersetzte Module werden wie normale Programme in anderen Sprachen zusammen mit den benötigten Bibliotheken zu einer ausführbaren Datei (*executable image*) gebunden und können im gastgebenden Betriebssystem (etwa UNIX[1]) als eigener Prozeß gestartet werden. Ein solcher Prozeß könnte sich wie das Züricher System verhalten und über *commands* und eine grafische Benutzeroberfläche mit dem Anwender kommunizieren, kann aber genausogut eine einzelne Aufgabe erledigen und danach terminieren, oder er bleibt als ein neuer Netzwerkdienst im Hintergrund und wartet auf Aufträge.

Damit auch hier kein ausgewiesenes Hauptmodul erforderlich ist, werden zum Programmstart einfach alle anfangs bekannten Module (genauer: eine beim Binden festgelegte Menge von Modulen) nacheinander initialisiert, wobei in bezug auf die Reihenfolge selbstverständlich die Importhierarchie eingehalten wird. Nach diesen Initialisierungen ist das Programm bereits beendet, es sei denn, neue Tasks[2] sind entstanden und noch aktiv.

[1] UNIX ist ein eingetragenes Warenzeichen der UNIX System Laboratories, Inc.

[2] Näheres zum Begriff der Task folgt im Abschnitt 1.3.4

Innerhalb eines einzigen UNIX-Prozesses können selbstverständlich mehrere Oberon-Tasks ablaufen, und je nach Implementierung ist auch dynamisches Nachladen von Modulen möglich. Insofern verbindet das Ulmer System die Möglichkeiten einer Oberon-typischen mit einer an eigenständigen Programmen orientierten Sichtweise, die beim Einsatz als „Allzweck"-Programmierumgebung sicherlich in vielen Fällen angemessen ist.

1.3 Aufbau der Bibliothek

1.3.1 Bestandteile

Die Basis der Bibliothek wird von Modulen gebildet, die allgemeine Abstraktionen darstellen. Eine Abstraktion wird modelliert als eine Sammlung von einem oder mehreren Datentypen (Objektklassen) sowie Operationen auf denselben (Methoden). Um abgeleitete Objektklassen mit jeweils eigenen Implementierungen der zugehörigen Methoden zu unterstützen, enthalten die Basismethoden in der Regel einen indirekten Aufruf der implementierenden Methode, die für jedes Objekt erst mit seiner Initialisierung festgelegt wird. Diese auf *upcalls*, d.h. Prozeduraufrufe entgegen der Importhierarchie fußende Technik unterscheidet sich von Ansätzen in einigen klassischen objektorientierten Sprachen, ist aber sehr zweckmäßig. Insbesondere schafft sie die Voraussetzung für die durchgehende Trennung von Implementierungen und Abstraktionen, sodaß z.B. systemnahe Bausteine den entsprechenden Abstraktionen nicht über- sondern untergeordnet werden können, was der Portabilität aller anderen Programmteile ungemein zugute kommt. Eine ausführliche Diskussion dazu bietet [Borchert94, Kapitel 2].

Zu den allgemeinsten Abstraktionen gesellen sich systemunabhängige Implementierungen und Erweiterungen; auf darüberliegenden Ebenen sind unter anderem Schnittstellen zum Betriebssystem und schließlich die systemspezifischen, auf diese Schnittstellen aufbauenden Implementierungen zu finden.

1.3.2 Importbeziehungen

Die unterschiedlichen Beziehungen zwischen Modulen der Bibliothek sollen im folgenden durch ein Beispiel veranschaulicht werden. Betrachtet wird ein (fiktives) Modul *MyModule*, das eine Prozedur *ShowTime* zur Ausgabe der momentanen Uhrzeit auf die Standardausgabe zur Verfügung stellen soll. Wie diese Prozedur realisiert werden könnte, zeigt Abbildung 1.1.

Abbildung 1.2 zeigt die wichtigsten daran beteiligten Module und ihre Importbeziehungen:

- *Times* ist eine allgemeine Abstraktion für Zeitskalen.

```
IMPORT
   Clocks, Dates, Times, Write;

PROCEDURE ShowTime;
   VAR
      now: Times.Time;
      exp: Dates.InfoRec;
BEGIN
   Clocks.GetTime(Clocks.system, now);
   Dates.Get(now, exp);
   Write.Int(exp.minute, 0); Write.String(" minutes past ");
   Write.Int(exp.hour, 0); Write.Ln;
END ShowTime;
```

Abbildung 1.1: Realisierung der Prozedur *ShowTime*

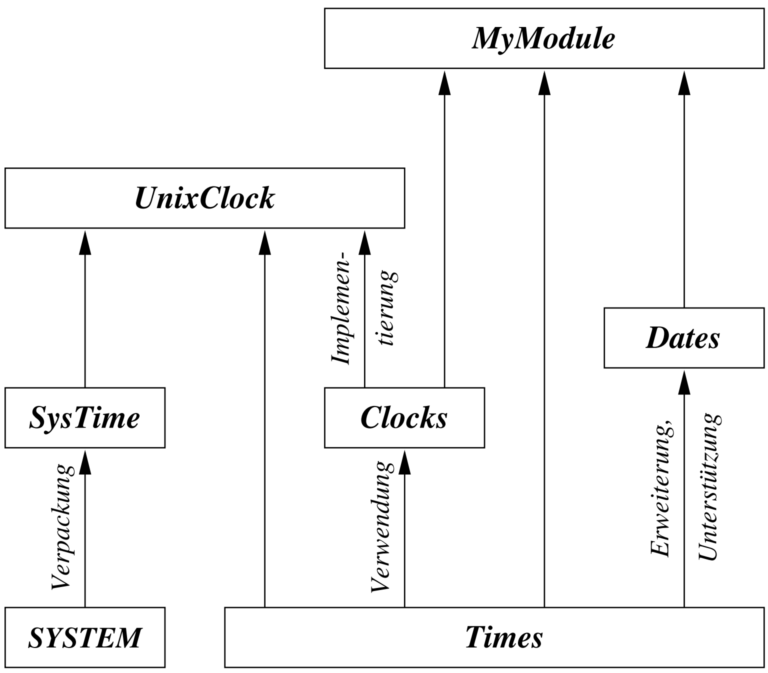

Abbildung 1.2: Vier verschiedene Arten von Importbeziehungen

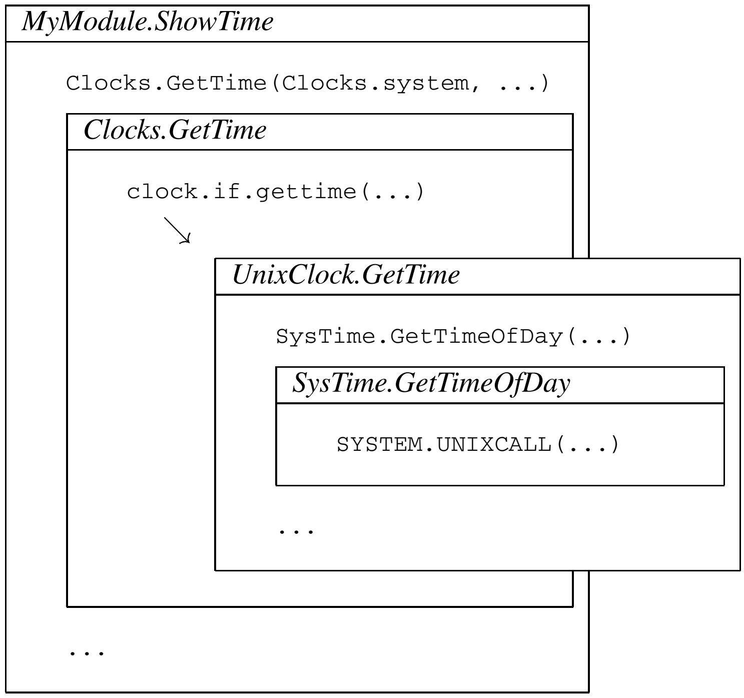

Abbildung 1.3: Prozeduraufrufkette mit *upcall*

- *Dates* erweitert diese Abstraktion um Methoden für kalendarische Berechnungen.
- *Clocks* ist eine Abstraktion für Uhren, welche die Abstraktion für Zeitskalen benutzt.
- *UnixClock* ist eine systemnahe — d.h. von nicht portablen Einrichtungen Gebrauch machende — Implementierung einer Uhr.
- *SysTime* stellt eine Schnittstelle zu den Zeitmessungsfunktionen des Betriebssystems dar. Aus Portabilitätsgründen sind solche Schnittstellen den direkten Systemaufrufen aus dem übersetzerinternen Modul SYSTEM vorzuziehen.

Das anwendende Modul *MyModule* benützt nur systemunabhängige Komponenten und bleibt somit portabel. Den Zugriff auf die eigentlich benötigte systemabhängig implementierte Funktion erhält es in diesem Fall über die Variable *Clocks.system*. Zur Verdeutlichung des Instruktionenflusses zeichnet Abbildung 1.3 einen Teil der Aufrufkette nach, die für die Bestimmung der Uhrzeit durchlaufen wird.

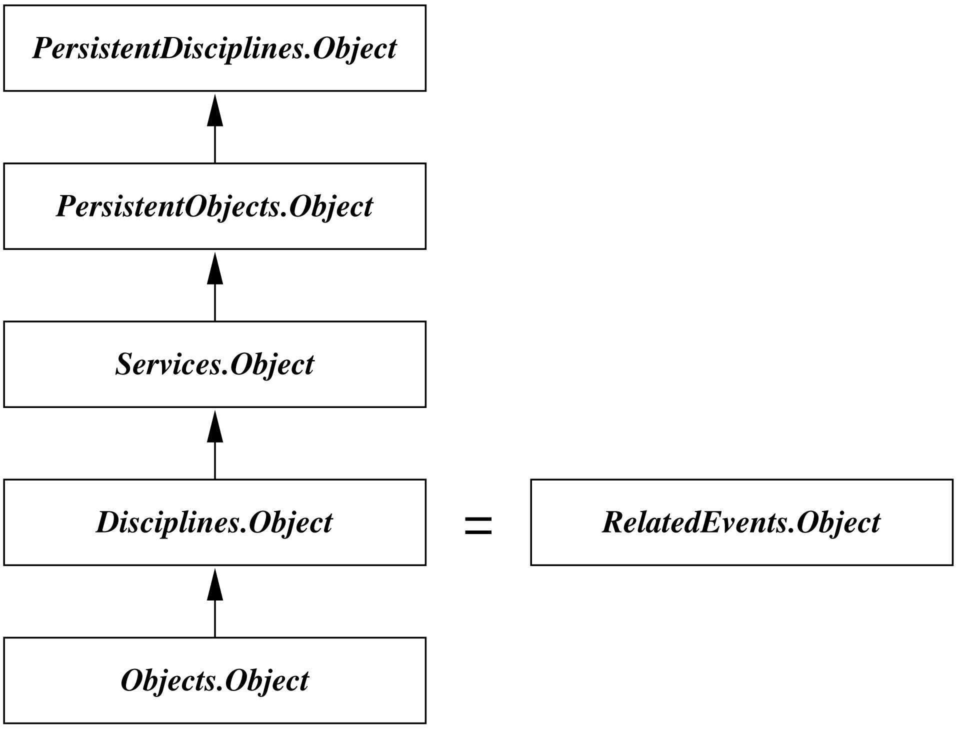

Abbildung 1.4: Grundlegende Objekttypen in Oberon

1.3.3 Typensystem

Eine wichtige Funktion einer Programmiersprache und im gleichen Sinne auch der sie ergänzenden Bibliotheken ist es, Softwarekomponenten so leicht wie möglich auf Korrektheit und Vorgabentreue überprüfbar zu machen. Dazu leistet ein gut durchdachtes System differenzierter Datentypen einen wertvollen Beitrag. Strenge Kompatibilitätsregeln, die mittels statischer und dynamischer Kontrollen durchgesetzt werden, erzwingen die Einhaltung bestimmter Bezüge und helfen zugleich, Fehler frühzeitig zu erkennen oder gar nicht erst zu machen. Die richtige Wahl von Verallgemeinerungen und Vereinheitlichungen erleichtert andererseits in vielen Fällen die Wiederverwendbarkeit und universelle Einsetzbarkeit einmal entwickelter Teile. Beide Gesichtspunkte kommen in Oberon sehr stark zum Tragen.

In der hier beschriebenen Bibliothek bilden alle Verbundtypen eine gemeinsame Hierarchie. Einige elementare Eigenschaften eines Objekts können bereits aus seiner statischen Typinformation entnommen werden. Abbildung 1.4 zeigt die wichtigsten dieser Typen. Im einzelnen repräsentieren sie die folgenden Abstraktionen:

- *Objects* stellt als gemeinsame Basis für alle anderen Verbundtypen das „eigenschaftslose“ Objekt zur Verfügung. Der Typ *Objects.Object* kann daher als Referenz auf etwas (nahezu) beliebiges dienen.

- *Disciplines* schafft die Möglichkeit, zusätzliche private Datenstrukturen an fremde Objekte anzuhängen. Diese Technik bildet die Grundlage für eines der Designprinzipien der Bibliothek, daß nämlich Objekte als gemeinsame Bezugspunkte für unterschiedliche Sichten verstanden werden, die keiner zentralen Kontrolle unterliegen. Daher sind Datentypen, die exportiert (d.h. anderen Modulen zugänglich gemacht) werden, in der Regel mindestens Erweiterungen von *Disciplines.Object*.

- *RelatedEvents* erfüllt die besonders für die Ausnahmenbehandlung nützliche Aufgabe, Ereignisse[3] Objekten zuzuordnen und dadurch einer kontextbezogenen Behandlung zugänglich zu machen. In Frage kommen dafür alle Objekte, die Sichten unterstützen. Oft dient eine Variable oder ein Parameter in einem bestimmten Zusammenhang ausschließlich dem Zweck, Ereignisse zu sammeln und weiterzuleiten. Dies wird am einfachsten dokumentiert, indem man den zu *Disciplines.Object* äquivalenten Typ *RelatedEvents.Object* dafür verwendet.

- In Oberon ist der Datentyp eines Objekts selbst kein Objekt erster Klasse (*first class object*), d.h. um Operationen auf Datentypen zuzulassen, wie sie für gewisse objektorientierte Techniken[4] gebräuchlich sind, bedarf es der Simulation eines solchen Typensystems. Dies (und noch mehr) leistet das Modul *Services*. Hier werden Typen als eigenständige Objekte eingeführt, die sowohl von den durch sie charakterisierten Originalobjekten abgeleitet als auch eins zu eins auf Typnamen in Form von Zeichenketten abgebildet werden können.

  Objekte, die als Erweiterung von *Services.Object* definiert werden, erlauben zudem das Zusammenwirken verschiedener orthogonaler, d.h. unabhängig voneinander und von einem bestimmten Typ formulierbarer Abstraktionen, da diese Objekte einen Henkel besitzen, um die notwendigen typbezogenen Implementierungen nachträglich anzukoppeln. Die Module, die diese Dienstleistungen (*services*) „überraschend“ anbieten, werden mit Hilfe des *Services*-Registriersystems oder gar unter Mitwirkung eines besonders darauf eingerichteten dynamischen Laders aufgespürt.

- *PersistentObjects* ist eine Abstraktion, die Objekte persistent, d.h. von ihrem ursprünglichen Adreßraum unabhängig macht. Dazu dienen zwei Methoden, mit denen ein Objekt in eine Folge von Bytes und wieder zurück verwandelt werden kann. Jede Erweiterung von *PersistentObjects.Object* steuert die zu ihrem Teil der Datenstruktur gehörenden Schreib- und Leseoperationen (*marshalling procedures*) dazu bei; das Basismodul selbst bedient sich des

[3] Näheres zum Ereignisbegriff siehe im folgenden Abschnitt 1.3.4.

[4] Siehe dazu etwa [Borchert95b, Services(3)].

Typensystems von *Services*, um Typnamen und Typen einander zuzuordnen und die Erweiterungshierarchie zu durchlaufen bzw. zu rekonstruieren.

Was damit effektiv erreicht wird, ist, daß byte-orientierte Kanäle (*streams*) Objekte transportieren können. Da solche Kanäle in gleicher Weise innerhalb eines Prozesses, als Verbindung eines Prozesses mit einem Speichermedium oder zur Kommunikation verschiedener Prozesse untereinander eingesetzt werden können, wird mit dieser einen Abstraktion eine sehr allgemeine Form der Persistenz verwirklicht, die sowohl den zeitlichen als auch den räumlichen Aspekt des Begriffes umfaßt.

- Die zu weiteren Sichten auf ein Objekt gehörenden Datenstrukturen (Disziplinen) haben oftmals einen kontextbezogenen, temporären Charakter, sollen also beim Verschieben des Objekts nicht erhalten bleiben. Nichtsdestotrotz rechtfertigt, ja verlangt das Sichtenmodell auch einen Mechanismus, zusätzliche Datenstrukturen persistent an persistente Objekte anzubinden. *PersistentDisciplines* stellt einen solchen Mechanismus zur Verfügung, indem es zum einen die *Disciplines*-Funktionalität für eine eigene Variante von Disziplinen und zum anderen einen persistenten Objekttyp, der beim Schreiben und Lesen diese Disziplinen mitabbildet, bereitstellt.

  Interessanterweise können persistente Disziplinen nicht nur mit diesen speziellen Objekten vom Typ *PersistentDisciplines.Object*, sondern auch mit nicht persistenten Objekten verbunden werden, und zwar aus einem einleuchtenden Grund: Von der Persistenz, also Transportfähigkeit einer Disziplin profitiert ja auch ein stationäres Objekt, wenn etwa das Anbinden und Abfragen selbst an einen Transportvorgang geknüpft ist. Dieser Fall ist für verteilte Systeme durchaus typisch, wie es sich weiter unten noch zeigen wird.

1.3.4 Kontrollfluß

In diesem Abschnitt sollen wichtige Abstraktionen, die den Kontrollfluß betreffen, zusammengefaßt werden, namentlich die schon erwähnten Ereignisse, Tasks und Synchronisationsmechanismen.

Ereignisse

Unter Ereignissen versteht man Träger von Informationen, die jederzeit anfallen können und ohne direkte Rückantwort an verschiedene Interessenten weiterzuleiten sind. Das Modul *Events* definiert die entsprechende Basisabstraktion. Ein Ereignis erreicht nicht unbedingt sofort nach seiner Entstehung die an dieser Sorte von Ereignissen interessierten Parteien, sondern kann, abhängig von seiner Priorität, zunächst auch in eine Warteschlange geraten, wo es so lange verzögert wird, bis die entsprechende Priorität wieder akzeptiert wird und es (ggf. nach Bearbeitung seiner länger wartenden Vorgänger) schließlich an die Reihe kommt; ferner kann es, wie schon angedeutet, zur späteren Untersuchung bei einem Objekt abgespeichert werden. Als Priorität kann ein beliebiger Wert aus einer eindimensionalen, aber konfigurierbaren Skala vergeben werden.

Anstatt zwischengespeichert oder an einen oder mehrere Bearbeiter (*event handlers*) weitergegeben zu werden, können Ereignisse auch zum Programmabbruch führen oder schlicht ignoriert werden. Als Informationsträger für die Ausnahmenbehandlung eingesetzt, erlauben sie somit sehr differenzierte Reaktionsmöglichkeiten auf die verschiedensten Fehlerbedingungen. Auch Laufzeitfehler wie Verletzungen von Indexgrenzen, Dereferenzieren von Nullzeigern, Division durch Null usw. und asynchrone externe Ereignisse (etwa UNIX-Signale) werden im System durch Ereignisse repräsentiert.

Koroutinen

Im Ulmer Oberon-System bilden Koroutinen die Grundlage für Mechanismen, mit denen die Eingleisigkeit des Kontrollflusses aufgebrochen werden kann. Koroutinen haben jeweils ihren eigenen Anweisungszeiger (*program counter*) und Keller (*stack*); sie sind nicht wirklich gleichzeitig, sondern immer abwechselnd aktiv, und zwar jeweils solange, bis sie die Kontrolle explizit an eine andere Koroutine weitergeben. Dieses noch relativ einfache Sprachmittel genügt allerdings, um auch nichtdeterministische Formen der Nebenläufigkeit zu simulieren, also genau das Verhalten, das bei echter paralleler Ausführung gegeben wäre. Die dafür entwickelten Kooperationsmechanismen eignen sich dementsprechend für jede Art der Parallelität, auch derjenigen, die auf natürliche Weise bei Beteiligung mehrerer Rechner an einem verteilten System gegeben ist.

Tasks

Die konkurrierend lauffähigen Ablaufeinheiten innerhalb eines Oberon-Programms werden als Tasks bezeichnet, deren Eigenschaften von der gleichnamigen Abstraktion definiert werden. Technisch ist eine Task eine Koroutine (oder ein Verband einander auf herkömmliche Weise explizit abwechselnder Koroutinen), die nicht direkt sondern unter Kontrolle eines Ablaufkoordinators (*scheduler*) aktiviert wird. Sie kann vier verschiedene Zustände annehmen: bereit, beschäftigt, auf etwas wartend, beendet. Die einem Ablaufkoordinator unterstehenden Tasks bilden eine Taskgruppe; von diesen können auch mehrere existieren, deren Ablaufkoordinatoren wiederum Mitglieder einer weiteren Taskgruppe sind usw., bis hinauf zur immer vorhandenen ersten Koroutine, die implizit eine Taskgruppe für sich allein bildet.

Ist eine Task einmal angestoßen worden, d.h. „beschäftigt“, bleibt sie das solange, bis sie selbst ihrem Ablaufkoordinator wieder eine Gelegenheit, die Kontrolle zu übernehmen, einräumt (sei es direkt, sei es durch einen Ereignisbearbeiter). Normalerweise ergeben sich solche Gelegenheiten ganz von selbst, sobald gewartet werden muß; sie können aber auch aus Fairneßgründen zusätzlich geschaffen werden.

Bedingungen

Die kooperative Weise, auf etwas zu warten, ergibt sich aus der Verwendung der Abstraktion für Bedingungen (*conditions*). Solche Bedingungen können in beliebiger Weise miteinander verknüpft werden und sich auf alle denkbaren Gegebenheiten beziehen, vom Freiwerden einer nur beschränkt verfügbaren Ressource bis zum Eintreten eines bestimmten Ereignisses. Wenn eine Task auf etwas warten will, teilt sie über *Tasks.WaitFor* ihrem Ablaufkoordinator in Form einer Bedingung mit, worauf sie wartet. Dieser wird, wenn er noch rechenbereite Tasks zur Verfügung hat, die Kontrolle an eine von ihnen weitergeben, oder andernfalls selbst unter der Vereinigung aller Bedingungen in den Wartezustand gehen. Schließlich kommt es irgendwann dazu, daß auch auf der höchsten Ebene (in der Task der Ur-Koroutine) keine Bedingung erfüllt ist; dann wird auf die gewöhnliche Art gewartet, d.h. den Implementierungen der Bedingungen entsprechend eine effiziente Warteoperation ausgeführt. Diese kostet in der Regel selbst bei komplizierten Kombinationen praktisch keine Rechenzeit, sondern läuft auf eine blockierende Ein-/Ausgabeoperation oder das Suspendieren des Prozesses hinaus; eine Diskussion der Algorithmen, die diese Optimierung bewerkstelligen, würde hier allerdings zu weit führen.[5]

1.3.5 Aufträge und entfernte Objekte

Verteiltheit mit Transparenz

Der letzte Abschnitt dieser Einführung stellt einige der Elemente vor, die die Bibliothek zur Unterstützung verteilter Systeme anbietet.

Als entscheidendes Merkmal, das ein verteiltes System charakterisiert, gilt hier die Annahme, daß die Objekte, auf denen operiert wird, nicht nur im eigenen, sondern auch in fremden Prozessen lokalisiert sein können. Dabei wird erwartet, daß der Umgang mit diesen Objekten — der Aufruf ihrer Methoden — völlig unabhängig von deren Lokalisierung sein sollte, sodaß die vorgesehenen Anwendungen funktionieren, ohne daß auch nur das geringste Wissen über die Herkunft der betreffenden Objekte eingebracht werden müßte.

[5] [Borchert94, Kapitel 7] geht darauf ein.

Die Technik, mit der diese Einheitlichkeit erreicht wird, beruht auf Stellvertreter-Objekten und Delegation. Das bedeutet, für jedes bekannt gewordene fremde Objekt existiert ein lokales Stellvertreter-Objekt, das zu nichts anderem in der Lage ist, als die Aufträge, die es erhält, zum Originalobjekt weiterzuleiten und dessen Reaktion anschließend für die eigene auszugeben.

Netzwerke

Verbindungen zu anderen Prozessen abstrahiert das Modul *Networks*. Selbstverständlich existieren Implementierungen für reale vom Betriebssystem bereitgestellte Netzwerkschnittstellen, unter UNIX etwa *Internet* und *UnixDomainSockets*. Wenn vom Betriebssystem gemeinsame Speicherbereiche (*shared memory*) unterstützt werden, können Prozesse auf derselben Maschine auch über schnelle *SM-Streams* miteinander kommunizieren. All diesen Verbindungen ist gemeinsam, daß sie für den eigentlichen Datentransfer die bekannten byte-orientierten Kanäle (*streams*) verwenden. Für die Delegation von Aufträgen über ein Netzwerk müssen diese also in Byte-Folgen und wieder zurück verwandelt werden, kurz: persistent sein. Einen Basistyp für persistente Aufträge definiert das Modul *Messages*.

Typenspezifische Vorbereitungen

In der Regel reagieren Objekte nicht von vornherein nur auf Aufträge in einem einheitlichen Nachrichtenformat, sondern realisieren ihre wichtigen Methoden als direkte Prozeduraufrufe. Damit ein Objekt „verteilfähig“ wird, bedarf es also der Umsetzung dieser Prozeduraufrufe in persistente Nachrichten (auf der Seite mit dem Stellvertreter-Objekt) und dazu passender Nachrichtenbearbeiter (auf der Seite mit dem Originalobjekt). Dies zusammen mit den für die Persistenz der Nachrichten notwendigen Schreib- und Leseoperationen zu implementieren, ist nicht schwierig und kann mit einem Hilfsprogramm (*genrem*) sogar weitgehend automatisiert werden. Sollen zusätzliche Sichten oder Erweiterungen aus der Ferne erreichbar sein, können dafür weitere Transporteinrichtungen angekoppelt werden. Wohlgemerkt sind alle diese Vermittlerdienste mit keinerlei Veränderungen der original mit dem Objekt befaßten Module verbunden. Sie werden üblicherweise zu einem Modul *RemoteXXX*, wobei *XXX* für den Namen der ursprünglichen Abstraktion steht, zusammengefaßt und unter diesem Namen auch als Dienst registriert.

Das Modul *RemoteObjects*

Die Schaltzentrale, die diese Elemente so koordiniert, daß der Prozeßgrenzen überschreitende Zugang zu Objekten so einfach zu benutzen ist wie z.B. gewöhnliche

Schreib- und Leseoperationen, ist in der Basisabstraktion *RemoteObjects* verwirklicht. *RemoteObjects* verwertet allgemeine, von *Networks* definierte Adressen (*addresses*) und Einschaltpunkte (*sockets*), um Verbindungen mit anderen Prozessen herzustellen. Über die so erhaltenen Verbindungskanäle wickelt es die Nachrichtenübermittlung zwischen Objekten und Stellvertreter-Objekten ab.

Kanäle bleiben in der Implementierung von *RemoteObjects* verborgen. Sie werden bei Bedarf eröffnet und automatisch den betreffenden Objekten zugeordnet. Über einen Kanal können Verbindungen zu mehreren Objekten gleichzeitig aufrecht erhalten werden (*multiplexing*), und die Aufträge für ein bestimmtes Objekt können — je nach Maßgabe der das Objekt unterstützenden Dienste — sequentiell oder auch quasi-parallel, d.h. ohne Warten auf die Erledigung des jeweils vorhergehenden Auftrags, empfangen werden. Die letztere Unterscheidung dient zur Feinabstimmung der Auftragsentgegennahme auf das zu erwartende Antwortverhalten, also der Verbesserung der Effizienz, ändert aber nichts an der Tatsache, daß ein von außen sichtbares Objekt jederzeit mit neuen Aufträgen rechnen muß, auch während es z.B. einen lokalen Aufruf ausführt. Kritische Regionen, die nur sequentiell durchlaufen werden dürfen, sollten sich daher immer selbst schützen, etwa mit Semaphoren und, falls auch Ereignisbearbeiter eine Bedrohung darstellen, erhöhter Priorität.

Das Modul *RemoteObjects* hält intern für jedes exportierte und für die Dauer der Verbindung auch für jedes importierte Objekt eine Identität aufrecht, was die erfreulichen Konsequenzen hat, daß Zeiger auf eigene und fremde Objekte gleichermaßen eindeutig sind, Mehrfachimporte auch bei verschiedenen Verbreitungswegen nicht zur Vervielfältigung der Datenstrukturen führen und re-importierte eigene Objekte ohne zusätzlichen Verwaltungsaufwand als das, was sie sind, behandelt werden.

Eine weitere Aufgabe dieser Abstraktion ist die Unterstützung persistenter Disziplinen, für die sie ja ein regelrechtes Paradebeispiel darstellt. Eigene Sichten, die auf der Basis von *PersistentDisciplines* realisiert werden, gelten also ohne irgendein weiteres Zutun, wenn sie sich zufällig auf ein Stellvertreter-Objekt beziehen sollten, dem Originalobjekt.

Das Exportieren (für die Außenwelt Sichtbarmachen) eines Objekts hat analog zum Versenden eines persistenten Objekts die Gestalt einer Schreiboperation. Sie führt implizit, falls noch nicht vorhanden, zum Aufbau der nötigen Infrastruktur, damit das Objekt Nachrichten empfangen kann, und zur Vergabe einer Adresse. Geschrieben wird dann diese Adresse, oder anders ausgedrückt: eine persistente Form der Information darüber, wie das Objekt zu erreichen ist. Das Importieren eines Objekts besteht dementsprechend aus dem Lesen dieser Information und, wenn nötig, der Erzeugung eines Stellvertreter-Objekts sowie dem Versuch, einen Kanal zum Originalobjekt zu eröffnen. Die durch den Export hergestellte Sichtbarkeit eines Objekts kann explizit widerrufen werden mit *RemoteObjects.Withdraw* oder implizit, indem pauschal alle Verbindungen gekappt werden, etwa im Zuge der Aufräumarbeiten vor der Beendigung des Prozesses.

Illustration

Abbildung 1.5 zeigt schematisch, wie ein Auftrag an ein entferntes Objekt weitergeleitet wird. Mit *RemoteXXX* wurde das Modul bezeichnet, das den Dienst *RemoteObjects* für die Abstraktion realisiert. Für die Seite des Klienten (Importeurs) definiert es eine Implementierung, die Methoden in Aufträge verwandelt. Für den Server (Exporteur) hingegen installiert es einen Auftragsbearbeiter, der diese Aufträge ausführt, indem er die entsprechenden Methoden aufruft. Jede in *RemoteXXX* implementierte Methode erzeugt einen Auftragsverbund, der die Prozedurparameter, Rückgabewerte und eine Komponente *errors* zum Sammeln von Fehlerereignissen enthält. Dieser Verbund ist insgesamt zweimal im Netz unterwegs (zum Server und zurück), wobei er natürlich erst auf dem Rückweg die Information der Resultate und/oder Fehlerereignisse trägt.

Dienste für spezielle Abstraktionen

Für eine Reihe von Abstraktionen der Oberon-Bibliothek gibt es Module, die die für entfernte Zugriffe nötigen Dienste implementieren. Teilweise sind dazu auch weitere Techniken notwendig, auf die hier nicht näher eingegangen werden kann. Erwähnenswert ist vor allem *RemoteEvents*, das sogar Ereignisse über Prozeßgrenzen hinweg verteilbar macht. Im dritten Kapitel kommen wir auf entfernte Objekte zurück.

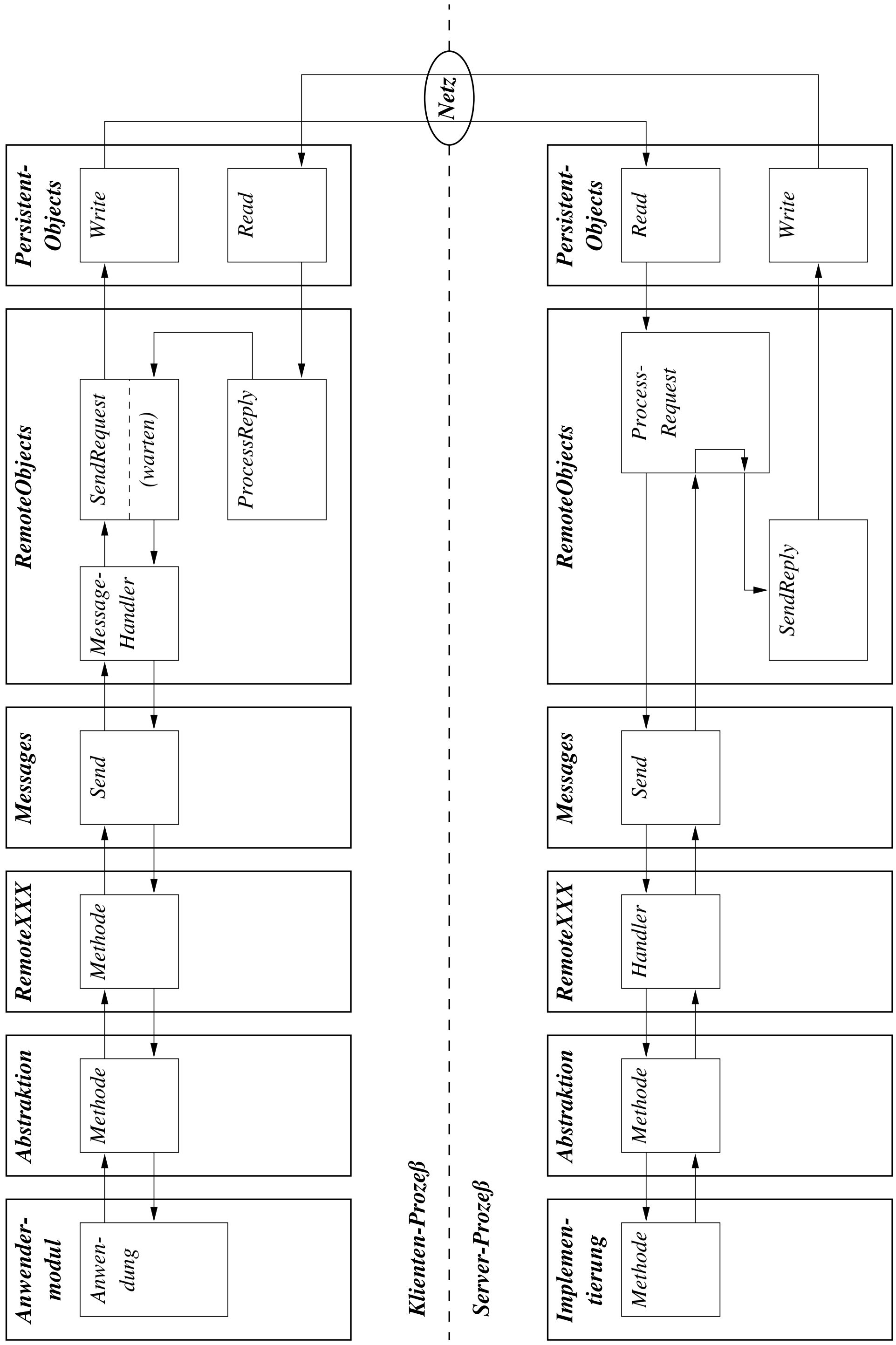

Abbildung 1.5: Kontrollfluß beim Zugriff auf ein entferntes Objekt (schematisch)

Kapitel 2

Autorisierungsprotokolle

2.1 Konzeption

Protokolle

Technisch gesehen stellt eine Sammlung von Datentypen und darauf operierenden Methoden nichts anderes dar als ein Protokoll. Abstraktionen werden also in Oberon als Protokolle repräsentiert. Ein spezielles Protokoll, das in diesem Kapitel näher beleuchtet werden soll, wurde für allgemein verwendbare Autorisierungsmechanismen entwickelt.

Autorisierung

Im täglichen Leben tritt die Problematik der Autorisierung sehr häufig auf. Existierende Lösungen folgen denselben Prinzipien, die auch von einer Abstraktion abgedeckt werden sollten.

- Schlösser und Schlüssel gewährleisten, daß Fahrzeuge, Räumlichkeiten und die verschiedensten „abschließbaren“ Einrichtungen nur von bestimmten Personen benutzt werden;
- Ausweise, Pässe, etc. belegen die Identität einer Person und ihre Zugehörigkeit zu bestimmten Personengruppen;
- Fahrkarten, Eintrittskarten, Gutscheine und dergleichen stellen (ohne nähere Identifikation des Nutzers) sicher, daß gewisse Leistungen nur in einer zuvor vereinbarten Quantität genutzt werden;
- Standesämter, Grundbuchämter, Notare usw. treten als neutrale Instanzen auf, die bestimmte Vereinbarungen dokumentieren und später auf Anfrage verifizieren können.

Aus all diesen Beispielen lassen sich zwei wesentliche Elemente für ein abstraktes Autorisierungsmodell erkennen:

1. Autorisierung ist mit einem Kontrollvorgang verbunden: Eine Partei, die eine Berechtigung in Anspruch nehmen will, muß sich einer Überprüfung stellen, aufgrund der diese Berechtigung entweder bestätigt oder widerlegt wird.

2. Solche Überprüfungen werden zweckmäßig mit eigens dafür vorgesehenen Informationsträgern (Schlüsseln, Bescheinigungen usw.) abgewickelt, die den Kontrollvorgang (hoffentlich) genügend sicher, aber dennoch einfach vollziehbar machen.

Das Modell

Mit diesen Bemerkungen ist das Modell bereits weitgehend charakterisiert. Ein erster Objekttyp (*Shards.Pot*) trägt die Information, die der kontrollierenden Instanz zur Verfügung steht; ein zweiter (*Shards.Lid*) stellt das Beweismittel dar, das die um Anerkennung ihrer Rechte ersuchende Partei vorlegen muß; und der Prüfvorgang selbst erhält schließlich die Form einer grundlegenden Methode (*Shards.Fits*) mit binärem Resultat.

Die Namen der beiden Objekttypen wurden durch das bekannte Sprichwort, *„for every pot there is a lid“* (etwa: „jedes Töpfchen findet sein Deckelchen“), motiviert. Der Modulname *Shards* (Scherben) erlaubt neben der naheliegenden Assoziation zur Töpferei noch eine weitere, und zwar zur vielleicht ältesten Methode einer an den Besitz eines Gegenstandes gekoppelten Autorisierung. Diese könnte darin bestanden haben, die Bruchstücke einer entzwei gebrochenen Tonscherbe, die die Vertragspartner separat aufzubewahren hatten, prüfend aneinander zu halten.

2.2 Realisierung

Persistenz

Eine Frage, die bei der Formulierung einer neuen Abstraktion früh geklärt werden sollte, ist, auf welche andere etwa schon vorhandene Abstraktion sie aufbauen kann. Eine solche übergeordnete Abstraktion gibt es immer, denn laut Konvention bilden ja alle Verbundtypen statisch eine gemeinsame Hierarchie, sind also mindestens Erweiterungen von *Objects.Object*.

Im Falle der *Shards* sollen die Objekte allerdings wesentlich speziellerer Natur sein, nämlich uneingeschränkt transportabel. Daher verlangen wir von vornherein

Persistenz. *Shards.Pot* und *Shards.Lid* müssen also als Erweiterungen von *PersistentDisciplines.Object* in die Typenhierarchie eingeführt werden[1], und korrekte Implementierungen benötigen die obligatorischen Methoden (*marshalling procedures*) für *PersistentObjects*.

Trivialfälle

Wie bereits angedeutet, werden Autorisierungsentscheidungen als Methoden der *Shards.Pot*-Objekte ausgeführt. Die denkbar einfachste Möglichkeit für eine solche „Entscheidung" ist die stereotype, vom präsentierten Gegenstück also völlig unabhängige Antwort. Es liegt daher nahe, einen stets ablehnenden und einen stets annehmenden Kontrolleur als triviale Implementierungen in das Basismodul *Shards* aufzunehmen.

Zu beachten ist, daß es umgekehrt einen Schlüssel, der überall bzw. nirgends paßt, nicht geben kann. Ein beliebiges unbekanntes *Shards.Lid*-Objekt wird zwar von den meisten Kontrolleuren zurückgewiesen werden, aber es gibt keine prinzipielle Möglichkeit, ein bestimmtes Verhalten zu erzwingen.

Konstruktoren

Jede Implementierung wird Möglichkeiten anbieten, Instanzen ihrer Objekte mit vorgegebenen Eigenschaften zu erzeugen. In vielen einfachen Fällen wird es genügen, lediglich „Töpfe" zur Verfügung zu stellen, die darauf eingerichtet sind, bei Bedarf passende „Deckel" nachzuliefern, wofür die optionale zweite Methode *Shards.Supply* vorgesehen ist. Diese Vorgehensweise ist aber nicht zwingend, weil sie schon aus konzeptionellen Gründen nicht immer möglich ist (man denke an den Topf, der überhaupt keinen Deckel akzeptiert). Auch ist es denkbar, daß zum Herstellen eines Deckels mehr Information benötigt wird, als der Topf allein enthält, oder daß in sonst einer Weise der Kontext eine Rolle spielt. Daher braucht *Supply* nicht implementiert zu werden und, falls doch, nicht unbedingt Erfolg zu garantieren.

Kombinationen

Eine Anwendung, die etwas schützen möchte, installiert einen Kontrolleur (*Pot*) und schaltet vor jeden Zugriff auf das geschützte Gut einen Aufruf der entsprechenden Methode (*Fits*). Nun erhebt sich die Frage, ob sich die Eigenschaften mehrerer Kontrolleure zu einem komplizierteren Prüfverfahren kombinieren lassen, etwa, um verschiedene Bedingungen alternativ zu akzeptieren (wie bei einer

[1] vgl. Abb. 1.4, Seite 12.

Schließanlage mit Hauptschlüsseln), oder um andererseits zur Erhöhung der Sicherheit mehrere Beschränkungen gleichzeitig einzufordern.

Um das Protokoll nicht zu verletzen, soll auch das zusammengesetzte Verfahren nur einen „Topf" und einen „Deckel" als Bezugspunkte benutzen. Im ersten Fall ist dies sehr leicht möglich. Soll ein Topf mehrere verschiedene Deckel akzeptieren, braucht er nur seinerseits mehrere einfache Töpfe enthalten und deren Kontrollmethoden mit „oder" zu verknüpfen.

Der zweite Fall ist kritischer. Ein „Supertopf", der mehrere Kontrollen mit „und" verknüpft, wird vermutlich jeden gewöhnlichen Deckel zurückweisen. Um so verfahren zu können, bedarf es also auch eines „Superdeckels", der gleichzeitig von verschiedenen Töpfen akzeptiert wird. Ein Deckel hat als an diesem Verfahren nur passiv Beteiligter allerdings nicht die Möglichkeit, zu bestimmen, welche Methode auf ihn anzuwenden ist. Dieses Problem läßt sich auf der Ebene der Abstraktion lösen, wenn sie zusammengesetzte Deckel intern so verwaltet, daß externe Implementierungen davon gar nicht behelligt werden. Eine implementierende Methode wird also, auch wenn ein „Superdeckel" getestet wird, implizit nur mit „Elementardeckeln" konfrontiert, dafür aber unter Umständen mehrmals.

Es bleibt festzuhalten, daß die Möglichkeit, Deckel zu kombinieren (mit Hilfe der Funktion *Shards.CombineLids*) genügt, um jede Art von „und"- und „oder"-Verknüpfungen sowohl von Töpfen als auch Deckeln innerhalb des einfachen Protokolls zuzulassen.

Programmierung

Abbildung 2.1 zeigt den gesamten Definitionsteil des Moduls *Shards* in Oberon. Zuerst werden die beiden Objekttypen *Pot* und *Lid* definiert; es folgen die Datentypen der Schnittstelle für Implementierungen (*interface*). Als Prozeduren treten natürlich die beiden Methoden *Fits* und *Supply* auf sowie die Initialisierungsroutine für neue Objekte, *Init*.

Drei Konstruktoren ebnen den Zugang zu den intern implementierten Objekttypen: *CombineLids* erzeugt einen „Superdeckel" aus zwei Komponenten — da diese wiederum zusammengesetzt sein können, lassen sich auch größere Konglomerate bilden —, *CreateSimplePot* liefert die beiden Sorten trivialer Töpfe und *CreateSomeLid* einen Deckel „ohne besondere Eigenschaften".

Zu bemerken ist, daß die wahren Typen der so erzeugten Objekte, die ja Erweiterungen der abstrakten Typen sind, gar nicht exportiert werden. Hinter dieser „Geheimniskrämerei" steckt das Bestreben, der Implementierung sogar in der Frage, ob eigene Datentypen verwendet werden, größtmögliche Freiheit zuzugestehen und dadurch Abhängigkeiten zu vermeiden.

```
DEFINITION Shards;

   IMPORT Objects, PersistentDisciplines;

   TYPE
      Pot = POINTER TO PotRec;
      PotRec = RECORD (PersistentDisciplines.ObjectRec) END;

      Lid = POINTER TO LidRec;
      LidRec = RECORD (PersistentDisciplines.ObjectRec) END;

      FitsProc = PROCEDURE (pot: Pot; lid: Lid): BOOLEAN;
      SupplyProc = PROCEDURE (pot: Pot; VAR lid: Lid): BOOLEAN;

      Interface = POINTER TO InterfaceRec;
      InterfaceRec =
         RECORD
            (Objects.ObjectRec)
            fits: FitsProc;
            supply: SupplyProc;                 (* may be NIL *)
         END;

   PROCEDURE Fits(pot: Pot; lid: Lid): BOOLEAN;

   PROCEDURE Supply(pot: Pot; VAR lid: Lid): BOOLEAN;

   PROCEDURE Init(pot: Pot; if: Interface);

   PROCEDURE CombineLids(VAR lid: Lid; lid1, lid2: Lid);

   PROCEDURE CreateSimplePot(VAR pot: Pot; permissive: BOOLEAN);

   PROCEDURE CreateSomeLid(VAR lid: Lid);

END Shards.
```

Abbildung 2.1: Definition des Moduls *Shards*

Abbildung 2.2 zeigt anhand der Methode *Shards.Fits*, wie eine typische Aufgabenverteilung zwischen einer Methode der Abstraktion und der zugehörigen implementierenden Methode aussehen kann:

- NIL ist als *Lid*-Argument zugelassen und steht dann für einen völlig unspezifischen Deckel. Die zum *Pot*-Objekt gehörende Implementierung der Methode braucht diesen Sonderfall aber nicht zu berücksichtigen, denn sie wird immer mit einem nichtleeren Argument aufgerufen.

- Die Behandlung kombinierter Deckel wird mittels einfacher Rekursion in Einzelabfragen aufgelöst.

```
TYPE
   SuperLid = POINTER TO SuperLidRec;
   SuperLidRec =
      RECORD
         (LidRec)
         lid1,
         lid2: Lid;
      END;

(* ... *)

PROCEDURE Fits(pot: Pot; lid: Lid): BOOLEAN;
   VAR
      result: BOOLEAN;
BEGIN
   IF pot.busy THEN
      RETURN FALSE            (* cyclic call? *)
   END;
   IF (lid # NIL) & (lid IS SuperLid) THEN
      WITH lid: SuperLid DO
         result := Fits(pot, lid.lid1) OR Fits(pot, lid.lid2);
      END;
   ELSE
      IF lid = NIL THEN
         lid := nilLid;
      END;
      pot.busy := TRUE;
      result := pot.if.fits(pot, lid);
      pot.busy := FALSE;
   END;
   RETURN result
END Fits;
```

Abbildung 2.2: Realisierung der Funktion *Shards.Fits*

- Die Boolesche Komponente *pot.busy* im privaten Teil des Verbundes *Shards.PotRec* bietet einen primitiven, aber wirksamen Schutz gegen Rückkopplungen.[2]

Eine externe Implementierung der Methode *Fits* benötigt noch einen Test, ob ihr *Lid*-Argument vom richtigen (eigenen) Typ ist, bevor sie dasselbe bearbeitet; im übrigen ist sie nur für die eigentliche Aufgabe, die gewünschte Kontrolle auf ihre Weise durchzuführen, verantwortlich.

Die Kontrolle selbst wird als elementarer Vorgang betrachtet, dessen Ergebnis nüchtern als Funktionsresultat zurückzumelden ist, ohne etwa den Ablehnungsfall mit großem Lärm als Ausnahmesituation zu behandeln.

[2] In der endgültigen Fassung des Moduls wird auf diesen Schutz verzichtet, da er die Parallelisierbarkeit beeinträchtigen würde.

```
PROCEDURE Fits(pot: Shards.Pot; lid: Shards.Lid): BOOLEAN;
BEGIN
   IF lid IS Lid THEN
      WITH lid: Lid DO
         WITH pot: Pot DO
            RETURN (lid.id = pot.id) & (lid.key = pot.key)
         END;
      END;
   ELSE
      RETURN FALSE
   END;
END Fits;
```

Abbildung 2.3: Realisierung der Funktion *PrivateShards.Fits*

2.3 Weitere Implementierungen von *Shards*

Standardlösung

Die trivialen Implementierungen, die das Basismodul anbietet, nützen offensichtlich keineswegs alle Möglichkeiten aus, die im Protokoll vorgesehen sind. Für eine sinnvolle — d.h. in erster Linie überhaupt stattfindende — Überprüfung der Zusammengehörigkeit bedarf es eines Topf- und Deckeltyps mit einem gewissen Maß an Individualität. *Shards*-Objekte mit dieser Eigenschaft implementiert das kleine Modul *PrivateShards*.

Ideal wäre eine unerschöpfliche Quelle immer neuer Topf-Individuen, von denen keine zwei denselben Deckel akzeptierten, und von Deckeln, die jeweils auf genau einen dieser Töpfe paßten. Ein auf einem Computer ablaufendes Programm, das selbst keine Identität und zudem nur eine begrenzte Menge von Unterscheidungsmerkmalen zur Verfügung hat, kann diese Idee natürlich nur ansatzweise verwirklichen. Immerhin läßt sich die Wahrscheinlichkeit ungewollter Übereinstimmungen mit etwas Heuristik auf einen für praktische Belange vertretbar kleinen Wert drücken: Eine Sequenz von Pseudozufallszahlen, die unter Einbeziehung von Variablen wie Datum und Uhrzeit, Prozeßnummer, Netzwerkadresse, Hardware-Seriennummern usw. initialisiert wird, hat nur ein geringes Risiko, systematisch in verschiedenen Programmläufen übereinzustimmen; eine ebenso „zufällig“ initialisierte, dann aber fortlaufende Nummer für Objektinstanzen sorgt zudem dafür, daß jedenfalls innerhalb eines Prozesses so rasch keine Wiederholung möglich ist.

Abbildung 2.3 zeigt, wie einfach die Methode *Fits* für Töpfe und Deckel mit einem „genetischen Code“ auf dieser Basis zu implementieren ist.

Bemerkung: Ob jemand in Australien den gleichen Hausschlüssel wie Herr X aus Ulm benutzt, ist nicht wichtig; daß die beiden verschiedene Kreditkartennummern haben, dagegen schon. — In Fällen, in denen es wirklich auf absolute Eindeutigkeit ankommt, bedarf es zusätzlicher Maßnahmen. Letztlich kann nur durch irgendeine Form der Koordination die theoretisch geforderte Sicherheit erreicht werden. (Diese muß nicht unbedingt zur Laufzeit erfolgen. Beispielsweise profitieren Netzwerke, in denen jeder Teilnehmer eine konstante eindeutige Identifikation besitzt, von einer vorher erzielten Einigung auf administrativer Ebene). Ein geeigneter Koordinationsmechanismus wird im nächsten Kapitel im Zusammenhang mit dem Modul *UniqueNames* vorgestellt.[3]

Zusätzliche Anforderungen

Mit der bisher besprochenen Funktionalität werden schon sehr viele Anwendungsmöglichkeiten abgedeckt. Ein typisches Objekt, das in das Autorisierungsschema eingebunden ist, wird eine oder mehrere private *Pot*-Komponenten besitzen und seine sensiblen Methoden mit je einem *Lid*-Argument versehen, das zu Beginn jeder Methode am entsprechenden Gegenstück geprüft wird. Folgende Optionen stehen ihm nun offen:

- Soll ein Dienst uneingeschränkt verfügbar sein, wird der ihm zugeordnete Topf durch einen „freundlichen“ Trivialtopf ersetzt.
- Sobald der Dienst geschützt werden soll, erhält er einen neuen von *PrivateShards* angeforderten Standardtopf.
- Ein Partner, dem die Nutzung des geschützten Dienstes eingeräumt werden soll, erhält einfach den dazugehörenden Deckel. Er muß eine Referenz darauf behalten, damit er ihn beim Aufruf der Methode als Argument angeben kann.
- Anstatt alle potentiellen Partner in nur zwei Gruppen einzuteilen (Berechtigung — ja oder nein), könnte auch zu jedem einzelnen eine separate Beziehung mittels eines eigenen Topf- und Deckelpaars unterhalten werden, die dadurch auch individuell wieder aufkündbar wäre.

Ein nützliches Autorisierungssystem sollte jedoch noch weiteren Ansprüchen gerecht werden. Neben der bereits angesprochenen Zuverlässigkeit des Prüfkriteriums ist hier vor allem die Sicherheit gegenüber verschiedenen Arten des Mißbrauchs zu nennen. (Die folgenden Überlegungen beschränken sich grundsätzlich auf Angriffe von außen, d.h. aus einem anderen Prozeß. Auf Programme, die sich selbst attackieren wollen, erstreckt sich das hier vorausgesetzte Verständnis des Begriffs „Mißbrauch“ nicht).

Objekte werden dann für die Außenwelt sichtbar, wenn sie Kommunikationskanäle benutzen. Persistente Objekte können dabei auch im ganzen zu einer neuen Lokation kopiert werden, und wer auch immer diese Daten empfängt, kann dadurch

[3] Siehe Abschnitt 3.3.3, Seite 50.

funktionsfähige Kopien der betreffenden Objekte erhalten. Insofern ist es einleuchtend, daß gerade Objekte, die gewisse Berechtigungen repräsentieren, tunlichst nur die richtigen Adressaten erreichen sollten.

Zu diesem Zweck empfiehlt sich der Einsatz kryptographischer Methoden: Die zu übertragenden Daten können so verschlüsselt werden, daß nur ein bestimmter Empfänger sie entschlüsseln kann, der wiederum in der Lage ist, den Absender und die Unversehrtheit der Nachricht eindeutig zu verifizieren. Ein *public-key*-Verfahren ist besonders zweckmäßig, wenn zu keinem Zeitpunkt ein „sicherer“ Übertragungsweg für Klartext zur Verfügung steht. Ferner können durch die Verwendung sogenannter Falltürfunktionen Töpfe und Deckel so konstruiert werden, daß es praktisch unmöglich ist, bei Kenntnis nur eines der beiden Teile den anderen zu rekonstruieren. Zum dritten kann nicht nur beim Versenden der Objekte, sondern gerade auch in dem Moment, wo sie kommunizieren (d.h. beim Prüfvorgang), ein gegen Abhören und Stören gesicherter Dialog verwendet werden. Eine Implementierung dieser Techniken in Oberon behandelt [Szczuka95].

Spezialfälle

Eine Reihe verschiedener Aufgaben läßt sich sehr gut mit geeigneten speziellen Implementierungen der Abstraktion *Shards* lösen. Das soll zum Abschluß dieses Kapitels durch einige Beispiele illustriert werden.

Eine erste Gruppe der Spezialisierungen zeichnet sich durch Variationen hinsichtlich dessen aus, was genau getestet werden soll. Das kann zum Beispiel die einfache Frage sein, ob die Funktion *Fits* in demselben Prozeß aufgerufen wird, in dem auch der kontrollierende Topf lebt. Diese Frage stellt sich in Situationen, wo bestimmte Dinge zwar von außen sichtbar sind, etwa weil sie in prozeßübergreifend vernetzten Datenstrukturen erscheinen, aber nur intern manipuliert werden können sollen. Eine zuverlässige Antwort hierauf liefert das Modul *VolatileShards* mit Hilfe eines kleinen Tricks, der gleichzeitig den Namen erklärt: Ein Deckel, der als (vermeintlich) persistentes Objekt auf die Reise geschickt wird, kommt stets in leicht veränderter Form auf der anderen Seite an.

Abbildung 2.4 zeigt alles Wesentliche der Realisierung des Moduls *VolatileShards*. Typspezifische Schreib- und Leseprozeduren werden hier nicht gebraucht. Zu beachten ist, daß in keinem der Konstruktoren die Funktion NEW verwendet wird, da sich die relevante Information problemlos in einer konstanten Zahl von Objekten unterbringen läßt.

Als weitere Vertreter auf bestimmte Tests spezialisierter Implementierungen seien noch genannt: *TransientShards*, ein Modul, das es erlaubt, Töpfe oder Deckel mit einem definierten Verfallsdatum auszustatten; *SemaphoreShards* — hier berücksichtigt jeder Topf den Zustand eines ihm zugeordneten Semaphors; und *CompositeShards*, ein Modul, mit dem man vorhandene Töpfe mit verschiedenen logischen Verknüpfungen zu neuen zusammensetzen kann.

```
PROCEDURE Fits(pot: Shards.Pot; lid: Shards.Lid): BOOLEAN;
BEGIN
   RETURN lid = mylid
END Fits;

PROCEDURE Supply(pot: Shards.Pot;
                 VAR lid: Shards.Lid): BOOLEAN;
BEGIN
   lid := mylid; RETURN TRUE
END Supply;

(* ===== marshalling procedures: ===== *)

PROCEDURE CreatePot(VAR obj: PersistentObjects.Object);
BEGIN
   obj := mypot;
END CreatePot;

PROCEDURE CreateLid(VAR obj: PersistentObjects.Object);
BEGIN
   obj := foreignlid;            (* will never fit to mypot *)
END CreateLid;
```

Abbildung 2.4: „Unlautere“ Methoden des Moduls *VolatileShards*

Eine andere Gruppe bilden Implementierungen, deren Objekte auf eine besondere Art der Handhabung ausgelegt sind. So zeichnen sich z.B. *PasswordShards* dadurch aus, daß Töpfe und Deckel völlig unabhängig voneinander, parametrisiert nach einer Zeichenkette hergestellt werden können. Sie passen dann zusammen, wenn ihre Zeichenketten übereinstimmen. *RemoteShards* schließlich machen Töpfe und mithin Autorisierungskontrollen fernsteuerbar.

Ausblick

Mit diesen Beispielen sind die denkbaren Konkretisierungen der vorgestellten Abstraktion *Shards* durchaus noch nicht ausgeschöpft. Jeder neue Anwendungsfall kann eine neue Sichtweise auf die Verwendungsmöglichkeiten eines Protokolls auslösen und auch neue konkrete Ausprägungen motivieren. Der Abstraktion selbst fällt die wichtige Rolle als gemeinsame Basis und Verständigungsgrundlage für unterschiedlichste Weiterentwicklungen einerseits und zahlreiche sich ebenfalls ausdifferenzierende Einsatzbereiche andererseits zu. Um eine solche Rolle angemessen erfüllen zu können, muß sie gut gewählt sein: so einfach wie irgend möglich — primitiv in des Wortes bester Bedeutung — und dennoch den Kern einer Sache treffend, eine brauchbare Ausdrucksmöglichkeit schaffend. James White, der in den siebziger Jahren neue Protokolle für das ARPANET [White76] entwickelte, bemerkt:

> The choice of model also constrains the kind and range of protocol extensions that are likely to occur to the systems programmer; one model may suggest a rich set of useful extensions, another lead nowhere (or worse still, in the wrong direction).

Autorisierung ist ein wichtiger Aspekt in Systemen, die heute entwickelt werden — von der Wegfahrsperre bis zum elektronischen Bargeld, von der Versicherungskarte bis zum Anrufbeantworter. Sie soll ohne unnötigen Aufwand, aber bei Bedarf mit ausreichender Sicherheit und Zuverlässigkeit funktionieren. Um diesem Ziel nahe zu kommen, muß mit großer Sorgfalt vorgegangen werden; vor allem sollte *security* (Sicherheit) nicht mit *obscurity* (Unübersichtlichkeit) verwechselt werden, will man nicht den Gebrauch einer Sache stärker behindern als ihren Mißbrauch.

Anwendungsbeispiele für das Protokoll finden sich in den folgenden Kapiteln.

Kapitel 3

Namensräume

3.1 Konzeption

Assoziativität

Wann immer Informationen „allgemein“, d.h. für Interessenten mit wenigen oder stark unterschiedlichen Vorkenntnissen, bereitgestellt werden sollen, kommt es besonders auf eine geeignete Strukturierung des Zugangs zu diesen Informationen an. Dabei bewährt sich das Prinzip der Assoziativität, daß also von einer Fundstelle Verweise zu thematisch zusammenhängenden anderen Stellen führen.

So kann etwa ein Benutzer einer öffentlichen Bibliothek vom Schlagwortkatalog zu einem Regalbrett mit spezieller Fachliteratur gelangen, dort ein Buch aufschlagen, über das Inhaltsverzeichnis ein bestimmtes Kapitel finden, im Text auf eine Fußnote stoßen, die wiederum einen Literaturhinweis enthält, der ihn zunächst wieder zum Katalog und schließlich genau zu einem benötigten Zeitschriftenartikel führt, von dessen Existenz er beim Betreten der Büchereiräume noch gar nichts wußte.

Abstrakt ausgedrückt handelt es sich bei assoziativ verknüpften Informationen um gerichtete Graphen. Ein sehr geläufiges Beispiel aus dem Bereich der Informatik für die Implementierung eines solchen Modells stellt das typische Dateisystem dar. Dateien (*ordinary files*) und Verzeichnisse (*directories*) entsprechen den Knoten, während die in einem Verzeichnis enthaltenen Namen die von dort wegführenden Kanten repräsentieren.

Reichweite

Das Dateisystem stellt einen Namensraum für die Daten, die ein Computer verwaltet, dar: Um eine bestimmte Datei anzusprechen, ist nur ein ihr zugeordneter Pfadname nötig und nicht etwa explizite Positionierungsbefehle für den Lesekopf des Plattenspeichers oder dergleichen. Entsprechend einfach sollen beliebige Objekte in Oberon durch Namensräume zugänglich gemacht werden.

Ähnlich, wie es durch geeignete Software ermöglicht wird, daß ein Dateisystem nicht nur lokale Dateien repräsentiert, sondern sich über ein ganzes Netzwerk erstreckt, ohne daß für entfernte und lokale Dateien etwa verschiedene Zugriffsoperationen benutzt werden müßten, sollen auch Objekt-Namensräume über mehrere Parteien verteilt sein können. Eine Partei ist hier ein beliebiger Prozeß mit Oberon-Objekten.

In einer konzeptionell so wenig begrenzten Welt ist Flexibilität ein Gesichtspunkt von zentraler Bedeutung. Ein Namensraum-Objekt muß dynamisch verändert werden können, wobei den beteiligten Parteien allerdings differenzierte Kontrollmöglichkeiten einzuräumen sind. Dazu gehören insbesondere Zugriffsbeschränkungen aller Art, aber auch rein passive Beobachtungsmechanismen. Nur so wird ein Objekt, das mit anderen zu kommunizieren bereit ist, nicht zwangsläufig beliebig manipulierbar.

Vielseitigkeit

Ein Namensraum soll Zugang zu vorher unbekannten Objekten vermitteln können. Bei konsequenter Auslegung muß sich diese Forderung nicht nur auf Instanzen, sondern auch auf Klassen erstrecken, d.h., bei den in die Graphenstruktur eingebetteten Objekten sollte kein bestimmter Datentyp vorausgesetzt werden. Neben Flexibilität innerhalb eines Namensraums verspricht dies auch Vorteile für die vielseitige Verwendbarkeit des Protokolls an sich. Gerade im Zusammenhang mit Namensräumen gibt es sehr gute Erfahrungen damit, ein und dieselbe Schnittstelle für Zugriffe auf die unterschiedlichsten Ressourcen zu verwenden, wie z.B. das *9P*-Protokoll in *Plan 9* [Pike93].

Eine weitere Konsequenz aus dem Bestreben, wenig Vorwissen zu verlangen, ist, daß auch die Namen selbst in Erfahrung zu bringen sein sollten. Das System soll also nicht nur bekannte Namen auf Objekte abbilden, sondern auch Auskunft darüber geben, welche Namen in einer gewissen lokalen Umgebung existieren. Eine globale Betrachtung verbietet sich selbstverständlich aus Gründen der Skalierbarkeit. Da das System dynamisch ist, kann eine einmalige Auskunft rasch veralten; wünschenswert erscheint deswegen auch ein Mechanismus, Veränderungen so zu propagieren, daß direkt darauf reagiert werden kann.

Versteht man Namen als Hinweise zur Bestimmung von Objekten, erkennt man leicht, daß es günstig sein kann, ein Objekt über mehrere verschiedene Namen anzusprechen, die dann verschiedene Kriterien wiedergeben, die es alternativ bezeichnen. Abbildung 3.1 mag dies illustrieren. Von Namensgraphen sollte also im allgemeinen keine Baumstruktur erwartet werden.

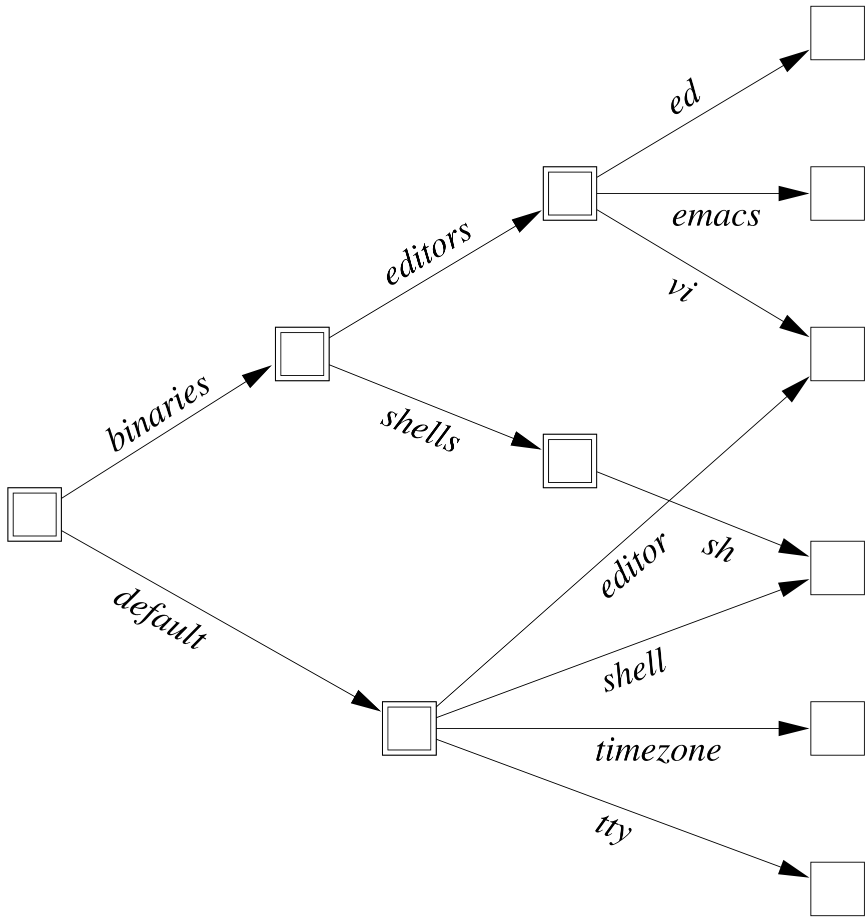

Abbildung 3.1: Beispiel für einen Namensraum

Relativität

Namen werden unter anderem dazu verwendet, daß ein Objekt von verschiedenen Seiten referenziert werden kann. Als Minimalvoraussetzung dafür gilt, daß die Auflösung dieser Namen im jeweiligen Kontext zum gleichen Ergebnis führt [Needham93].

Viele Adressierungssysteme benutzen einen globalen Namensraum in der Weise, daß jedes Objekt einen überall gültigen Namen erhält. Zur eigentlichen Lokalisierung des Objekts und zur Umsetzung der absoluten Lokation in eine relative Verbindungsroute (*routing*) bedarf es dann einer geeigneten Suchstrategie, etwa in Gestalt einer Kaskade von Abfragen regionaler Agenten, die jeweils entweder die fragliche Adresse auflösen können oder den Auftrag an einen weiteren Agenten delegieren. Beispiele für diese Technik, die keineswegs zufällig der Zustellung von Briefpost ähnelt, finden sich unter anderem in Grapevine [Birrell82] und bei Host-

Adressen im Internet [Clark82]. Ein anderer Zugriffsmechanismus, aber ebenfalls das Prinzip der Globalität wird in DHS, einem in Oberon geschriebenen verteilten Betriebssystem, verwendet [Traub94], [Lupper94].

Hingegen wird im Ulmer Oberon-System, wie im Abschnitt 1.3.5 bereits angedeutet, der Aufbau von Verbindungen zu möglicherweise entfernten Objekten auf einer eigenen Abstraktionsebene gelöst, hängt also nicht von einer speziellen Eigenschaft der Abstraktion für Namensräume ab. Deswegen kann auf Globalität in dem Sinne, daß die Abbildung von Namen auf Objekte immer ortsunabhängig sein muß, verzichtet werden. Das ist ein Vorteil, denn so können innerhalb einer Laufzeitumgebung Teile desselben Namensraums für lokal gültige, andere für absolute Zuordnungen verwendet werden. Im Beispiel von Abbildung 3.1 könnte etwa der *default*-Zweig zur Laufzeit in Abhängigkeit von Umgebungsvariablen gebildet worden sein, während der Rest aus einer systemspezifischen Datenbank stammt und wieder ein anderer, nicht dargestellter Teil tatsächlich einen über externe Namensserver zugänglichen globalen Adreßraum realisiert.

Daß es nützlich sein kann, nur bestimmte Teile eines Namensraums für Objekte, die mehreren Parteien zugänglich sein sollen, zu verwenden, während der Rest proprietär ist, zeigt sich auch am Beispiel verteilter Dateisysteme wie z.B. *Andrew* [Satyanarayanan93].

Konkretisierung

Ausgehend von dem bisher Gesagten lassen sich nun einige wesentliche Eigenschaften einer Abstraktion von Namensräumen in Oberon festlegen:

- Namensräume sind gerichtete Graphen.
- Jeder Knoten kann über (untereinander verschiedene) Namen auf andere Knoten verweisen.
- Knoten können anderen Prozessen gehören. Jeder Prozeß kann Knoten für die Außenwelt verfügbar machen und wird so zum Namensserver.
- Jeder Namensraum ist relativ zu einem bestimmten Startpunkt zu verstehen. Einheitlichkeit von Teilen davon in einem größeren Kontext bis hin zur Globalität wird erreicht durch den Import geeigneter Bezugsknoten.
- Zugriffsoperationen, z.B. Löschen, Einfügen usw., können mit einem Autorisierungsschema kontrolliert werden.
- Ein Interessent kann sich über Änderungen der von einem Knoten ausgehenden Verweise automatisch benachrichtigen lassen.

- Die geforderte Einbindung beliebiger Objekte in den Graphen wird durch eine optionale Eins-zu-eins-Zuordnung von allgemeinen Objekten zu Knoten erreicht.

In bezug auf diesen letzten Punkt könnte die Frage gestellt werden, warum die Knoten-Eigenschaften nicht einfach als spezielle Sicht auf ein ansonsten unspezifisches Objekt realisiert werden. Ein eigener Datentyp für Knoten spiegelt jedoch stärker die Tatsache wieder, daß diese Eigenschaften häufig nur für sich allein auftreten werden, besonders bei den Knoten, die Nachfolger haben, also höchstwahrscheinlich eher für eine Verallgemeinerung als etwas Konkretes stehen. Die Semantik ist in beiden Fällen dieselbe, d.h., zu einem Objekt gehört maximal ein Satz von Eigenschaften als Knoten im Sinne der Namensraum-Abstraktion. Sollte jedoch eines Tages der Wunsch nach konkurrierenden Namensräumen mit unterschiedlichen Eigenschaften für dieselben Objekte aufkommen, genügte es bei der gewählten Form, die Zuordnungsregel zu lockern, die, wie sich zeigen wird, ohnehin separat implementierbar ist. Bei der anderen Vorgehensweise bestünde diese Möglichkeit nicht.

Zusammenfassend kann festgestellt werden, daß mit der vorgestellten Art von Namensräumen die Voraussetzung für die Identifikation und Lokalisierung von Objekten auch über Prozeßgrenzen hinweg, aber dennoch mit der Möglichkeit zu beliebigen Abstufungen der Lokalität gegeben ist.

3.2 Realisierung

3.2.1 Objekt- und Parametertypen

Entsprechend der im ersten Kapitel genannten Prinzipien wird eine neue Abstraktion in Form eines Basismoduls in die Oberon-Bibliothek integriert, das die für die Abstraktion charakteristischen Typen und Methoden definiert. Das Basismodul für Namensräume in Oberon heißt schlicht *Names*. Es dient als gemeinsamer Bezugspunkt für Anwendungen, Erweiterungen und spezielle Implementierungen des vorgestellten Konzeptes. Eine einfache Implementierung ist direkt eingebaut.

Namensräume sind gerichtete Graphen, und Graphen bestehen aus Knoten und Kanten (*vertices* und *edges*). Da die Kanten durch einfache Zeichenfolgen repräsentiert werden sollen, ergibt sich als alleiniger zentraler Objekttyp dieser Abstraktion der Knoten (*Names.Node*).

Da für Knoten auch typenunabhängige Abstraktionen in Frage kommen, etwa um Dienste wie *RemoteNames*[1] separat implementieren zu können, ist *Names.Node* eine Erweiterung von *Services.Object*.

[1] siehe Abschnitt 3.3.4

Ein Name, über den von einem Knoten auf einen nachgeordneten Knoten verwiesen wird, soll eine beliebig lange Folge beliebiger Zeichen sein können. Zwar ist es sicherlich sinnvoll, in konkreten Fällen gewisse Namenskonventionen einzuhalten, jedoch spielen dergleichen Einschränkungen für das abstrakte Modell keine Rolle. Im Gegenteil kann nur durch größtmögliche Allgemeinheit erreicht werden, daß eine Abstraktion mit jeder erdenklichen Konkretisierung harmoniert.

Für den effizienten Umgang mit solchen Zeichenketten stellt die Oberon-Bibliothek einen eigenen Datentyp zur Verfügung — *ConstStrings.String* —, der somit auch als *Names.Name* zum Einsatz kommt. Diese Art von Zeichenketten verursacht einen kleinen Zusatzaufwand beim Einlesen, wird aber platzsparend gespeichert und kann sogar günstiger als gewöhnliche (Null-Byte-terminierte) Zeichenketten verglichen oder vervielfältigt werden.

Einzelheiten des Autorisierungsschemas wurden im vorhergehenden Kapitel behandelt. In der Schnittstelle von *Names* schlägt sich die Autorisierungsproblematik in erster Linie durch die Anwesenheit von Parametern des Typs *Shards.Lid* bei einzelnen Methoden nieder.

Wichtig ist, daß Zugriffe auf Knoten genau nach Zugriffsarten (*access modes*) unterschieden werden, für die ggf. getrennte Berechtigungen verwaltet werden. *Names.Permissions* ist ein Vektor von Autorisierungsagenten, der nach Zugriffsarten indiziert wird.

In Analogie zum Dateisystem werden „administrative“ Attribute eines Knotens als sein Status bezeichnet. In der Standard-Implementierung umfaßt der Status genau die Berechtigungen.

Ein Ereignis, das eine Veränderung der Kantenmenge eines Knotens signalisiert, trägt als Information neben der Art der Veränderung auch diesen Knoten und den hinzugekommenen bzw. nicht mehr gültigen Namen. Ein zusätzlicher Verweis auf den über den Namen referenzierten Unterknoten ist, wie es sich zeigen wird, wohlweislich nicht enthalten.

In Abbildung 3.2 sind die entscheidenden Typdefinitionen der Abstraktion zusammengestellt.

3.2.2 Methoden

In welcher Weise bei einem Knoten die zu ihm gehörenden Kanten abgespeichert werden, wird von der Abstraktion nicht festgelegt. Die Kantenmenge soll allein über die zum Knoten-Objekt gehörenden Methoden verfügbar sein.

Methoden werden von dem die Abstraktion definierenden Modul als gewöhnliche Prozeduren vereinbart, deren erstes Argument das betroffene Objekt ist. Aufrufe von Methoden werden in der Regel auf Aufrufe von Prozeduren aus einer zum Objekt gehörenden Schnittstelle (*interface*) abgebildet.

```
CONST
   examine = 0;  change = 1;  read = 2;  search = 3;
   insert = 4;  delete = 5;  destroy = 6;  accessmodes = 7;

   inserted = insert;  deleted = delete;

   accessCap = 0;  statusCap = 1;
   domainCap = 2;  destroyCap = 3;

TYPE
   Node = POINTER TO NodeRec;
   NodeRec = RECORD (Services.ObjectRec) END;

   Name = ConstStrings.String;
   Lister = Iterators.Iterator;

   AccessMode = SHORTINT;     (* [examine..destroy] *)
   State = SHORTINT;          (* inserted or deleted *)
   CapabilitySet = SET;       (* of [accessCap..destroyCap] *)

   Permissions = ARRAY accessmodes OF Shards.Pot;

   Status = POINTER TO StatusRec;
   StatusRec =
      RECORD
         (Objects.ObjectRec)
         perm: Permissions;
      END;

   MemberEvent = POINTER TO MemberEventRec;
   MemberEventRec =
      RECORD
         (Events.EventRec)
         state: State;
         name: Name;          (* of inserted/deleted member *)
         node: Node;          (* whose member it became/was *)
      END;
```

Abbildung 3.2: Typdefinitionen des Moduls *Names*

Das Modul *Names* gibt Implementierungen die Freiheit, gewisse Gruppen von Methoden zu implementieren und andere nicht, wobei die genaue Auswahl durch die „Fähigkeiten“ (*capabilities*) des Knotens bestimmt wird. Je nach vorliegenden Fähigkeiten kann die Methode dann, anstatt eine Interfaceprozedur aufzurufen, auch eine Standard-Aktion durchführen oder nur eine Fehlerbedingung weitergeben.

In Tabelle 3.1 sind alle Methoden zusammengestellt, die im Folgenden näher erläutert werden:

InitNode ist die übliche Methode, neue Instanzen eines von *Names.Node* abgelei-

<table>
<tr><th>Methode</th><th>Zugriffsart</th><th>Fähigkeit</th><th>vordefiniert?</th></tr>
<tr><td>InitNode</td><td rowspan="2"></td><td rowspan="2"></td><td rowspan="2"></td></tr>
<tr><td>CreateNode</td></tr>
<tr><td>Access</td><td></td><td>accessCap</td><td>ja</td></tr>
<tr><td>GetStatus</td><td>examine</td><td rowspan="2">statusCap</td><td rowspan="2">nein</td></tr>
<tr><td>SetStatus</td><td>change</td></tr>
<tr><td>GetMembers</td><td rowspan="2">read</td><td rowspan="5">domainCap</td><td rowspan="5">nein</td></tr>
<tr><td>TakeInterest</td></tr>
<tr><td>GetNode</td><td>search</td></tr>
<tr><td>Insert</td><td>insert</td></tr>
<tr><td>Delete</td><td>delete</td></tr>
<tr><td>Destroy</td><td>destroy</td><td>destroyCap</td><td>ja</td></tr>
</table>

Tabelle 3.1: Methoden des Moduls *Names* im Überblick

teten Objekts zu initialisieren, d.h., mit dem zur Abstraktion gehörenden Interface zu verbinden.

CreateNode erzeugt einen Knoten mit der eingebauten einfachen Implementierung, wobei die gewünschten Fähigkeiten angegeben werden können.

Access erteilt Auskunft über die Berechtigung zu einer bestimmten Zugriffsart, ohne den Zugriff tatsächlich auszuführen. Falls für den betreffenden Knoten *Access* nicht implementiert ist, wird lediglich geprüft, ob er die vorausgesetzte Fähigkeit besitzt.

GetStatus und *SetStatus* stellen den Status eines Knotens zur Verfügung, falls dieser darauf eingerichtet ist.

GetMembers liefert die Menge der von einem Knoten wegführenden Kanten (d.h. die Namen seiner „Unterknoten") in Form eines Iterators. Iteratoren abstrahieren eine Art Pipeline für Objekte. Sie eignen sich besonders gut für sequentielle Abfragen großer Datenstrukturen, weil sie Container überflüssig machen und weil durch die implizite Synchronisierung des Produzenten mit dem Konsumenten die Länge der Traverse einfach und direkt von der Menge der tatsächlich abgeholten Datensätze bestimmt wird.

TakeInterest veranlaßt, daß bei jeder Änderung der Menge der nachfolgenden Knoten eines Knotens ein Ereignis mit einer entsprechenden Nachricht ausgelöst wird. Dies erlaubt interessierten Parteien, stets auf dem Laufenden zu bleiben, ohne ständig aktiv nach dem neuesten Stand fragen zu müssen.

Wie *GetMembers* gilt auch *TakeInterest* als ein Lesezugriff auf den Knoten (Zugriffsart *read*), der dieselbe Art von Informationen liefert und an dieselbe Berechtigung geknüpft ist.

GetNode ist wohl die wichtigste Methode, denn sie realisiert den Schritt entlang einer Kante von einem Knoten zum nächsten. Konkret heißt das, der Knoten wird nach einem Unterknoten mit einem bestimmten Namen gefragt und liefert als Resultat diesen Knoten, falls er ihn tatsächlich kennt und die Abfrage aufgrund ihrer Autorisierung als berechtigt ansieht.

In diesem Zusammenhang sollte die Bedeutung der Differenzierungsmöglichkeiten zwischen Berechtigungen für verschiedene Zugriffsarten nicht unterschätzt werden. Beispielsweise kann über die Autorisierung bei *GetNode* kontrolliert werden, welche konkreten Objekte für eine bestimmte Partei erreichbar sind, während *GetMembers* unabhängig davon die Sichtbarkeit der dazugehörenden Namen regelt.

Da die Namen selbst bereits einen wesentlichen Teil der Information bilden, die durch Namensräume dargestellt wird, kann es sinnvoll sein, diese einem weiteren Kreis möglicher Interessenten zugänglich zu machen als die dahinter verborgenen Objekte. Umgekehrt könnte man auch bewußt die Abfrage kompletter Verzeichnisse einschränken (etwa um Übertragungskosten zu sparen), während die einzelnen Objekte durchaus allgemein verfügbar bleiben.

Bemerkung: Ein anderes Detail in der Systematik der Berechtigungen erwies sich nach ersten praktischen Erfahrungen allerdings als diskussionswürdig: die Differenzierung von *GetStatus* und *SetStatus* als unterschiedliche Zugriffsarten. Weil Oberon bei Zeigern nicht unterscheidet, ob die referenzierten Daten nur gelesen oder auch verändert werden dürfen, muß die Weitergabe eines nicht zum Verändern bestimmten Verbunds zwangsläufig in Form einer Kopie erfolgen, was die entsprechende Methode leider verkompliziert. Andererseits werden diese Informationen in der Regel genau deswegen angefordert, weil sie gleich darauf verändert werden sollen — Berechtigungen lediglich zu testen erlaubt ja schon *Access*, wenn man sich an die Konvention hält, *statusCap* nur zusammen mit *accessCap* zu implementieren. All dies legt nahe, in einer überarbeiteten Version von *Names* Statusinformationen doch als kanonische Objekte — mit allen Vorteilen für die Handhabung und Erweiterbarkeit — zu behandeln und entweder die Zusammenfassung der Zugriffsarten *examine* und *change* zu einer einzigen in Betracht zu ziehen oder zumindest innerhalb eines Prozesses auf die durch das Kopieren gewonnene Schutzwirkung zu verzichten.

Insert und *Delete* vergrößern bzw. verkleinern die Menge der von einem Knoten wegführenden Kanten. Diese Methoden betreffen also Namen, unter denen Knoten einander bekannt sind; die Knoten selbst werden so nicht erzeugt oder zerstört.

Destroy dient hingegen genau zum Löschen eines Knotens und implizit aller Kanten, die von ihm ausgehen oder zu ihm führen. Die Fähigkeit *destroyCap* regelt nicht, ob ein Knoten gelöscht werden kann, sondern ob eine spezifische Methode der Implementierung dabei eingeschaltet wird.

Anders als viele andere Abstraktionen offeriert das Modul *Names* also eine explizite Destruktor-Methode für seine Objekte. Das hat verschiedene Gründe. Die stärkste Motivation zu dieser Methode kommt aus der nützlichen Rolle, die sie im Autorisierungssystem einnimmt. Mit ihr kann für jedes Objekt die Erlaubnis, seine Termination einzuleiten, unabhängig von den Namen, unter denen es erreichbar ist, verwaltet werden. Das Löschen eines Knotens unter der Kontrolle dieser Methode erspart überdies den Aufwand, auch die zugehörigen Namen einzeln zu löschen, da dies implizit erledigt werden kann. Dies wiederum ist sinnvoll, weil veraltete Namen in jedem Fall und ohne weitere Autorisierungshindernisse verschwinden sollen. Diese für die Benutzung angenehme Semantik muß freilich mit etwas mehr Aufwand bei der Implementierung bezahlt werden.

Die genauen Definitionen der Methoden sind aus Abbildung 3.3 ersichtlich.

3.2.3 Variable

Eine besondere Funktion hat die globale Variable *Names.root*, deren Definition in Abbildung 3.4 wiedergegeben ist. Sie ist ein Knoten, der immer existiert und unter dem vom Laufzeitsystem Referenzen auf externe Objekte angelegt werden können. Auf diese Weise können sich viele Prozesse einen Namensraum teilen, ohne daß für eine Kontaktaufnahme zu Namensservern eigens gesorgt werden müßte. *Names* kann somit als eine Abstraktion verwendet werden, die den Import und Export von Objekten generell transparent macht.

3.2.4 Interne Implementierung

Standard- versus Triviallösung

Basismodule von Abstraktionen enthalten häufig minimale Implementierungen, die nicht viel mehr tun, als die formalen Eigenschaften auf eine triviale Art zu erfüllen. So sind etwa in *Streams* die immerzu leeren Kanäle, in *Clocks* die stehende Uhr, in *Timezones* die Zone der Weltzeit und in *Shards* die stereotypen Antworten eingebaut. Trotz ihrer Trivialität kann kaum bezweifelt werden, daß diese Implementierungen nützlich sind, besonders, wo sie auch zur Initialisierung globaler Variabler dienen.

Bei *Names* liegen die Verhältnisse ein wenig anders. Eine denkbare Trivialform von Knoten ohne Unterknoten, also auch ohne Namen, wäre offensichtlich absurd, zumal der Startknoten *Names.root* ja Verweise zu weiteren Objekten aufnehmen können soll. Sobald aber Knoten mit der Fähigkeit *domainCap* und sämtlichen dazugehörenden Methoden realisiert sind, fehlt nur noch so wenig zu einer vollwertigen Implementierung, daß es wohl vernünftig ist, dieses wenige — die Unterstützung von *accessCap*, *statusCap* und *destroyCap* — hinzuzunehmen und so bereits intern eine substantielle, alle Möglichkeiten des Protokolls realisierende Lösung anzubieten.

```
PROCEDURE InitNode(      node: Node;
                         if: Interface; caps: CapabilitySet);

PROCEDURE CreateNode(VAR node: Node;
                         status: Status; caps: CapabilitySet);

PROCEDURE Access(        node: Node; mode: AccessMode;
                         auth: Shards.Lid): BOOLEAN;

PROCEDURE GetStatus(     node: Node;
                         auth: Shards.Lid;
                     VAR status: StatusRec): BOOLEAN;

PROCEDURE SetStatus(     node: Node; status: Status;
                         auth: Shards.Lid): BOOLEAN;

PROCEDURE GetMembers(    node: Node;
                         auth: Shards.Lid;
                     VAR members: Lister): BOOLEAN;

PROCEDURE TakeInterest(  node: Node;
                         auth: Shards.Lid;
                     VAR eventType: Events.EventType): BOOLEAN;

PROCEDURE GetNode(       node: Node; name: Name;
                         auth: Shards.Lid;
                     VAR subnode: Node): BOOLEAN;

PROCEDURE Insert(        node, subnode: Node; name: Name;
                         auth: Shards.Lid): BOOLEAN;

PROCEDURE Delete(        node: Node; name: Name;
                         auth: Shards.Lid): BOOLEAN;

PROCEDURE Destroy(       node: Node;
                         auth: Shards.Lid): BOOLEAN;
```

Abbildung 3.3: Methoden des Moduls *Names*

```
VAR
   root: Node;
```

Abbildung 3.4: Definition der Variablen *Names.root*

Eine solche Standardlösung ist also im Basismodul *Names* enthalten. Sie beruht auf einer privaten Erweiterung des Knoten-Objekttyps um einen Status, von dem nichts als der obligate Vektor für Berechtigungen benutzt wird, und einer Datenstruktur zur Aufnahme von Kanten. Objekte dieses Typs können mit *Names.Create* erzeugt werden. Obwohl sie im Grunde alle Fähigkeiten hätten, können diese selektiv festgelegt werden. Alle zur Standardlösung gehörenden Definitionen sind privat.

Datenrepräsentation

Standardknoten sind — wie Knoten im allgemeinen — nicht persistent. Ihre Kantenmengen werden in der vorläufigen Fassung als gewöhnliche, im dynamischen Speicher angelegte binäre Bäume verwaltet. Eine spätere Fassung könnte ausgefeiltere Algorithmen verwenden, die dann aber wohl in einer separaten Abstraktion für Datenstrukturen anzusiedeln wären. Derartige Abstraktionen gibt es bisher in der Ulmer Oberon-Bibliothek noch nicht,[2] weil die Bedürfnisse der vorhandenen Module entweder zu unterschiedlich oder hinreichend gut mit elementaren Mitteln erfüllbar waren. Namensräume selbst könnten sogar in Konkurrenz zu manchen dieser künftigen Abstraktionen treten, da sie ein relativ allgemeines Konzept ohne allzuviel störenden Ballast verwirklichen (es sei daran erinnert, daß z.B. die Autorisierung nur sehr wenig kostet, solange nur einfache oder gar keine Autorisierungsagenten benutzt werden).

Kooperation bei Aufräumarbeiten

Nicht zuletzt wegen der Anforderungen an das Verhalten von Namensräumen beim „Zerstören“ von Knoten ist es nützlich, in einem kleinen Ausflug die Mechanismen näher zu betrachten, die in der Bibliothek für Koordinationsaufgaben im Zusammenhang mit der Termination von Objekten vorgesehen sind. Eine geeignete Abstraktion offeriert das Modul *Resources*. Sie gibt eine Vorgehensweise vor, um Aufräumarbeiten für nicht mehr benötigte Ressourcen zum richtigen Zeitpunkt zu veranlassen. Soll ein System skalierbar sein, also prinzipiell keinerlei festen Begrenzungen unterliegen, ist es unabdingbar, daß alle beschränkt vorhandenen Mittel nach Gebrauch der Wiederverwendung zugänglich gemacht werden. Dazu bedarf es der Kooperation unter den Nutznießern dieser Mittel — was leider nicht in allen Bereichen so einfach zu erreichen ist wie bei Softwarekomponenten.

Die Kanten, die zu einem ungültig gewordenen Knoten führen, sollen automatisch entfernt werden. Dazu muß jede Implementierung, die Kanten verwaltet, erfahren können, wann der Zeitpunkt zum Bereinigen einer solchen Situation gegeben ist. An der Stelle, wo über die Termination eines Objektes entschieden wird, dient

[2] Es existieren vielversprechende Ansätze, vgl. etwa [Borchert95b, Keys(3)].

Resources.Notify dazu, potentielle Interessenten von dessen „tragischem Ende" zu benachrichtigen. Umgekehrt sorgt in einem Kontext, in dem auf die Termination reagiert werden soll, *Resources.TakeInterest* dafür, daß ein entsprechendes Ereignis ausgelöst wird, sobald dieser Fall eingetreten ist.

Dieses Ereignis gilt pauschal für das terminierte Objekt, ohne daß für die Interessenten ein Zusammenhang zu einzelnen davon betroffenen anderen Objekten hergestellt würde. Daher müssen z.B. Module, die Namensräume implementieren, sofern sie tatsächlich Kantenbeziehungen selbst verwalten,[3] Informationen über die eigenen zu einem Knoten führenden Kanten bei diesem Knoten aufbewahren. Woher ein Knoten sonst noch referenziert werden mag, ist für die einzelne Implementierung unerheblich; sie kennt ja auch nur die Kanten, für die sie selbst zuständig ist.

Speicherplatz wird in Oberon nicht explizit freigegeben, sondern mit automatischer Speicherbereinigung (*garbage collection*) zurückgewonnen. Diese kann freilich nicht erkennen, daß ein Objekt nicht mehr gebraucht wird, solange noch irgendwelche Referenzen darauf existieren, selbst wenn diese nur zu Verwaltungsstrukturen gehören, die nicht als eigentlicher Zugang zu dem Objekt gedacht sind. Mit Hilfe von *Resources* besteht eine Möglichkeit, diese nicht entscheidenden Referenzen (*lightweight references*) und damit letztlich das Objekt unter geeigneten Vorkehrungen dennoch loszuwerden. Dazu gehören zwei Dinge: Erstens sollte überall dort, wo Verweise aus anderen Datenstrukturen auf ein Objekt gehalten werden, auf das Terminationsereignis reagiert werden, indem diese Verweise entfernt werden, sodaß auch die „leichtgewichtigen", von außen nicht zugänglichen Referenzen anschließend der automatischen Speicherfreigabe nicht mehr im Wege stehen. Zweitens muß das Einrichten sowie die Aufhebung jeder (ge-)wichtigen Referenz mit *Resources.Attach* bzw. *Resources.Detach* angekündigt werden. Den Umstand, daß die letzte Referenz dieser Art auf ein Objekt aufgehoben worden ist, können Interessenten dann über ein Ereignis erfahren, das dem Terminationsereignis ähnelt, aber statt *Resources.terminated* die Information *Resources.unreferenced* trägt.

Solche Ereignisse sind grundsätzlich als Signal dafür zu verstehen, daß ein Objekt nicht mehr benötigt wird und die Beendigung seiner Existenz unmittelbar bevorsteht. Gleichwohl gilt das betroffene Objekt in diesem Moment noch nicht als terminiert, sondern bleibt voll funktionsfähig und offen für Interaktionen, was für Parteien interessant ist, die bei einer angekündigten Termination mehr zu tun bereit sind als bei schon vollendeten Tatsachen (in einen Datenkanal etwa können noch gepufferte Daten ausgegeben werden, was keinen Sinn mehr hat, sobald er tatsächlich unbrauchbar geworden ist). Letztendlich sollte dieses Zwischenstadium jedoch zur endgültigen Termination führen, damit auch die dazu gehörenden

[3] Im Abschnitt 3.3.4 lernen wir mit *RemoteNames* eine Implementierung kennen, die keine Kantenverwaltung nötig hat.

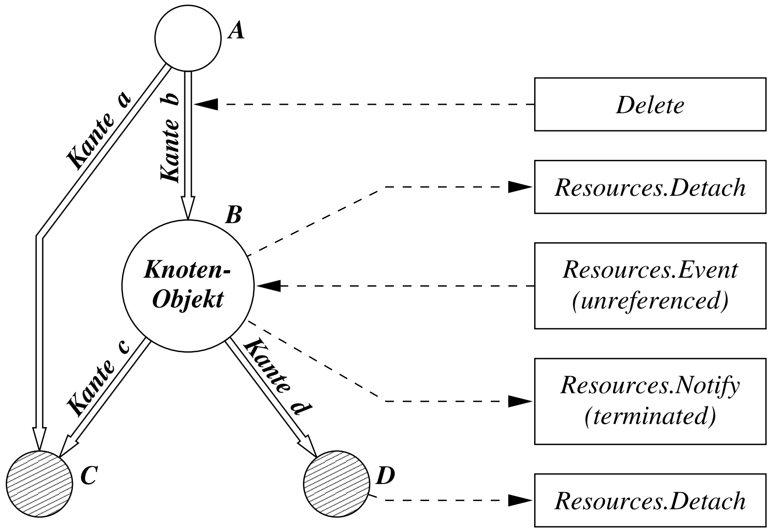

Abbildung 3.5: Koordination von Aufräumarbeiten für Knoten (Beispiel)

Reaktionen folgerichtig ausgelöst werden. Wenn die Basisabstraktion dies veranlaßt, spart das nicht nur Arbeit bei allen Implementierungen, sondern garantiert auch — weil Ereignisbearbeiter in umgekehrter Reihenfolge ihrer Anmeldung aktiviert werden — dafür, daß alle Interessenten zum Zuge gekommen sind, bevor das Objekt in den nächsten, letzten Zustand wechselt.

Dementsprechend wird in *Names* für jeden neu erzeugten Knoten sofort ein Ereignisbearbeiter eingesetzt, der unter anderem bei „Unreferenziertheit“ unverzüglich dessen Termination veranlaßt. Die Rückwärtsverweisliste auf von Standardknoten ausgehende Kanten, die Bestandteil der privaten Sicht der Standardimplementierung auf jeden Knoten ist, stellt ein typisches Beispiel für „leichtgewichtige“ Referenzen dar. Ob diese Liste leer ist oder nicht, ist ein eindeutiger Hinweis, ob Standardkanten zu einem Knoten führen (Kanten selbst sind selbstverständlich gewöhnliche wichtige Referenzen). Daher braucht nicht über jede einzelne Standardkante mit *Resources.Attach* und *Resources.Detach* Buch geführt zu werden, sondern nur darüber, ob diese Liste ein erstes Element erhält bzw. leer wird.

Da Knoten somit grundsätzlich empfindlich auf den Verlust wichtiger Referenzen reagieren, muß jeder Knoten, dessen Lebensdauer nicht an seine Einbindung in einen Namensraum gebunden sein soll — etwa weil er selbst die Wurzel eines Namensraumes bildet —, mit *Attach* gegen diesen Verlust geschützt werden. Etwas allgemeiner gesprochen sollten sich Anwendungen also dessen bewußt sein, daß ein Objekt den Konventionen von Resources gemäß zu behandeln ist. Speziell bei Implementierungen von Namensräumen verursacht dies erfreulicherweise überhaupt keinen nennenswerten zusätzlichen Aufwand, da „leichte“ und wichtige Referenzen hier stets paarweise auftreten und gemeinsam verwaltet werden und

Aufräumarbeiten ohnehin auf Veranlassung der Methode *Destroy* durchzuführen sind.

Abbildung 3.5 illustriert, wie die Abstraktion *Resources* innerhalb des Moduls *Names* benutzt wird. Die schraffiert dargestellten Knoten mögen zu einer anderen Implementierung gehören. Wenn hier z.B. die Kante b gelöscht wird, führt dies zu *Resources.Detach* für den Knoten B, da ihn anschließend keine Standardkante mehr erreicht. Dies wiederum könnte ein *Resources.unreferenced*-Ereignis auslösen, das mit *Resources.Notify* sofort zur Termination von B umgemünzt wird. Infolge dieser Termination werden dann die Kanten c und d beseitigt, was eine *Detach*-Operation für den Knoten D, nicht jedoch für C bewirkt.

Anmerkungen

Mit skurrilen Autorisierungen oder zyklischen Graphen kann man erreichen, daß Teile von Namensräumen unerreichbar werden, ohne daß *Resources* dessen gewahr wird. In der Praxis spielen diese konstruierten Beispiele allerdings deswegen kaum eine Rolle, weil Namensräume typischerweise dergestalt über verschiedene Prozesse verteilt werden, daß Knoten genau dort angesiedelt sind, wo auch die zu den Namen gehörenden Inhalte existieren. Unabhängig von ihrer Erreichbarkeit bleiben diese Knoten natürlich in ihrer Lebensdauer an die sie beherbergenden Prozesse gebunden und können daher immer noch eliminiert werden, ohne daß andere Parteien davon beeinträchtigt würden.

Im Gegensatz zu *Names.Destroy* ist für *Resources.Notify* keinerlei Autorisierung nötig oder Plausibilitätskriterium zu erfüllen. Allerdings werden Informationen über Termination immer nur von Objekten zu Stellvertreter-Objekten[4] und nicht in der umgekehrten Richtung über Prozeßgrenzen weitergegeben, schon allein um die Möglichkeit zu haben, gezielt Stellvertreter-Objekte loszuwerden. Daher öffnen die verschiedenen Terminationsmechanismen keine Sicherheitslücke bei Objekten, die einem potentiellen Angreifer nicht selbst gehören, und deren Schutz allein als relevant zu betrachten ist.[5]

3.3 Auf *Names* aufbauende Module

3.3.1 *NamedObjects*

Zu den im ersten Abschnitt festgelegten Eigenschaften unserer Namensräume gehört auch die Forderung, daß beliebige Objekte in den Graphen eingebunden werden können sollen. Diese Möglichkeit wird von dem Modul *NamedObjects* eingeführt.

[4] Siehe Abschnitt 1.3.5.

[5] Vergleiche Abschnitt 2.3.

```
TYPE
   Object = Services.Object;

PROCEDURE Create(     VAR node: Names.Node; object: Object);

PROCEDURE Destroy(    node: Names.Node): BOOLEAN;

PROCEDURE Get(        node: Names.Node; VAR object: Object;
                      errors: RelatedEvents.Object): BOOLEAN;

PROCEDURE GuardedGet(node: Names.Node; guard: Services.Type;
                      VAR object: Object;
                      errors: RelatedEvents.Object): BOOLEAN;
```

Abbildung 3.6: Methoden des Moduls *NamedObjects*

Diese Abstraktion erfüllt eine Aufgabe, die in manchen Programmiersprachen im Wege von Mehrfachvererbung (*multiple inheritance*) lösbar wäre, nämlich Objekte mit Merkmalen aus verschiedenen unabhängigen Abstraktionen zu versehen. In Oberon stehen hierfür andere Techniken zur Verfügung. Handelte es sich bei den gewünschten Merkmalen um eine Art Dienst, der je nach vorliegendem Datentyp stark unterschiedlich zu implementieren wäre, wäre die Verwendung der *Services*-Schnittstelle angebracht. In diesem konkreten Fall ist allerdings kein so „schweres Geschütz“ erforderlich, denn aus der Sicht des Namensraumes ist der Typ eines Objekts, das lediglich als Referenz betrachtet, von dem aber keine Methode verwendet wird, praktisch unerheblich.

Das Mittel der Wahl ist hier die Delegation auf der Basis von Filterobjekten. Ein Filterobjekt ist ein Objekt, das Aufträge entweder selbst bearbeitet oder an ein angekoppeltes Objekt weiterleitet, je nachdem, ob es sie kennt oder nicht. Dadurch wirkt es insgesamt wie ein einziges Objekt mit einem erweiterten Sprachschatz.

Abbildung 3.6 zeigt die Methoden dieser Abstraktion:

Create erzeugt einen neuen Knoten, der das angegebene Objekt repräsentiert und Aufträge an es weiterleitet. Voraussetzung ist, daß für das Objekt nicht bereits ein solcher Knoten existiert. Dieser Knoten wird als gewöhnlicher Knoten ohne die Fähigkeit, Unterknoten zu erzeugen, angelegt. Soll der Knoten ein Objekt ohne weitere statische Merkmale darstellen, kann auch NIL angegeben werden. Zusätzliche Auftragsbearbeiter wären dann direkt an den Knoten anzuhängen.

Die Kopplung zwischen einem Filterobjekt und dem Objekt, an das delegiert wird, ist als besonders eng zu betrachten. Damit das repräsentierte Objekt nicht damit rechnen muß, seine Namen — etwa durch einen Aufruf von *Names.Destroy* von dritter Seite — unerwartet einzubüßen, wird dieser Zugriff auf den Knoten durch eine spezielle Autorisierung eingeschränkt. Das Filterobjekt kann jedoch jederzeit mit *NamedObjects.Destroy* wieder terminiert werden, vorausgesetzt, diese Methode wird im eigenen Prozeß aufgerufen. Letzteres wird durch die Verwendung von *VolatileShards* (siehe Abschnitt 2.3) sichergestellt.

Das Vorschalten eines Filterobjekts bewirkt zwangsläufig, daß die Bearbeitung von Aufträgen um mindestens eine Stufe indirekter wird. Anwendungen, die *NamedObjects* bewußt einsetzen, können sich dieses zusätzlichen Aufwands wieder entledigen, indem sie das hinter dem Filter liegende Objekt bei Bedarf hervorholen. Dazu dienen die Methoden *Get* und *GuardedGet*, die sich lediglich darin unterscheiden, ob eine Typenzusicherung stattfindet oder nicht. *GuardedGet* prüft, ob das Objekt den erwarteten Typ besitzt, bevor es eine Zuweisung vornimmt, und vermeidet so einen Laufzeitfehler und Programmabbruch, falls dies tatsächlich nicht der Fall sein sollte.

3.3.2 *Paths*

Namensräume in der Form, wie sie die Abstraktion *Names* definiert, lassen insofern etwas Komfort vermissen, als konkrete Namen stets über mehrere Zwischenschritte aufgebaut werden müssen und daher z.B. nicht als einfache Konstanten angegeben werden können.

Dieses „Manko" versucht das Modul *Paths* auszugleichen, indem es einige Hilfsfunktionen zur Umwandlung von einfachen Zeichenketten in Pfade und Namen bereitstellt. Konkret definiert es für jede *Names*-Methode mit einem Parameter vom Typ *Names.Name* zwei äquivalente Varianten mit einem entweder absoluten oder relativen Pfad in Form einer Zeichenkette. Abbildung 3.7 zeigt deren Definitionen.

Da einer Zeichenkette keine Ereignisse zugeordnet werden können, übernimmt jeweils ein *errors*-Argument die Rolle des Objekts, auf das sich eventuell auftretende Fehler beziehen. Innerhalb der durch ein Null-Byte begrenzten Zeichenkette gelten Schrägstriche als Trennzeichen zwischen Pfadkomponenten. Ein Pfad wird in der Weise aufgelöst, daß von *Names.root* bzw. dem angegebenen Knoten ausgehend für jede Komponente eine Kante gleichlautenden Namens durchlaufen wird, bis noch genau eine Komponente übrig ist. Der erreichte Knoten und die verbleibende Komponente bilden dann das Objekt und das Namensargument, mit dem die entsprechende Methode von *Names* aufgerufen wird. Für alle Autorisierungen wird das gleiche *auth*-Argument herangezogen. Sollte einer Kante aus irgendwelchen Gründen nicht gefolgt werden können, bricht der Vorgang vorzeitig ab.

In seiner jetzigen Fassung erspart das Modul *Paths* in vielen Fällen, wo Namen als gewöhnliche Zeichenketten vorliegen, eine gewisse Menge an Programmieraufwand. Allerdings wären noch Verbesserungen vorstellbar. So könnte z.B. die noch sehr simple, nicht allgemeingültige Art der Abbildung gewöhnlicher Zeichenketten auf Pfade surjektiv gemacht werden, indem für die Zeichen mit Sonderbedeutung eine Ersatzdarstellung eingeführt würde. Im Sinne größerer Flexibilität sollte *Paths*

```
PROCEDURE GetNode( path: ARRAY OF CHAR;
                   auth: Shards.Lid;
                   errors: RelatedEvents.Object;
                   VAR subnode: Names.Node): BOOLEAN;

PROCEDURE GetNodeN(node: Names.Node; path: ARRAY OF CHAR;
                   auth: Shards.Lid;
                   errors: RelatedEvents.Object;
                   VAR subnode: Names.Node): BOOLEAN;

PROCEDURE Insert(  path: ARRAY OF CHAR;
                   subnode: Names.Node; auth: Shards.Lid;
                   errors: RelatedEvents.Object): BOOLEAN;

PROCEDURE InsertN( node: Names.Node; path: ARRAY OF CHAR;
                   subnode: Names.Node; auth: Shards.Lid;
                   errors: RelatedEvents.Object): BOOLEAN;

PROCEDURE Delete(  path: ARRAY OF CHAR;
                   auth: Shards.Lid;
                   errors: RelatedEvents.Object): BOOLEAN;

PROCEDURE DeleteN( node: Names.Node; path: ARRAY OF CHAR;
                   auth: Shards.Lid;
                   errors: RelatedEvents.Object): BOOLEAN;
```

Abbildung 3.7: Methoden des Moduls *Paths*

diese Abbildung nicht selbst festlegen, sondern eine Schnittstelle dafür anbieten. Eine gute Standardimplementierung könnte sich, um einer denkbaren späteren Integration in heterogene Systeme Vorschub zu leisten, an einem weitverbreiteten Konzept wie den im *World Wide Web* gebräuchlichen *uniform resource identifiers* [Berners-Lee94] orientieren, deren Format auch darauf abgestimmt ist, Zeichen, die in anderen Kontexten Probleme bereiten könnten, möglichst zu vermeiden. Damit würde das an Oberon-Objekte gebundene Namenssystem eine komfortable und unproblematische Transportmöglichkeit für ungebundene Pfade hinzugewinnen.

3.3.3 *UniqueNames*

Bei verschiedenen Gelegenheiten entsteht ein Bedarf an unverwechselbaren, jedoch für sich allein genommen keine besondere Information tragenden Namen. Individualität hilft, jede Art von Konflikten zu vermeiden, die aus Verwechslungen resultieren würden. Indessen ist es von praktischer Relevanz, daß ein Unterscheidungsmerkmal nicht übermäßig redundant ausfällt — ein Kraftfahrzeugkennzeichen soll beispielsweise auf eine bestimmte Fläche passen und dennoch aus einigem Abstand noch ablesbar sein. Ein weiterer Aspekt, der von Interesse sein kann, ist die Streuung innerhalb des gegebenen Wertebereichs. Im Gegensatz zu

```
DEFINITION UniqueNames;

   IMPORT Names, Shards;

   PROCEDURE Insert(node, subnode: Names.Node; auth: Shards.Lid;
                    VAR name: Names.Name): BOOLEAN;

END UniqueNames.
```

Abbildung 3.8: Definition des Moduls *UniqueNames*

einer fortlaufenden Nummer kann ein scheinbar „zufällig" vergebener Name dazu benutzt werden, von einer de facto bestehenden zeitlichen Reihenfolge zu abstrahieren.

Das kleine Modul *UniqueNames*, dessen komplette Definition in Abbildung 3.8 dargestellt ist, realisiert Namen mit diesen drei Eigenschaften. Die Eindeutigkeit ergibt sich bereits aus der Abstraktion für Namensräume, denn *Names* gestattet das Einfügen weiterer Kanten, die von einem Knoten wegführen, nur mit noch nicht vorhandenen Namen. Da die Abstraktion auf parallele Benutzung eingerichtet ist, gibt es keine auch nur kurzzeitige Ungewißheit (*race condition*) bei der diesbezüglichen Entscheidung. *UniqueNames* „erfindet" notfalls einfach so oft einen anderen Namen, bis die Einfügeoperation klappt oder aus einem anderen Grund als dem der Duplizität abgelehnt wird.

Die erzeugten Namen sind Folgen von Klein- und Großbuchstaben. Um eine gute Streuung zu erzielen, werden Pseudozufallszahlen des Moduls *Random* eingesetzt; die Taktik bei Übereinstimmungen setzt sich aus dem Anfordern weiterer Pseudozufallszahlen und systematischen Veränderungen zusammen, um nicht ganz vom „Wohlverhalten" des Zufallsgenerators abhängig zu sein (obwohl dieser in der Praxis noch keinen Anlaß zu Beanstandungen gab). Die aktuelle Implementierung arbeitet mit Namen fixer Länge, was sicherlich aus einem theoretischen Blickwinkel beanstandet werden könnte, aber praxisgerecht ist, solange die gewählten Konstanten den zu gewärtigenden Größenordnungen angemessen sind.[6]

3.3.4 *RemoteNames*

Den Konventionen für die Unterstützung entfernter Objekte gemäß wird der Dienst, Objekte der Abstraktion *Names* exportierbar zu machen, in einem Modul *RemoteNames* realisiert.

[6]Einer Länge von sieben entsprechen z.B. über eine Billion Kombinationen. Anwendungen, die derart viele verschiedene Namen benötigen, wären wohl auch aus anderen Gründen gut beraten, sich dazu einer Struktur aus mehreren Knoten zu bedienen.

In der historischen Entwicklung der Oberon-Bibliothek wurden Module, die solche Dienstleistungen implementierten, im Laufe der Zeit immer einfacher und übersichtlicher, denn die besonderen Aufgabenstellungen, denen sie gerecht werden mußten, wurden mehr und mehr durch geeignete Abstraktionen in allgemein verwendbarer Form gelöst. Dementsprechend läßt sich das Modul *RemoteNames* heute direkt aus den entsprechenden Konventionen ableiten:

- Definition und Implementierung persistenter Aufträge für alle Methoden der Abstraktion,
- Definition des bzw. der Stellvertreter-Objekte und Implementierung der Methoden als Auftraggeber,
- Implementierung eines Auftragsbearbeiters, der die Methoden der Abstraktion aufruft,
- Implementierung der Konstruktoren für Stellvertreter-Objekte,
- (optional) Definition und Implementierung persistenter Verbunde als Parameter für Konstruktoren,
- Registrierung bei *RemoteObjects*.

Bei der Implementierung der Aufträge muß darauf geachtet werden, wie die beteiligten Datenstrukturen im einzelnen zu transferieren sind. Die Abstraktion *Names* definiert einen zentralen Objekttyp (*Names.Node*) und verwendet eine Anzahl weiterer Typen für Prozedurparameter. In Tabelle 3.2 sind die bei der Übermittlung von Aufträgen zu benutzenden Schreiboperationen für jeden dieser Typen zusammengestellt, woraus sich auf naheliegende Weise auch die passenden Leseoperationen ableiten lassen. Parameter des Typs *Names.Status* stellen einen Sonderfall dar, weil sie (noch) nicht von allein persistent sind. Sie bestehen aber aus persistenten Komponenten und können daher mit einer aus einzelnen Schreiboperationen zusammengesetzten Operation weitergegeben werden.

Man beachte, daß sowohl Iteratoren als auch Ereignistypen importiert und exportiert werden können. Ebenfalls von Bedeutung ist, daß *RemoteObjects* den Mechanismus unterstützt, auf den die Standardimplementierung von *Names* angewiesen ist, um veraltete Namen entfernen zu können, nämlich die Abstraktion *Resources*, durch die unter anderem Nachrichten über die Termination von Objekten verbreitet werden (vgl. Abschnitt 3.2.4). Als Konsequenz ergibt sich, daß Knoten, zu denen die Netzwerkverbindung endgültig abgebrochen ist, genau wie terminierte lokale Knoten mitsamt ihren Namen aus dem Namensraum verschwinden, und daß die Lebensdauer von Stellvertreter-Objekten die der Originalobjekte im Prinzip nicht überschreitet.

Da das Modul „im Verborgenen“ (als Dienstanbieter) arbeitet, ist sein Definitionsteil leer.

Datentyp	Transportmethode
BOOLEAN	*NetIO.WriteBoolean*
Names.AccessMode	*NetIO.WriteShortInt*
Names.CapabilitySet	*NetIO.WriteSet*
Names.Name	*NetIO.WriteConstString*
Shards.Pot	*PersistentObjects.WriteObjectOrNIL*
Shards.Lid	*PersistentObjects.WriteObjectOrNIL*
Names.Status	(zusammengesetzte Operation)
Iterators.Iterator	*RemoteObjects.Export*
Events.EventType	*RemoteObjects.Export*
Names.Node	*RemoteObjects.Export*

Tabelle 3.2: Transferoperationen für die Datentypen aus *Names*

3.3.5 Laufzeitunterstützung

Die Variable *Names.root* dient — wie bereits erwähnt — dazu, einzelnen Oberon-Programmen einen globaleren Namensraum als Kontext bereitzustellen. Unter anderem aus Gründen der Kontinuität werden Namensräume mit übergeordneter Bedeutung zumeist in eigens dazu bestimmten Dienstprogrammen, den Namensservern im engeren Sinn, etabliert werden. Andere Prozesse sollten in der Lage sein, mit einem für sie zuständigen und geeigneten Server Kontakt aufzunehmen, ohne auf festgelegte und daher unflexible Information zurückgreifen zu müssen.

Die Lösung, die für das Oberon-System unter UNIX gewählt wurde, sieht wie folgt aus: Aus einer Datei, deren Pfadname dem Laufzeitsystem bekannt ist, wird ein Oberon-Objekt des Typs *Names.Node* importiert. Es ist dies nicht etwa ein persistentes Objekt ungewisser Herkunft und Aktualität, sondern ein lebendes, von einem Namensserver exportiertes Objekt, falls überhaupt ein solcher existiert. Die von diesem Knoten ausgehenden Kanten sind dafür vorgesehen, in die Wurzel aller Oberon-Prozesse aufgenommen zu werden, die ihn erhalten haben. Eine solche Vorgehensweise wird dadurch möglich, daß der Export von Objekten kein synchroner bilateraler Vorgang ist, sondern durchaus auch zeitlich verzögerten Antworten mehrerer potentieller Adressaten gerecht werden kann. Erst beim Import wird ja die eigentliche Verbindung hergestellt.

Der Knoten aus dem lokalen bzw. netzgeographisch in der Nähe liegenden „Zwischenhändler“ spielt keine Rolle mehr, nachdem er seine Aufgabe, eine Konfiguration von Knoten naher und ferner anderer Server weiterzugeben, erfüllt hat. Die seitens des Klienten erforderlichen Operationen, damit diese fremden Objekte über Namen in *Names.root* angesprochen werden können, führt das Modul *UnixNames* bei seiner Initialisierung aus.

Abbildung 3.9, mit der dieses Kapitel endet, zeigt die Importhierarchie aller wichtigen mit Namensräumen zusammenhängenden Module.

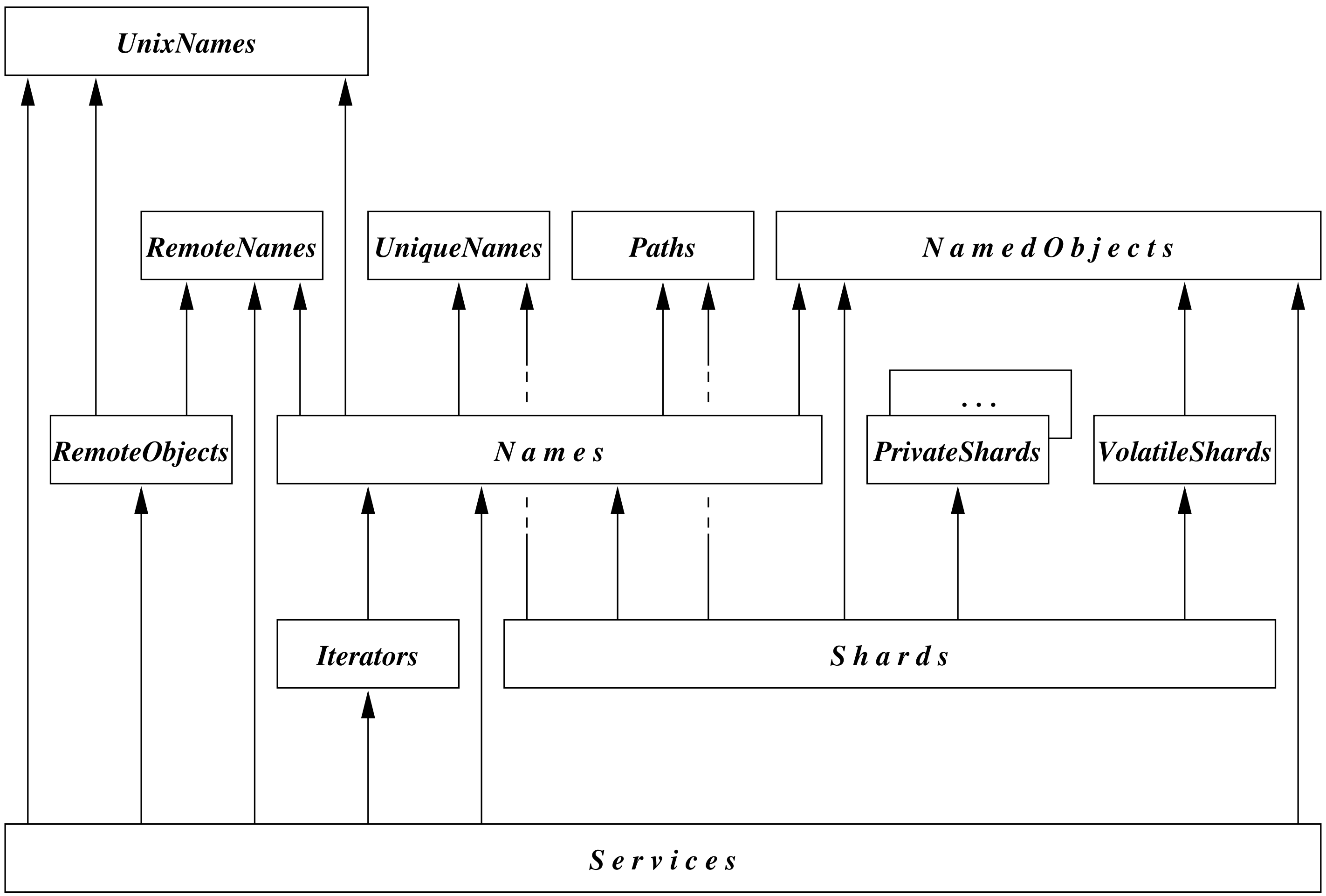

Abbildung 3.9: Modulhierarchie für Namensräume

Kapitel 4

Zusammenspiel verteilter Anwendungen

4.1 Namensräume als Informationsträger

Vorbemerkungen

Alle Beispiele dieses Kapitels ranken sich um ein kleines, auf einem verteilten Namensraum basierendes System für persönliche Mitteilungen, das eigens zu Demonstrationszwecken entwickelt wurde. Will man den Erfordernissen der Skalierbarkeit und der Wiederverwendbarkeit von Software Rechnung tragen, wird man sich naturgemäß nur solcher Techniken bedienen, die nicht auf einen bestimmten Systemumfang zugeschnitten sind. Im Kleinen zeigt sich, wenn schon nicht ihre wahre Leistungsfähigkeit, so doch wohl immerhin ihre Form und Art der Verwendung. Übertrieben realistische Komplexität wäre dem beabsichtigten instruktiven Charakter ja auch nicht förderlich, und schließlich wird immer erst etwas Zeit vergehen müssen, bis für eine neu entwickelte Bibliothek umfangreichere Anwendungen vorliegen.

Das ausgewählte Problem animiert, wie es sich erweisen wird, dazu, manche Eigenschaften der im vorherigen Kapitel eingeführten Namensräume auf eine vielleicht etwas ungewöhnliche Weise zu nutzen. Allerdings ist in jeder aus erweiterbaren Abstraktionen aufgebauten Bibliothek ein Potential für Nutzungsmöglichkeiten zu erwarten, an die ursprünglich keineswegs gedacht wurde.

Aufgabenstellung

Die konkrete Aufgabenstellung soll zunächst durch einige funktionelle Anforderungen umrissen werden:

- Gedacht ist an ein System, mit dem verschiedene Personen einander Mitteilungen zukommen lassen können.
- Die einzelnen Personen sollen sich jederzeit anmelden können, um Nachrichten zu lesen oder abzuschicken, und auch wieder abmelden, ohne daß ihre Nachrichten verloren gehen.
- Solange ein Benutzer mit dem System verbunden ist, kann er neue Mitteilungen sofort bemerken, wenn sie eintreffen.
- Die für einen bestimmten Empfänger gespeicherten Mitteilungen sollen nicht von Anderen gelesen werden können.
- Jede Nachricht kann direkt beantwortet werden, d.h., sie enthält einen Bezug auf den „Briefkasten“ des Absenders.

Lösungsansatz

Mit einer grundsätzlichen Entscheidung sei festgelegt, daß das logische Medium für Mitteilungen und Briefkästen ein Namensraum sein soll. Einige Charakteristiken der Lösung ergeben sich daraus fast von selbst. Ein Knoten in einem Namensraum kann von sich aus Daten aufnehmen, und zwar in Form der von ihm wegführenden Kanten. Ein Name für eine Kante ist bekanntlich eine beliebige Anzahl beliebiger Zeichen, kann also ohne weiteres aus dem Inhalt einer eingegebenen Mitteilung und einigen Zusatzinformationen bestehen. Also sollten Knoten als Briefkästen dienen können. Da jede Kante zu einem Knoten führt, bietet es sich sich an, hier genau den Absenderbriefkasten zu referenzieren. Mit geeignet zu vergebenden Berechtigungen kann sichergestellt werden, daß das Einfügen neuer Kanten (Mitteilungen) allgemein zugelassen, das Lesen und Löschen hingegen auf einen autorisierten Besitzer beschränkt wird.

Abbildung 4.1 verdeutlicht die Verhältnisse an einem Beispiel. Es zeigt ein von drei Personen, *Friedrich*, *Karl-Otto* und *Theodor*, genutztes Mitteilungssystem, nachdem Theodor eine Nachricht A an Karl-Otto und dieser eine Nachricht B an Friedrich und Theodor gesandt hat. Zu beachten ist, daß die Pfeile die Kanten des entstandenen Graphen und nicht etwa Richtungen des Datentransports wiedergeben.

Modulstruktur

Abbildung 4.2 zeigt eine sich ergebende Modulhierarchie, die im folgenden motiviert wird. Fest steht zunächst, daß es angebracht ist, das System zu verteilen, denn Sitzungen für Benutzer sollen mit ihnen kommen und gehen, die verwalteten Briefkästen und Mitteilungen dagegen eine Zeitlang resident bleiben. Zu

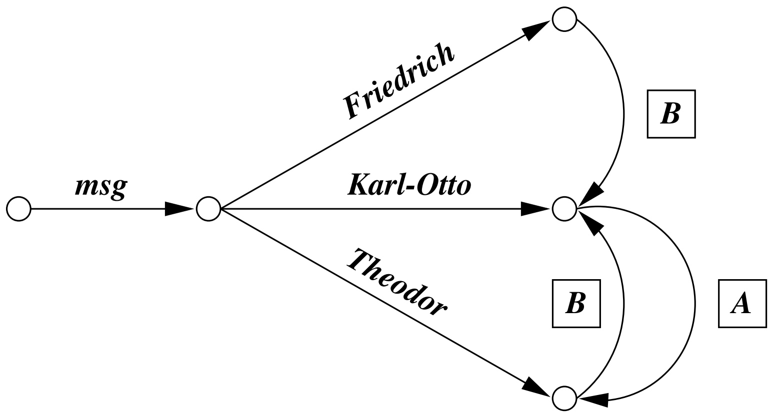

Abbildung 4.1: Beispiel für einen Graphen des Mitteilungs-Systems

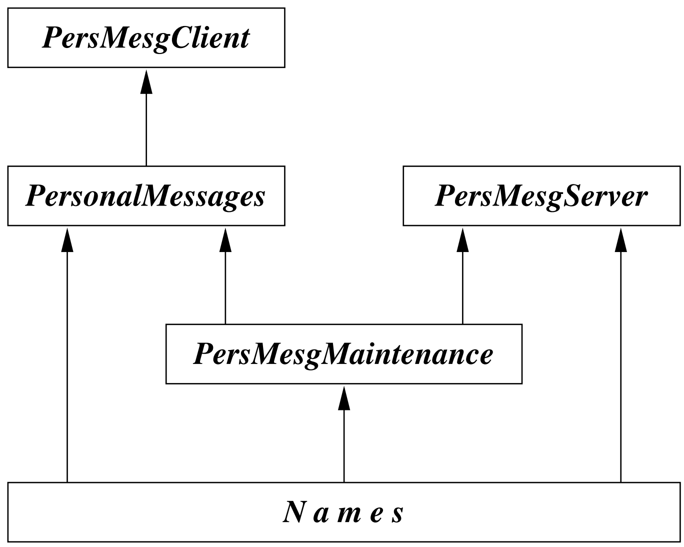

Abbildung 4.2: Modulhierarchie für das Mitteilungs-System

diesem Zweck sollten Knoten nicht nur, wie sonst üblich, lokal in jeweils dem Prozess angelegt werden, der sie hervorruft, sondern alternativ auch in besonders dazu bestimmten länger laufenden Serverprozessen. Eine Methode dafür realisiert das Modul *PersMesgMaintenance*.

Abgesehen von dieser besonderen Art, Knoten anzulegen, lassen sich alle denkbaren Operationen im Umgang mit Briefkästen und Mitteilungen auf Methoden von *Names* zurückführen. Die Aufgabe des Moduls *PersonalMessages* ist es, diese Details zu verbergen und die Operationen in einer problembezogenen Form zur Verfügung zu stellen, sodaß Programme mit Benutzerschnittstellen es möglichst einfach haben, darauf zurückzugreifen. *PersMesgClient* realisiert eine derartige Schnittstelle, während *PersMesgServer* ein Hauptprogramm für einen passiven, auf Aufträge von *PersMesgMaintenance* eingerichteten Namensserver darstellt.

4.2 Erzeugung von Knoten in fremdem Auftrag

Das Modul *PersMesgMaintenance*

Wenden wir uns zunächst dem Problem zu, in einem fremden Prozeß neue Knoten zu erzeugen. Die Lokation eines Objektes ist im Ulmer Oberon-System im Prinzip weder zu sehen, noch spielt sie für dessen Verwendung eine Rolle. Erst die möglichen Konsequenzen für die Lebensdauer des Objekts rücken diesen Aspekt doch wieder ins Blickfeld.

Im konkreten Fall ist es keineswegs notwendig, daß ein Objekt nach seiner Entstehung beliebig migrieren kann — was mit Persistenz zu erreichen wäre —, sondern nur, daß gesteuert werden kann, wo es stationär eingerichtet wird. Das ist nicht weiter schwierig, denn es wird ohne besondere Vorkehrungen genau dort bleiben, wo es erzeugt wird. Wenn dies also durch eine Methode eines anderen Objekts veranlaßt wird, teilt das neue Objekt anschließend dessen Adreßraum. Daher ist nichts weiter zu tun, als eine solche Methode zu realisieren, und zwar am zweckmäßigsten in Form eines zusätzlichen persistenten Auftrags an ein Objekt, das bereits exportierbar, d.h. mit Aufträgen aus anderen Prozessen erreichbar ist.

Wir wollen einen neuen Knoten, der uneingeschränkt neue Kanten aufnimmt und alle anderen Operationen durch eine bestimmte Berechtigung schützt, in einen Namensraum einfügen lassen. Es ist naheliegend, genau diesen Vorgang als zusätzliche Methode für den Knoten, von dem die Kante ausgehen soll, zu realisieren. Abbildung 4.3 zeigt die vollständige Definition des Moduls *PersMesgMaintenance*, das diese Methode implementiert. Weil nicht alle Knoten um diese Methode erweitert werden sollen, sondern kooperationswillige Server explizit einen ihrer Knoten dazu bestimmen sollen, wird für die Einrichtung des Auftragsbearbeiters eine eigene Prozedur angeboten, statt dies mit Hilfe von *Services* zu automatisieren.

```
DEFINITION PersMesgMaintenance;

   IMPORT
      Names, Shards;

   (* === server side: install service for a node === *)

   PROCEDURE InitServer(msgsys: Names.Node);

   (* === client side: request for a new message box === *)

   PROCEDURE SupplyBox( msgsys: Names.Node;
                        name: Names.Name; perm: Shards.Pot;
                        VAR box: Names.Node): BOOLEAN;

END PersMesgMaintenance.
```

Abbildung 4.3: Definition des Moduls *PersMesgMaintenance*

Realisierung

In diesem Abschnitt soll Schritt für Schritt die Implementierung des Moduls *PersMesgMaintenance* gezeigt werden. Machen wir uns zunächst klar, was die Methode *PersMesgMaintenance.SupplyBox* intern im Server eigentlich tun soll (ein eigener Name ist notwendig, um die lokale Version von der im gleichen Modul angesiedelten allgemeingültigen zu unterscheiden):

```
PROCEDURE DoSupplyBox(msgsys: Names.Node;
                      name: Names.Name; perm: Shards.Pot;
                      VAR box: Names.Node): BOOLEAN;
   VAR
      status: Names.Status;
      amode: Names.AccessMode;
BEGIN
   NEW(status);
   amode := 0;
   WHILE amode < Names.accessmodes DO
      status.perm[amode] := perm;
      INC(amode);
   END;
   Shards.CreateSimplePot(status.perm[Names.insert], TRUE);
   Names.CreateNode(box, status, Names.allCaps);
   IF Names.Insert(msgsys, box, name, NIL) THEN
      RETURN TRUE
   END;
   box := NIL;
   RETURN FALSE
END DoSupplyBox;
```

Der angeforderte Briefkasten wird als Standardknoten erzeugt, dessen sämtliche Berechtigungen durch den angegebenen Autorisierungsagenten bestimmt werden, mit Ausnahme der Zugriffsart *insert*, die uneingeschränkt zugelassen wird. Wenn sich eine auf diesen Knoten verweisende Kante in den Verwaltungsknoten einfügen läßt, wird die Funktion erfolgreich beendet, andernfalls der neue Briefkasten wieder gelöscht und Mißerfolg signalisiert. Es wäre üblich, dem Verwaltungsknoten dazu ein entsprechendes Fehlerereignis mitzugeben, was hier der Einfachheit halber ausgespart wird. Mindestens ein Ereignis wird im Fehlerfall auch bereits von *Names.Insert* beigesteuert.

Es folgen die Definition eines persistenten Auftragstyps und ein Bearbeiter, der die interne Implementierung aufruft. Man beachte die temporäre Umlenkung von Fehlerereignissen und die Art, wie das Funktionsresultat im Auftragsverbund vermerkt wird.

```
TYPE
   SupplyBoxMsg = POINTER TO SupplyBoxMsgRec;
   SupplyBoxMsgRec =
      RECORD
         (Messages.MessageRec)
         name: Names.Name;   (* IN *)
         perm: Shards.Pot;   (* IN *)
         box:  Names.Node;   (* OUT *)
      END;

PROCEDURE Handler(obj: Messages.Object;
                  VAR msg: Messages.Message);
BEGIN
   IF (msg IS SupplyBoxMsg) & (obj IS Names.Node) THEN
      WITH msg: SupplyBoxMsg DO
         WITH obj: Names.Node DO
            RelatedEvents.Save(obj);
            RelatedEvents.Forward(obj, msg.errors);
            msg.done :=
               DoSupplyBox(obj, msg.name, msg.perm, msg.box);
            RelatedEvents.Restore(obj);
            msg.processed := TRUE;
         END;
      END;
   END;
END Handler;
```

Die Konstruktor-, Schreib- und Leseprozeduren, die den Auftragsverbund persistent machen, *CreateSupplyBoxMsg*, *WriteSupplyBoxMsg* und *ReadSupplyBoxMsg*, ergeben sich ohne Schwierigkeiten aus der Typdefinition:

```
PROCEDURE CreateSupplyBoxMsg(VAR obj: PersistentObjects.Object);
   VAR
      msg: SupplyBoxMsg;
BEGIN
   NEW(msg);
   PersistentObjects.Init(msg, supplyBoxMsgType);
   Messages.Init(msg);
   obj := msg;
```

```
END CreateSupplyBoxMsg;

PROCEDURE WriteSupplyBoxMsg(s: Streams.Stream;
                          msg: PersistentObjects.Object): BOOLEAN;
BEGIN
   WITH msg: SupplyBoxMsg DO
      IF msg.processed THEN
         RETURN ~msg.done OR RemoteObjects.Export(s, msg.box)
      END;
      RETURN
         NetIO.WriteConstString(s, msg.name) &
         PersistentObjects.Write(s, msg.perm)
   END;
END WriteSupplyBoxMsg;

PROCEDURE ReadSupplyBoxMsg(s: Streams.Stream;
                          msg: PersistentObjects.Object): BOOLEAN;
BEGIN
   WITH msg: SupplyBoxMsg DO
      IF msg.processed THEN
         RETURN ~msg.done OR RemoteObjects.Import(s, msg.box)
      END;
      RETURN
         NetIO.ReadConstString(s, msg.name) &
         PersistentObjects.Read(s, msg.perm)
   END;
END ReadSupplyBoxMsg;
```

Beim Lesen und Schreiben der Komponenten wird anhand des Erledigungskennzeichens *msg.processed* die Transportrichtung unterschieden, um nur die jeweils relevanten Komponenten übertragen zu müssen. Der neue Knoten wird über *RemoteObjects* weitergegeben.

Man beachte, daß für das Versenden des Auftrags selbst weder in diesem Modul noch in *Messages* explizit Bezug auf *RemoteObjects* genommen wird; vielmehr ist es eine Eigenschaft von Stellvertreter-Objekten, Aufträge über Netzverbindungen hinweg zu delegieren (vgl. Abbildung 1.5, Seite 20). Sollte sich ein Auftrag zufällig an ein Objekt im eigenen Adreßraum richten, ist kein Stellvertreter-Objekt zwischengeschaltet und die Persistenz des Auftrags bleibt unerheblich; stattdessen aktiviert er direkt den lokalen Bearbeiter.

In *CreateSupplyBoxMsg* wird auf die Typvariable *supplyBoxMsgType* Bezug genommen, die global definiert ist und anfangs ein einziges Mal mit *PersistentObjects.RegisterType* initialisiert wird:

```
VAR
   supplyBoxMsgType: Services.Type;

PROCEDURE InitInterface;
   VAR
      if: PersistentObjects.Interface;
BEGIN
```

```
   NEW(if);
   if.create := CreateSupplyBoxMsg;
   if.read := ReadSupplyBoxMsg;
   if.write := WriteSupplyBoxMsg;
   if.createAndRead := NIL;
   PersistentObjects.RegisterType(supplyBoxMsgType,
      "PersMesgMaintenance.SupplyBoxMsg", "Messages.Message", if);
END InitInterface;
```

Nun sind alle Voraussetzungen gegeben, um die von *PersMesgMaintenance* exportierten Prozeduren zu implementieren:

```
PROCEDURE InitServer(msgsys: Names.Node);
BEGIN
   Messages.InstallHandler(msgsys, Handler);
END InitServer;

PROCEDURE SupplyBox( msgsys: Names.Node;
                     name: Names.Name; perm: Shards.Pot;
                     VAR box: Names.Node): BOOLEAN;
   VAR
      msg: SupplyBoxMsg;
BEGIN
   CreateSupplyBoxMsg(msg);
   msg.name := name;
   msg.perm := perm;
   Messages.Send(msgsys, msg);
   box := msg.box;
   RETURN msg.done;
END SupplyBox;
```

Das Serverprogramm

Damit andere Programme die neue Dienstleistung nutzen können, muß mindestens ein Serverprogramm diese zur Verfügung stellen, d.h. ein Objekt einrichten, das die Methode *PersMesgMaintenance.SupplyBox* versteht und für die potentiellen Klienten auch auffindbar ist. Für die besagte Methode vorbereitet wird ein Knoten bekanntlich mit der Prozedur *PersMesgMaintenance.InitServer*. Ein probates Mittel, ihn allgemein erreichbar zu machen, ist es, einen Namen zu vergeben, unter dem er in einen prozeßübergreifenden Namensraum eingebunden wird. Dies könnte im einfachsten Fall etwa so formuliert werden:

```
CONST
   myname = "/etc/msg";

PROCEDURE StartService;
   VAR
      msgsys: Names.Node;
      etype: Events.EventType;
BEGIN
   Names.CreateNode(msgsys, NIL, Names.allCaps);
   PersMesgMaintenance.InitServer(msgsys);
   IF ~Paths.Insert(myname, msgsys, NIL, msgsys) THEN
```

```
      Conclusions.Conclude(
         msgsys, Errors.fatal, "can't install msgsys")
   END;
END StartService;
```

Ein Prozeß muß seine Bereitschaft, mit anderen Prozessen in Kontakt zu treten, ausdrücklich bekunden, indem er einen oder mehrere Netzwerkanschlüsse als *ports* für *RemoteObjects* registriert. Dadurch wird unter anderem für jeden dieser Anschlüsse eine Task aufgesetzt, die auf Verbindungen wartet. Ein „Kochrezept“, um alle verfügbaren Verbindungsarten ausfindig und für *RemoteObjects* nutzbar zu machen, stellt die folgende Prozedur dar:

```
PROCEDURE OpenAllKindsOfPorts;
   VAR
      network: Networks.Network;
      it: Iterators.Iterator;
      port: Networks.Socket;
      address: Networks.Address;
      errors: RelatedEvents.Object;
BEGIN
   NEW(errors); RelatedEvents.QueueEvents(errors);
   Networks.GetNetworks(it);
   WHILE Iterators.Get(it, network) DO
      IF ~Networks.CreateSomeSocket(port, network, address,
    errors) OR
            ~RemoteObjects.AddPort(port, address, errors) THEN
         Conclusions.Conclude(errors, Errors.error, "open ports");
      END;
   END;
END OpenAllKindsOfPorts;
```

Die beiden gezeigten Prozeduren sind alles, was ein Programm aufzurufen braucht, um zu einem minimalen Server für unser Mitteilungs-System zu werden. Weil von *RemoteObjects* Tasks ins Leben gerufen werden, wird es auch nicht mehr von selbst terminieren. Zum guten Benehmen eines Servers gehörte allerdings ein Weiteres, und zwar sollte gerade seine Beendigung in angemessener Weise steuerbar sein. Naheliegend wäre es z.B., auf die Termination des exportierten Objektes mit der Termination des Prozesses als Ganzem zu reagieren. In der Prozedur *StartService* wäre dies etwa mit folgender Anweisungsfolge zu erreichen:

```
   Resources.TakeInterest(msgsys, etype);
   Events.Handler(etype, ExitHandler);
```

Hierbei bezeichnet *etype* eine Variable vom Typ *Events.EventType* und steht *ExitHandler* für einen geeigneten Ereignisbearbeiter, auf den hier aber nicht näher eingegangen wird. Es sei lediglich angemerkt, daß die von *RemoteObjects* erzeugten auf neue Verbindungen wartenden Tasks mit *RemoteObjects.RemoveAllPorts* und die anderen, für bestehende Verbindungen zuständigen Tasks dieses Moduls mit *RemoteObjects.CloseAllConnections* wieder beendet werden können.

4.3 Verteilte Briefkasten-Objekte

Das Modul *PersonalMessages*

Im folgenden geht es um den begrifflichen Kern des Mitteilungs-Systems, das Modul *PersonalMessages*, welches die Sicht der Anwendungsprogramme auf das System realisiert. Es führt als Abstraktion drei Objektklassen ein:

- Der Objekttyp *MessageBox* abstrahiert einen Zugang einer Person zu ihrem Briefkasten, ähnlich wie *UnixFiles.Stream* einen Zugang zu einer Datei, nicht jedoch die Datei selbst symbolisiert.
- Instanzen von *Message* sind die eigentlichen Mitteilungen. Sie können u.a. sequentiell von den Briefkasten-Objekten bezogen werden.
- *MessageEvent* ist der Datentyp von Ereignissen, die zur asynchronen Benachrichtigung über den Empfang neuer Mitteilungen dienen.

Abbildung 4.4 stellt diese Typen in der Form dar, wie sie von anderen Modulen gesehen werden. *MessageBoxRec* und *MessageRec* haben keine öffentlichen Komponenten. Ihre Struktur bleibt in der Implementierung verborgen. Bezüge auf Namensräume und *ConstStrings* werden sogar im ganzen Definitionsteil vermieden, sodaß Anwendungen dieses Moduls nur mit gewöhnlichen Zeichenketten umzugehen brauchen.

Intern handelt es sich beim Versenden von Mitteilungen um nichts anderes als das Einfügen von Kanten in Knoten, die den Briefkasten des jeweiligen Empfängers repräsentieren. Die Kante selbst wird aus einem Präfix, welches Datum und Uhrzeit der Erzeugung sowie den Namen des Absenders enthält, und der eigentlichen Mitteilung gebildet. Durch das Präfix wird bei alphabetisch sortierten Kanten nebenbei die chronologische Reihenfolge beim Abholen der Mitteilungen sichergestellt. Bevor wir exemplarisch auf einige Einzelheiten der Implementierung eingehen, wollen wir die Definitionen der Methoden betrachten.

Abbildung 4.5 enthält die Methoden zum Eröffnen und Beenden des Zugangs zu Briefkasten-Objekten. Mit *Open* wird ein neues Objekt angelegt, falls noch keines dieses Namens vorhanden war, oder versucht, ein vorhandenes wiederzueröffnen. Letzteres hat nur Erfolg, wenn das Paßwort mit demjenigen übereinstimmt, mit dem das Objekt ursprünglich angelegt wurde. Mit *Close* oder *Kill* wird die Beziehung zu dem Objekt wieder beendet, wobei *Kill* zusätzlich das Objekt löscht. Durch das Löschen gehen sämtliche Mitteilungen verloren, die von diesem Briefkasten aus abgeschickt wurden, denn zu ihm führende Kanten werden ja ungültig. Die gewählte Datenrepräsentation sorgt also dafür, daß nur Mitteilungen mit gültigem Absender aufbewahrt werden.

```
TYPE
   MessageBox = POINTER TO MessageBoxRec;
   MessageBoxRec =
      RECORD
         (Disciplines.ObjectRec)
      END;

   Message = POINTER TO MessageRec;
   MessageRec =
      RECORD
         (Disciplines.ObjectRec)
      END;

   MessageEvent = POINTER TO MessageEventRec;
   MessageEventRec =
      RECORD
         (Events.EventRec)
         box: MessageBox;
         msg: Message;           (* newly arrived message *)
      END;
```

Abbildung 4.4: Typdefinitionen des Moduls *PersonalMessages*

```
PROCEDURE Open(   VAR box: MessageBox;
                  name, password: ARRAY OF CHAR): BOOLEAN;

PROCEDURE Reopen( box: MessageBox);      (* restart reading *)

PROCEDURE Close(  box: MessageBox);

PROCEDURE Kill(   box: MessageBox);      (* close & destroy *)
```

Abbildung 4.5: Methoden zum Eröffnen, Schließen und Löschen von Briefkasten-Objekten

```
PROCEDURE Get(    box: MessageBox; VAR msg: Message): BOOLEAN;

PROCEDURE Remove( box: MessageBox; msg: Message);

PROCEDURE Watch(  box: MessageBox;
                  VAR eventType: Events.EventType);
```

Abbildung 4.6: Methoden zum Abholen und Löschen von Mitteilungen

```
PROCEDURE Create( VAR msg: Message;
                  from: MessageBox; text: ARRAY OF CHAR);

PROCEDURE Extract(msg: Message;
                  VAR header, text: ARRAY OF CHAR);
```

Abbildung 4.7: Methoden zum Erzeugen und Konvertieren von Mitteilungen

```
PROCEDURE Send(   msg: Message; to: ARRAY OF CHAR): BOOLEAN;

PROCEDURE Reply(  msg, ref: Message): BOOLEAN;
```

Abbildung 4.8: Methoden zum Versenden von Mitteilungen

```
PROCEDURE Users(  VAR list: UserList);

PROCEDURE GetUser(list: UserList;
                  VAR user: ARRAY OF CHAR): BOOLEAN;

PROCEDURE Known(  user: ARRAY OF CHAR): BOOLEAN;
```

Abbildung 4.9: Methoden zum Bestimmen möglicher Adressaten

Die Methoden zum Abholen und Löschen von Mitteilungen zeigt Abbildung 4.6. Nachdem ein Briefkasten-Objekt eröffnet wurde, stehen die schon vorhandenen Mitteilungen zum sequentiellen Abholen mit *Get* zur Verfügung. Das Resultat von *Get* ist FALSE, wenn schon alle Nachrichten gelesen worden sind. Der Lesezyklus kann jederzeit mit *Reopen* von vorn begonnen werden. Nicht mehr gebrauchte Mitteilungen können mit *Remove* aus dem Briefkasten entfernt werden. Falls auf das Eintreffen neuer Mitteilungen direkt reagiert werden soll, können mit *Watch* entsprechende Ereignisse angefordert werden.

Zum Umwandeln von Zeichenketten in die abstrakte Form der *PersonalMessages.Message* und wieder zurück dienen die in Abbildung 4.7 dargestellten Methoden *Create* und *Extract*. Das Präfix mit dem Zeitstempel usw. wird bei *Create* automatisch erzeugt; *Extract* liefert es in *header* getrennt vom Rest der Mitteilung zurück.

Für das Versenden einer Mitteilung gibt es zwei Varianten. Die Methoden dafür zeigt Abbildung 4.8. Wenn es sich um eine Rückantwort handelt, braucht nur die zu beantwortende Mitteilung als zweites Argument von *Reply* angegeben zu werden. Den allgemeinen Fall mit einem anhand seines Namens zu suchenden Empfänger realisiert *Send*.

Vor dem Anlegen eines neuen Briefkastens und dem Senden einer Nachricht kann es interessant sein, die bereits bekannten Namen abzufragen. Dazu dienen die drei

```
PROCEDURE OpenS(  VAR box: MessageBox;
                  server,
                  name, password: ARRAY OF CHAR): BOOLEAN;

PROCEDURE SendS(  msg: Message;
                  server, to: ARRAY OF CHAR): BOOLEAN;

PROCEDURE UsersS( server: ARRAY OF CHAR; VAR list: UserList);

PROCEDURE KnownS( server, user: ARRAY OF CHAR): BOOLEAN;

PROCEDURE Default(server: ARRAY OF CHAR);
```

Abbildung 4.10: Erweiterte Methoden für die Integration mehrerer Server

Methoden von Abbildung 4.9. *Known* testet, ob ein angegebener Name bekannt ist. *Users* fordert einen Iterator an, der mit *GetUser* der Reihe nach die Namen der bekannten Benutzer liefert.

Die bisher genannten Methoden setzen voraus, daß stets derselbe Serverknoten anzusprechen ist, sonst wäre ein zusätzlicher Parameter erforderlich. In einem weit verteilten System sollte dieser Fall allerdings nicht ausgeschlossen werden, denn es ist dann sicherlich zweckmäßig, mehrere Server zu integrieren. Abbildung 4.10 zeigt, für welche Methoden eine Variante mit einem Server-Parameter existieren sollte. Um im Schema der gewöhnlichen Zeichenketten zu bleiben, empfiehlt sich die Bezeichnung des Servers nach der Syntax des Moduls *Paths*. Die Varianten ohne Server-Parameter könnten einen mit *Default* einstellbaren Server benutzen.

Anmerkungen zur Realisierung

Zur Realisierung und internen Datenstruktur des Moduls *PersonalMessages* ist folgendes anzumerken:

- Mitteilungen sind Kanten, die in einen oder mehrere Empfängerknoten eingefügt werden und zum Absenderknoten führen. Sie werden also als eine *Names.Name*-Zeichenkette und ein Knotenobjekt dargestellt. Da der Absendername und ein Zeitstempel in den Kantennamen einbezogen werden, ist nicht damit zu rechnen, daß verschiedene Mitteilungen zu einem Duplizitätskonflikt führen. Sollte hingegen versucht werden, die gleiche Mitteilung mehrfach zuzustellen, etwa wegen Redundanz in einer Verteilerliste, hätte genau dieser Konflikt den positiven Effekt, die Vervielfachung der Daten zu verhindern.

- Briefkasten-Objekte sind Knoten zugeordnet. Neben diesem Knoten benötigt jeder Briefkasten für die abgehenden Nachrichten den Namen, unter dem er eröffnet wurde; ferner sein spezielles Autorisierungsobjekt für alle *Names*-Operationen außer *Insert* und einen Iterator, der seine Kantenmenge traversiert.

Wenn ein Ereignistyp für ankommende Nachrichten angefordert wird, wird für den Knoten *Names.TakeInterest* aufgerufen. Da im Prinzip von mehreren Seiten auf denselben Knoten zugegriffen werden kann, Ereignisse aber individuell für einzelne dieser *MessageBox*-Zuordnungen gelten sollen, wird für jede offene Briefkasten-Verbindung ggf. ein eigener Ereignistyp eingesetzt. Ein Bearbeiter des originalen *Names*-Ereignisses filtert die *insert*-Fälle heraus und leitet sie genau an die interessierten *MessageBox*-Objekte in Form ihres individuellen Ereignistyps weiter. Welche Objekte interessiert sind, wird durch eine dem *Names*-Ereignistyp zugeordnete Disziplin festgehalten. Wenn ein *MessageBox*-Objekt terminiert, muß es aus dieser Menge wieder entfernt werden. Mit einem Verweis beim Objekt auf die Disziplin und seinen dortigen Listeneintrag kann die Verwaltung der Liste mit konstantem Aufwand erfolgen.

- In den folgenden Beispielen wurde die Fehlerbehandlung weggelassen oder sehr einfach gehalten. Ferner wurde auf konsequenten Schutz kritischer Regionen und penible Plausibilitätskontrollen verzichtet.

Die internen Definitionen von *PersonalMessages.MessageBox* und den dazu gehörenden Datenstrukturen sehen wie folgt aus:

```
TYPE
   BoxesList = POINTER TO BoxesListRec;
   Discipline = POINTER TO DisciplineRec;

   MessageBox = POINTER TO MessageBoxRec;
   MessageBoxRec =
      RECORD
         (Disciplines.ObjectRec)
         node:  Names.Node;
         name:  Names.Name;
         auth:  Shards.Lid;
         msgs:  Iterators.Iterator;
         etype: Events.EventType;
         disc:  Discipline;
         entry: BoxesList;
      END;

   BoxesListRec =
      RECORD
         box: MessageBox;
         prev, next: BoxesList;
      END;

   DisciplineRec =
      RECORD
```

```
         (Disciplines.DisciplineRec)
         boxes: BoxesList;
      END;

VAR
   default: Names.Node;
   defaultKey: Resources.Key;
   discId: Disciplines.Identifier;
```

Daraus ergibt sich, wie in *PersonalMessages.Open* vorzugehen ist: Ein neues Briefkasten-Objekt anlegen und Name und Autorisierungsobjekt initialisieren. Wenn das entsprechende Knotenobjekt nicht erhältlich ist (was darauf hindeutet, daß es den Knoten noch nicht gibt), einen solchen mit passender Berechtigung vom Server anfordern. Bei Mißerfolg aufgeben. Wenn von dem alten (oder neuen) Knoten die Kantenmenge nicht erhältlich ist, deutet das auf ein unkorrektes Paßwort hin, was wiederum Mißerfolg bedeutet. *Resources* wird veranlaßt, (Terminations-) Ereignisse, die den Knoten betreffen, an das Briefkasten-Objekt weiterzugeben. Ferner wird ein Bearbeiter für diese Ereignisse vereinbart, der dafür zuständig ist, terminierte Briefkasten-Objekte aus der Liste der Interessenten an einem Ereignistyp wieder herauszunehmen:

```
PROCEDURE Open(VAR box: MessageBox;
                  name, password: ARRAY OF CHAR): BOOLEAN;
   VAR
      pw: Names.Name;
      perm: Shards.Pot;
      queue: RelatedEvents.Queue;
      etype: Events.EventType;
BEGIN
   NEW(box);
   ConstStrings.Create(box.name, name);
   ConstStrings.Create(pw, password);
   PasswordShards.CreateLid(box.auth, pw);
   IF ~Names.GetNode(default, box.name, NIL, box.node) THEN
      (* create new node for message box *)
      PasswordShards.CreatePot(perm, pw);
      IF ~PersMesgMaintenance.SupplyBox(default,
            box.name, perm, box.node) THEN
         Conclusions.Conclude(default, Errors.error, name);
         box := NIL;
         RETURN FALSE
      END;
   END;
   IF ~Names.GetMembers(box.node, box.auth, box.msgs) THEN
      RelatedEvents.GetQueue(box.node, queue);      (* ignore *)
      box := NIL;
      RETURN FALSE
   END;
   box.etype := NIL;
   box.disc := NIL;
   box.entry := NIL;
   Resources.DependsOn(box, box.node);
   Resources.TakeInterest(box, etype);
   Events.Handler(etype, ResourcesEventHandler);
```

```
   RETURN TRUE
END Open;
```

Die Methoden *Close* und *Kill* sind fast trivial:

```
PROCEDURE Close(box: MessageBox);
BEGIN
   Resources.Notify(box, Resources.terminated);
END Close;

PROCEDURE Kill(box: MessageBox);
BEGIN
   IF ~Names.Destroy(box.node, box.auth) THEN
      Conclusions.Conclude(box.node, Errors.error, "destroy box");
   END;
END Kill;
```

Ein wenig interessanter ist die Methode *Watch*. Es muß auf jeden Fall darauf geachtet werden, daß der Bearbeiter für *Names*-Ereignisse pro Ereignistyp höchstens ein Mal vereinbart wird. Wir stellen dies sicher, indem wir diesen Vorgang mit dem Neuanlegen der Disziplin für den Ereignistyp verbinden. Unabhängig davon wird die Aufnahme des Knotens in die Datenstruktur der Disziplin, eine doppelt verkettete Liste, gehandhabt. Das Briefkasten-Objekt gilt genau dann bereits als Glied dieser Liste, wenn die Komponente *entry* nicht gleich NIL ist. Sie zeigt dann auf den entsprechenden Eintrag.

```
PROCEDURE Watch(box: MessageBox; VAR eventType: Events.EventType);
   VAR
      namesEventType: Events.EventType;
      this: BoxesList;
BEGIN
   IF box.etype = NIL THEN
      Events.Define(box.etype);
   END;
   IF box.disc = NIL THEN
      IF Names.TakeInterest(box.node, box.auth,
            namesEventType) THEN
         IF ~Disciplines.Seek(namesEventType, discId,
               box.disc) THEN
            NEW(box.disc);
            box.disc.id := discId;
            box.disc.boxes := NIL;
            Disciplines.Add(namesEventType, box.disc);
            Events.Handler(namesEventType, NamesEventHandler);
         END;
      ELSE
         Conclusions.Conclude(box.node, Errors.error, "watch");
      END;
   END;
   IF (box.disc # NIL) & (box.entry = NIL) THEN
      NEW(this);
      this.box := box;
      this.prev := NIL;
      this.next := box.disc.boxes;
```

```
      box.disc.boxes := this;
      IF this.next # NIL THEN
         this.next.prev := this;
      END;
      box.entry := this;
   END;
   eventType := box.etype;
END Watch;
```

Die übrigen Methoden des Moduls sind relativ unproblematisch aus den Definitionen abzuleiten. Exemplarisch seien hier die Methoden *Send* und *Reply* herausgegriffen. Zunächst die interne Typdefinition von *PersonalMessages.Message*:

```
TYPE
   Message = POINTER TO MessageRec;
   MessageRec =
      RECORD
         (Disciplines.ObjectRec)
         from:    Names.Node;
         content: Names.Name;
      END;

PROCEDURE Send(msg: Message; to: ARRAY OF CHAR): BOOLEAN;
   VAR
      name: Names.Name;
      node: Names.Node;
BEGIN
   ConstStrings.Create(name, to);
   IF ~Names.GetNode(default, name, NIL, node) THEN
      Conclusions.Conclude(default, Errors.error, "recipient");
      RETURN FALSE
   END;
   IF ~Names.Insert(node, msg.from, msg.content, NIL) THEN
      Conclusions.Conclude(node, Errors.error, "send");
      RETURN FALSE
   END;
   RETURN TRUE
END Send;

PROCEDURE Reply(msg, ref: Message): BOOLEAN;
BEGIN
   IF ~Names.Insert(ref.from, msg.from, msg.content, NIL) THEN
      Conclusions.Conclude(ref.from, Errors.error, "send");
      RETURN FALSE
   END;
   RETURN TRUE
END Reply;
```

Es sei noch einmal daran erinnert, daß die Fehlerbehandlung hier nur angedeutet wird. Anstatt an Ort und Stelle Fehlermeldungen auszugeben, sollten spezifische Ereignisse generiert und zusammen mit denen aus verschachtelten Aufrufen an die Umgebung zurückgegeben werden.

Zum Abschluß der zitierten Codefragmente sei noch ein Beispiel für den korrekten Umgang mit „gewichtigen“ Referenzen angeführt. Die globale Variable *default* stellt hier eine solche Referenz dar. Das hat die Methode, mit der sie bestimmt wird, zu berücksichtigen:

```
PROCEDURE Default(server: ARRAY OF CHAR);
   VAR
      errors: RelatedEvents.Object;
BEGIN
   NEW(errors);
   RelatedEvents.QueueEvents(errors);
   IF default # NIL THEN
      Resources.Detach(default, defaultKey);
   END;
   IF ~Paths.GetNode(server, NIL, errors, default) THEN
      Conclusions.Conclude(errors, Errors.error, server);
   END;
   Resources.Attach(default, defaultKey);
END Default;
```

Benutzerschnittstelle

Ein Blick zurück auf die Modulhierarchie (Abschnitt 4.2, Seite 57) zeigt uns, daß die Benutzerschnittstelle *PersMesgClient* nur Methoden aus der Abstraktion *PersonalMessages* benutzt. Namensräume oder gar entfernte Objekte im Allgemeinen sind auf dieser Ebene völlig transparent. Mithin beschränkt sich die Aufgabe dieses Moduls auf die Ein- und Ausgabe von Zeichenketten und einen herkömmlichen Dialog zur Steuerung der verschiedenen Funktionen. Beispiele hierzu sind allenthalben zu finden und sollen daher an dieser Stelle keinen breiteren Raum einnehmen.

4.4 Resumé

Die Beispiele dieses Kapitels waren nicht dazu gedacht, zu beweisen, daß alles, was hier erarbeitet wurde, so und nicht anders sein mußte. Wer eine Softwarebibliothek entwickelt, sollte viel größeren Wert auf Einfachheit und Vielseitigkeit der zur Verfügung gestellten Hilfsmittel legen, als darauf, daß ihnen ihr Verwendungszweck eindeutig anzusehen ist. Erfahrungen mit diesen und anderen Komponenten haben ergeben, daß es sehr attraktiv sein kann, pro Abstraktion nur wenige Eigenschaften festzulegen, um umso mehr Gewinn aus den verschiedenen möglichen Kombinationen zu ziehen. So sind im Ulmer Oberon-System etwa Exportierbarkeit und Persistenz in gewisser Weise analoge und doch voneinander unabhängige Konzepte. Sie treten kombiniert auf bei den persistenten Sichten, die eine erstaunliche neue Funktionalität einführen, und sie haben eine gemeinsame Wurzel bei den Diensten, die zwar für diesen Kontext entwickelt wurden, aber plötzlich auch völlig anderen Anwendungen zugute kommen.

In diesem Licht scheint es gerechtfertigt, einen größeren Aufwand für die Beschäftigung mit Grundlagen zu treiben, als „praktische Erfordernisse“ zuweilen sinnvoll erscheinen lassen wollen. Auch die vorliegende Arbeit tendiert weniger zu fertigen Lösungen als in die Richtung möglichst solider Ausgangspunkte — bei allen Zugeständnissen, die einem ein zeitlich begrenzter Rahmen und die natürliche „Lernkurve“ beim Umgang mit einem sich rasch weiterentwickelnden System abnötigen.

Das soll nicht heißen, daß die Praxis außer acht gelassen würde. Vielmehr liefern reale Gegebenheiten und Analogien immer wieder die Motivation zu neuen Begriffsbildungen und Mechanismen des Zusammenwirkens verschiedener Komponenten. Letztlich steht und fällt gute Software mit der Schärfe der Problemanalyse und Ausdruckskraft der Spezifikation.

Die nächsten Schritte bei der Entwicklung von Namensräumen und daran ansetzenden Protokollen werden jedoch eindeutig näher zum Konkreten führen. Einer ersten Generation experimenteller Namensserver werden leistungsfähige Dienstprogramme folgen, die Persistenz, Sicherheit und schnelle Algorithmen in den Dienst verteilter Oberon-Anwendungen stellen werden. Namensräume werden die Rolle von portablen Schnittstellen zu vielerlei verwandten mehr oder minder stark systemabhängigen Bereichen annehmen, unter anderem den immer wieder ärgerlich uneinheitlichen Dateisystemen. Schließlich weisen auch Beispiele wie das *World Wide Web* auf ein besonderes Potential, das Namensräumen zu eigen ist hin: die Fähigkeit zur Integration heterogener Umgebungen.

Anhang A

Manualseiten

Ulm’s Oberon Library: Shards(3)

NAME

Shards — abstraction for authorization protocols

SYNOPSIS

```
TYPE Pot = POINTER TO PotRec;
TYPE PotRec = RECORD (PersistentDisciplines.ObjectRec) END;
TYPE Lid = POINTER TO LidRec;
TYPE LidRec = RECORD (PersistentDisciplines.ObjectRec) END;
TYPE FitsProc = PROCEDURE (pot: Pot; lid: Lid): BOOLEAN;
TYPE SupplyProc = PROCEDURE (pot: Pot; VAR lid: Lid): BOOLEAN;
TYPE Interface = POINTER TO InterfaceRec;
TYPE InterfaceRec =
   RECORD
      (Objects.ObjectRec)
      fits: FitsProc;
      supply: SupplyProc; (* may be NIL *)
   END;

PROCEDURE Fits(pot: Pot; lid: Lid): BOOLEAN;
PROCEDURE Supply(pot: Pot; VAR lid: Lid): BOOLEAN;
PROCEDURE Init(pot: Pot; if: Interface);
PROCEDURE CombineLids(VAR lid: Lid; lid1, lid2: Lid);
PROCEDURE CreateSimplePot(VAR pot: Pot; permissive: BOOLEAN);
PROCEDURE CreateSomeLid(VAR lid: Lid);
```

DESCRIPTION

Shards come in two types, *Pot* and *Lid*, to serve as tokens in authorization protocols. A pot is an object that can tell whether a given lid object fits to it. Some pot objects can also supply new lids guaranteed to fit to them.

Both of the most trivial cases of pots (either accepting any or no lid at all) can be obtained using *CreateSimplePot*.

CreateSomeLid creates a lid that is likely not to fit to any pot other than trivial pots of the permissive kind. By convention, **NIL** given as a lid serves the same purpose.

CombineLids creates a new lid fitting to any pot at least one of *lid1* or *lid2* fits to. Note that *lid1* will always be considered first.

Implementors of other shard types must initialize their pots using *Init*. The *if.fits* method will be called with lids of any type, but not **NIL**. If *if.supply* is **NIL**, *Supply* will always return **FALSE**.

SEE ALSO

PrivateShards(3)	standard nontrivial shards implementation[1]
PersistentObjects(3)	abstraction for persistent objects

AUTHOR

Martin Hasch, University of Ulm

[1] cf. p. 78

Ulm's Oberon Library: PrivateShards(3)

NAME

PrivateShards — standard nontrivial shards implementation

SYNOPSIS

```
TYPE Pot = POINTER TO PotRec;
TYPE PotRec = RECORD (Shards.PotRec) END;
TYPE Lid = POINTER TO LidRec;
TYPE LidRec = RECORD (Shards.LidRec) END;

PROCEDURE Create(VAR pot: Shards.Pot; VAR lid: Shards.Lid);
```

DESCRIPTION

PrivateShards provides pairs of pots and lids intended to fit exactly to each other but to none else. Each call of *Create* yields a different pot. Uniqueness, however, can not be granted for pots created by different processes, although even in that case coincidences are very improbable.

SEE ALSO

Shards(3)	abstraction for authorization protocols[1]
PersistentObjects(3)	abstraction for persistent objects
Random(3)	pseudo random number generation

[1] cf. p. 76

Ulm's Oberon Library: VolatileShards(3)

NAME

VolatileShards — not quite persistent shards

SYNOPSIS

```
TYPE Pot = POINTER TO PotRec;
TYPE PotRec = RECORD (Shards.PotRec) END;
TYPE Lid = POINTER TO LidRec;
TYPE LidRec = RECORD (Shards.LidRec) END;

PROCEDURE Create(VAR pot: Shards.Pot; VAR lid: Shards.Lid);
```

DESCRIPTION

VolatileShards implements a shard type that stays valid only within a single process, because lids get changed on transmission via *PersistentObjects.Read/Write*.

Create returns in *pot* a pot all original lids of this module fit to. Original are lids obtained by *Create* or *Shards.Supply*, but not *PersistentObjects.Read*.

SEE ALSO

Shards(3)	abstraction for authorization protocols[1]
PersistentObjects(3)	abstraction for persistent objects

AUTHOR

Martin Hasch, University of Ulm

[1] cf. p. 76

Ulm's Oberon Library: Names(3)

NAME

Names — abstraction for name hierarchies

SYNOPSIS

```
TYPE Name = ConstStrings.String;
TYPE Node = POINTER TO NodeRec;
TYPE NodeRec = RECORD (Services.ObjectRec) END;
TYPE Lister = Iterators.Iterator;

CONST examine = 0;
CONST change = 1;
CONST read = 2;
CONST search = 3;
CONST insert = 4;
CONST delete = 5;
CONST destroy = 6;
CONST accessmodes = 7;
TYPE AccessMode = SHORTINT;        (* [examine..destroy] *)
TYPE Permissions = ARRAY accessmodes OF Shards.Pot;
TYPE Status = POINTER TO StatusRec;
TYPE StatusRec =
   RECORD
      (Objects.ObjectRec)
      perm: Permissions;
   END;

TYPE AccessProc = PROCEDURE (node: Node;
   mode: AccessMode; auth: Shards.Lid): BOOLEAN;
TYPE GetStatusProc = PROCEDURE (node: Node;
   auth: Shards.Lid; VAR status: StatusRec): BOOLEAN;
TYPE SetStatusProc = PROCEDURE (node: Node;
   status: Status; auth: Shards.Lid): BOOLEAN;
TYPE GetMembersProc = PROCEDURE (node: Node;
   auth: Shards.Lid; VAR members: Lister): BOOLEAN;
TYPE TakeInterestProc = PROCEDURE (node: Node;
   auth: Shards.Lid; VAR eventType: Events.EventType): BOOLEAN;
TYPE GetNodeProc = PROCEDURE (node: Node;
   name: Name; auth: Shards.Lid; VAR subnode: Node): BOOLEAN;
TYPE InsertProc = PROCEDURE (node,
   subnode: Node; name: Name; auth: Shards.Lid): BOOLEAN;
TYPE DeleteProc = PROCEDURE (node: Node;
   name: Name; auth: Shards.Lid): BOOLEAN;
TYPE DestroyProc = PROCEDURE (node: Node;
   auth: Shards.Lid): BOOLEAN;

TYPE Interface = POINTER TO InterfaceRec;
```

```
TYPE InterfaceRec =
   RECORD
      (Objects.ObjectRec)
      access:           AccessProc;
      getStatus:        GetStatusProc;
      setStatus:        SetStatusProc;
      getMembers:       GetMembersProc;
      takeInterest:     TakeInterestProc;
      getNode:          GetNodeProc;
      insert:           InsertProc;
      delete:           DeleteProc;
      destroy:          DestroyProc;
   END;

CONST accessCap = 0;
CONST statusCap = 1;
CONST domainCap = 2;
CONST destroyCap = 3;
CONST allCaps = {accessCap .. destroyCap};
CONST strictCaps = {statusCap, domainCap};
TYPE CapabilitySet = SET;          (* of [accessCap..destroyCap] *)

CONST inserted = insert;
CONST deleted = delete;
TYPE State = SHORTINT;             (* inserted or deleted *)
TYPE MemberEvent = POINTER TO MemberEventRec;
TYPE MemberEventRec =
   RECORD
      (Events.EventRec)
      state: State;
      name: Name;                  (* of inserted/deleted member *)
      node: Node;
   END;

VAR root: Node;

TYPE ErrorCode = SHORTINT;
TYPE ErrorEvent = POINTER TO ErrorEventRec;
TYPE ErrorEventRec =
   RECORD
      (Events.EventRec)
      errorcode: ErrorCode;
   END;
VAR errormsg: ARRAY errorcodes OF Events.Message;
VAR error: Events.EventType;

PROCEDURE InitNode(    node: Node;
                       if: Interface; caps: CapabilitySet);
PROCEDURE CreateNode(  VAR node: Node;
                       status: Status; caps: CapabilitySet);
PROCEDURE Error(errors: RelatedEvents.Object; code: ErrorCode);

PROCEDURE Capabilities(node: Node): CapabilitySet;
PROCEDURE Access(      node: Node;
```

```
                         mode: AccessMode; auth: Shards.Lid): BOOLEAN;
PROCEDURE GetStatus(   node: Node;
                         auth: Shards.Lid;
                         VAR status: StatusRec): BOOLEAN;
PROCEDURE SetStatus(   node: Node;
                         status: Status; auth: Shards.Lid): BOOLEAN;
PROCEDURE GetMembers(  node: Node;
                         auth: Shards.Lid;
                         VAR members: Lister): BOOLEAN;
PROCEDURE TakeInterest(node: Node;
                         auth: Shards.Lid;
                         VAR eventType: Events.EventType): BOOLEAN;
PROCEDURE GetNode(     node: Node;
                         name: Name; auth: Shards.Lid;
                         VAR subnode: Node): BOOLEAN;
PROCEDURE Insert(      node,
                         subnode: Node; name: Name;
                         auth: Shards.Lid): BOOLEAN;
PROCEDURE Delete(      node: Node;
                         name: Name; auth: Shards.Lid): BOOLEAN;
PROCEDURE Destroy(     node: Node; auth: Shards.Lid): BOOLEAN;
```

DESCRIPTION

Names defines and basically implements an abstraction for globally distributable name spaces for objects. Name spaces are directed (and in general cycle-free) graphs. Nodes in the "root" hierarchy as defined in this module can belong to other processes (name servers). Access to individual nodes can be controlled by use of an authorization scheme.

Nodes can have a combination of following capabilities:

accessCap — indicates that an explicit access method is present, so that results of *Access* (see below) can be considered authoritative;

statusCap — indicates that status information is maintained;

domainCap — denotes a node's capability to manage subnodes (members);

destroyCap — indicates that an explicit destroy method is present.

For convenience, *allCaps* defines a set of all capabilities, and *strictCaps* defines a set of those capabilities that will cause failures of associated methods when missing.

A *status*, if maintained, contains authorization agents (of type *Shards.Pot*) controlling the use of following methods:

examine	*GetStatus*
change	*SetStatus*
read	*GetMembers*, *TakeInterest*
search	*GetNode*
insert	*Insert*
delete	*Delete*
destroy	*Destroy*

Procedures of general interest

Capabilities returns capabilities of *node*.

Access tries to guess whether *node* would allow a particular access mode *mode*. When in doubt, e.g. because *node* has no *access* capability, result is **TRUE**.

GetStatus examines the current status of *node*. *SetStatus* changes the status of *node*.

GetMembers reads from *node* all names of current subnodes.

TakeInterest returns an event type for events notifying changes (additions or deletions) of members. Different nodes will yield different event types. Notification events carry a reference to the domain node whose set of members has changed and the name of the member.

GetNode searches *node* for a direct subnode named *name*.

Insert inserts name *name* for subnode *subnode* into *node*. *Delete* deletes name *name* for an existing subnode of *node*.

Destroy terminates *node* and deletes references to it wherever possible.

Example

This is an example of a procedure printing the names of all direct subnodes of a given node to a stream, supposed they are publicly visible:

```
PROCEDURE PrintMembers(node: Names.Node; s: Streams.Stream);
   VAR
      members: Iterators.Iterator;
      member: Names.Name;
BEGIN
   IF Names.domainCap in Names.Capabilities(node) THEN
      IF Names.GetMembers(node, NIL, members) THEN
         WHILE Iterators.Get(members, member) DO
            ConstStrings.Write(s, member); Write.LnS(s);
```

```
            END;
         ELSE
            Conclusions.Conclude(node, Errors.error, "PrintMembers");
         END;
      END;
END PrintMembers;
```

Procedures for implementors

Implementations other than the standard implementation must initialize new instances of nodes using *InitNode*, specifying a set of capabilities *caps* and an interface *if* containing following procedures:

If *statusCap* **IN** *caps*:

getStatus: **PROCEDURE**(*node*: *Node*; *auth*: *Shards.Lid*; **VAR** *status*: *StatusRec*);
: copy node's status information into *status*.

setStatus: **PROCEDURE**(*node*: *Node*; *status*: *Status*; *auth*: *Shards.Lid*);
: set node's status information to *status*.

If *domainCap* **IN** *caps*:

getMembers: **PROCEDURE**(*node*: *Node*; *auth*: *Shards.Lid*; *members*: *Lister*);
: return in *members* an iterator that will supply all names of node's direct subnodes.

takeInterest: **PROCEDURE**(*node*: *Node*; *auth*: *Shards.Lid*; **VAR** *eventType*: *Events.EventType*);
: return in *eventType* a type for events notifying changes of members of *node*.

getNode: **PROCEDURE**(*node*: *Node*; *name*: *Name*; *auth*: *Shards.Lid*; **VAR** *subnode*: *Node*);
: return in *subnode* node's subnode named *name*.

insert: **PROCEDURE**(*node*: *Node*; *subnode*: *Node*; *name*: *Name*; *auth*: *Shards.Lid*);
: insert subnode *subnode* by name *name* into node.

delete: **PROCEDURE**(*node*: *Node*; *name*: *Name*; *auth*: *Shards.Lid*);
: delete name *name* from node.

If *destroyCap* **IN** *caps*:

destroy: **PROCEDURE**(*node*: *Node*; *auth*: *Shards.Lid*);
check authorization to destroy node; clean up if appropriate.

CreateNode can be used to create a node of standard behaviour. In that case, *status* may be **NIL** if no restrictions are required.

Implementors are free to share the builtin *Error* procedure to raise error events as defined below.

DIAGNOSTICS

All procedures return **TRUE** on success, otherwise **FALSE**. Errors are related to nodes and queued on default; their priority is *Priorities.liberrors*.

Following error codes are currently implemented:

noPermission	operation was not permitted
badAccessMode	illegal access mode
noStatus	node does not maintain a status
noDomain	node is not a domain
nodeNotFound	node could not be found
nodeDestroyed	node has been destroyed
nameExists	name is already in use
nodeBusy	node busy, possible deadlock situation
cyclicLink	operation would cause a cyclic link (not detected by some implementations)
serverFailure	name server failure
unknownError	*Error* was called with unknown error code

SEE ALSO

ConstStrings(3)	handling of arbitrarily long strings
Iterators(3)	sequential access of data structures
Paths(3)	resolving string constants into name paths
Services(3)	type-independent definition of extensions

RelatedEvents(3)	error event handling
Resources(3)	object termination handling
Shards(3)	abstraction for authorization protocols[1]

AUTHOR

Martin Hasch, University of Ulm

[1] cf. p. 76

Ulm's Oberon Library: NamedObjects(3)

NAME

NamedObjects — add Names(3) functionality to arbitrary objects

SYNOPSIS

```
TYPE Object = Services.Object;
TYPE ErrorEvent = POINTER TO ErrorEventRec;
TYPE ErrorEventRec =
   RECORD
      (Events.EventRec)
      errorcode: SHORTINT;
   END;
VAR errormsg: ARRAY errorcodes OF Events.Message;
VAR error: Events.EventType;

PROCEDURE Create(VAR node: Names.Node; object: Object);
PROCEDURE Destroy(node: Names.Node): BOOLEAN;
PROCEDURE Get(node: Names.Node; VAR object: Object;
                     errors: RelatedEvents.Object): BOOLEAN;
PROCEDURE GuardedGet(node: Names.Node; guard: Services.Type;
                     VAR object: Object;
                     errors: RelatedEvents.Object): BOOLEAN;
```

DESCRIPTION

NamedObjects allows to supply arbitrary extensions of *Services.Object* with functionality of *Names.Node* by way of attaching filter objects.

Create creates a node representing *object*, i.e., messages sent to *node* will reach *object*, too. If *object* is **NIL** *Create* creates a node representing itself.

In order not to introduce a new termination method for the original object via *Names.Destroy*, destroy permission of *node* is especially restricted. A filter object *node* can however be revoked with *Destroy*.

Get assigns *object* the object which is represented by *node*. *GuardedGet* does the same, but only if the object's type is an extension of *guard*.

DIAGNOSTICS

All procedures with **BOOLEAN** results return **TRUE** on succes, otherwise **FALSE**. Following error codes are implemented:

noObject	node is not representing a named object
objectNamed	object is already represented by another node
typeGuardFailure	*GuardedGet* failed due to type guard violation

SEE ALSO

Names(3)	abstraction for name hierarchies[1]
Services(3)	type-independent definition of extensions
Shards(3)	abstraction for authorization protocols[2]
RelatedEvents(3)	error event handling

BUGS

Does not yet use *Forwarders(3)*, for historical reasons.

AUTHOR

Martin Hasch, University of Ulm

[1] cf. p. 80
[2] cf. p. 76

Ulm's Oberon Library: Paths(3)

NAME

Paths — resolving string constants into name paths

SYNOPSIS

```
CONST separator = "/";
TYPE Path = POINTER TO PathRec;
TYPE PathRec =
   RECORD
      (Objects.ObjectRec)
      component: Names.Name;
      next: Path;
   END;
TYPE ErrorCode = SHORTINT;
TYPE ErrorEvent = POINTER TO ErrorEventRec;
TYPE ErrorEventRec =
   RECORD
      (Events.EventRec)
      errorcode: ErrorCode;
   END;
VAR errormsg: ARRAY errorcodes OF Events.Message;
VAR error: Events.EventType;

PROCEDURE Convert(path: ARRAY OF CHAR): Path;
PROCEDURE GetNode(path: ARRAY OF CHAR;
  auth: Shards.Lid; errors: RelatedEvents.Object;
  VAR subnode: Names.Node): BOOLEAN;
PROCEDURE GetNodeN(node: Names.Node; path: ARRAY OF CHAR;
  auth: Shards.Lid; errors: RelatedEvents.Object;
  VAR subnode: Names.Node): BOOLEAN;
PROCEDURE Insert(path: ARRAY OF CHAR; subnode: Names.Node;
  auth: Shards.Lid; errors: RelatedEvents.Object): BOOLEAN;
PROCEDURE InsertN(node: Names.Node; path: ARRAY OF CHAR;
  subnode: Names.Node;
  auth: Shards.Lid; errors: RelatedEvents.Object): BOOLEAN;
PROCEDURE Delete(path: ARRAY OF CHAR;
  auth: Shards.Lid; errors: RelatedEvents.Object): BOOLEAN;
PROCEDURE DeleteN(node: Names.Node; path: ARRAY OF CHAR;
  auth: Shards.Lid; errors: RelatedEvents.Object): BOOLEAN;
```

DESCRIPTION

Paths provides an interface to *Names(3)* resolving string constants into name paths. For each procedure in *Names(3)* with an argument of type *Names.Node* and another one of type *Names.Name* there are two equivalent procedures in *Paths* with either an absolute or a relative path instead. In all cases *path* is a **0X**-terminated array of characters containing path components. Within *path*, two consecutive *separator* characters are replaced by a single one, whereas a single *separator* character is replaced by nothing, separating path components.

Paths are evaluated from left to right as a sequence of *Names.GetNode* calls until exactly one component remains unresolved, which becomes the *name* argument to the requested procedure.

Convert simply converts a string to a *Path* structure.

EXAMPLE

```
success := Paths.Insert("etc/example", subnode, auth, errors)
```

is essentially equivalent to:

```
ConstStrings.Create(etcString, "etc");
ConstStrings.Create(exampleString, "example");
success :=
   Names.GetNode(Names.root, etcString, auth, tmpnode) &
   Names.Insert(tmpnode, subnode, exampleString, auth)
```

except for the redirection of error events.

DIAGNOSTICS

All procedures return **TRUE** on success, otherwise **FALSE**. Error events, usually raised in the context of *Names(3)* procedures, are forwarded to *errors*. If not stated otherwise, their priority is *Priorities.liberrors*. In case of an inappropriate *path* argument an error event of type *ErrorEvent* is related to *errors* with following error code and message:

zeroPathLen	insert/delete need a nonempty path

SEE ALSO

Names(3)	abstraction for name hierarchies[1]
ConstStrings(3)	handling of arbitrarily long strings
Services(3)	type-independent definition of extensions
RelatedEvents(3)	error event handling

[1]cf. p. 80

AUTHOR

Martin Hasch, University of Ulm

Ulm's Oberon Library: UniqueNames(3)

NAME

UniqueNames — supply unique names

SYNOPSIS

```
PROCEDURE Insert(node, subnode: Names.Node; auth: Shards.Lid;
                     VAR name: Names.Name): BOOLEAN;
```

DESCRIPTION

Insert is a variant of *Names.Insert* that chooses some unique name by itself and supplies it in *name* for later reference.

DIAGNOSTICS

All error conditions and reactions are the same as with *Names(3)*, with the exception of *Names.nameExists* which is avoided by definition (in the worst case after some retries).

SEE ALSO

Names(3)	abstraction for name hierarchies[1]
Random(3)	pseudo random number generation

AUTHOR

Martin Hasch, University of Ulm

[1] cf. p. 80

Ulm’s Oberon Library: RemoteNames(3)

NAME

RemoteNames — network interface for Names(3)

SYNOPSIS

```
(* empty definition *)
```

DESCRIPTION

RemoteNames is a service provider (see *RemoteObjects(3)* and *Services(3)*) supporting export and import of nodes defined by *Names(3)*. There are no extra features and thus no publicly visible identifiers.

SEE ALSO

Names(3)	abstraction for name hierarchies[1]
RemoteObjects(3)	access of remote objects
Services(3)	type-independent definition of extensions

AUTHOR

Martin Hasch, University of Ulm

[1] cf. p. 80

Anhang B

Literatur

[Berners-Lee94] Timothy Berners-Lee, *Universal Resource Identifiers in WWW—A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*, Internet „Request for Comments“ (RFC) document no. 1630, June 1994.

[Birrell82] Andrew D. Birrell, Roy Levin, Roger M. Needham, Michael D. Schroeder, *Grapevine: An Exercise in Distributed Computing*, in: Communications of the ACM, Vol. 25, No. 4, pp. 260–274, April 1982.

[Borchert94] Andreas F. Borchert, *Zur Erweiterung von Programmiersprachen durch Bibliotheken*, Dissertation, Universität Ulm, Mai 1994.

[Borchert95a] Andreas F. Borchert, *Report of Ulm's Oberon Compiler*, in: Ulm's Oberon System Reference, Fifth Edition, ed. by Andreas Borchert, pp. 36–45, Universität Ulm, April 1995.

[Borchert95b] Andreas F. Borchert et al., *Manual Pages*, in: Ulm's Oberon System Reference, Fifth Edition, ed. by Andreas Borchert, Universität Ulm, April 1995.

[Clark82] David D. Clark, *Name, Addresses, Ports and Routes*, Internet „Request for Comments“ (RFC) document no. 814, July 1982.

[Lupper94] Alfred Lupper, *Namensverwaltung und Adressierung in Distributed Shared Memory-Systemen*, Ulmer Informatik-Berichte Nr. 94–15, Fakultät für Informatik, Universität Ulm, Dezember 1994.

[Needham93] Roger M. Needham, *Names*, in: Sape J. Mullender (ed.), Distributed Systems, second edition, pp. 315–327, Addison-Wesley Publishing Company, 1993.

[Pike93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom, *The Use of Name Spaces in Plan 9*, in: ACM Operating Systems Review, vol. 27, no. 4, pp. 72–76, New York, 1993.

[Satyanarayanan93] Mahadev Satyanarayanan, *Distributed File Systems*, in: Sape J. Mullender (ed.), Distributed Systems, second edition, pp. 353–383, Addison-Wesley Publishing Company, 1993.

[Szczuka95] Michael Szczuka, *Implementierung kryptographischer Mechanismen in die Bibliothek einer objekt-orientierten Sprache*, Diplomarbeit, Universität Ulm, 1995.

[Traub94] Stefan Traub, *The Design of a Distributed Oberon System*, in: Advances in Modular Languages, Proceedings of the Joint Modular Languages Conference 1994, ed. by Peter Schulthess, pp. 189–200, Universitätsverlag Ulm, September 1994.

[White76] James E. White, *A high-level framework for network-based resource sharing*, in: Proceedings of the National Computer Conference, pp. 561–570, 1976.

[Wirth88] Niklaus Wirth, *The Programming Language Oberon*, in: Software—Practice and Experience, vol. 18, no. 7, pp. 671–690, July 1988.

[Wirth92] Niklaus Wirth, Jürg Gutknecht, *Project Oberon—The Design of an Operating System and Compiler*, Addison-Wesley, New York, 1992.

Verzeichnis der Abbildungen und Tabellen

Abbildungen

Tabellen

Dank

Viele Personen haben durch Rat und Tat, ideellen oder materiellen Beistand das Zustandekommen dieser Arbeit gefördert. Dank gebührt Herrn Prof. Dr. Franz Schweiggert und besonders Herrn Dr. Andreas Borchert für die unermüdliche Betreuung, sowie Herrn Prof. Dr. Helmuth Partsch, dem zweiten Gutachter, ferner (in alphabetischer Reihenfolge) Andrea Ballhause, Elisabeth Böckel, Claus Drechsler, Gerda Drechsler, Peter Gschwendtner, Gabi Hybel, Norbert Hybel, Klaus Schumacher, Margarete Schumacher, Jan Stibane, Ernst Wilken und allen, die dem Autor in der Zeit der „heißen Phase“ mit Verständnis begegneten.

Erklärung

Name: Martin Hasch　　　　　　　　Matrikel-Nr.: 0117572

Ich erkläre, daß ich die Diplomarbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den 30. Juli 1996